APPLYING ASPECT-ORIENTED SOFTWARE DEVELOPMENT TO MIDDLEWARE FRAMEWORKS

TAL COHEN

APPLYING ASPECT-ORIENTED SOFTWARE DEVELOPMENT TO MIDDLEWARE FRAMEWORKS

RESEARCH THESIS

IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE
DEGREE OF DOCTOR OF PHILOSOPHY

TAL COHEN

SUBMITTED TO THE SENATE OF
THE TECHNION — ISRAEL INSTITUTE OF TECHNOLOGY

ADAR, 5767 HAIFA FEBRUARY 2007

THIS RESEARCH THESIS WAS DONE UNDER THE SUPERVISION OF DR. JOSEPH (YOSSI) GIL IN THE DEPARTMENT OF COMPUTER SCIENCE

I would like to thank the many people who have directly or indirectly helped me during the long period of research. I thank my supervisor, Yossi Gil, who never ceased to amaze me; my friend and co-student Itay Maman, who greatly helped in large parts of this research; Prof. Shmuel Katz, who first introduced me to the world of aspect-oriented programming; Omer Barkol, who was a wonderful office-mate these last few years; Prof. Ron Pinter, who advised and encouraged; and lastly, Yardena Kolet, the final authority in the CS faculty.

This work is dedicated to my family: to my parents, Rachel and Nahum; to my beloved wife Irit; and to my two wonderful children, Stav and Yuval: May you always have reasons for joy!

THE GENEROUS FINANCIAL HELP OF THE TECHNION IS GRATEFULLY ACKNOWLEDGED.

Contents

List of Figures

List of Tables

Abstract

This work presents a suite of related programming constructs and technologies aimed at integrating *aspect-oriented programming* (AOP) in the middleware frameworks used for enterprise application development. These constructs include *shakeins* (and the ASPECTJ2EE language design based on it), *JTL* (the Java Tools Language), *factories*, and *object evolution*.

Shakeins are a novel programming construct which, like mixins and generic classes, generates new classes from existing ones in a universal, uniform, and automatic manner: Given a class, a shakein generates a new class which has the same type as the original, but with different data and code implementation. This thesis argues that shakeins are restricted, yet less chaotic, aspects. It further claims that shakeins are well suited for the introduction of AOP into existing middleware applications.

To demonstrate the applicability of shakeins to the middleware framework domain, we introduce the ASPECTJ2EE language which, with the help of shakeins and a new *deploy-time* weaving mechanism, brings the blessings of AOP to the enterprise JAVA (J2EE) framework. A unique advantage of ASPECTJ2EE, which is less general (and hence less complicated) than ASPECTJ, is that it can be smoothly integrated into J2EE implementations without breaking their architecture.

Any aspect-oriented language or framework must provide developers with a mechanism for specifying a set of join points, i.e., program locations that should be modified by relevant aspects. Such "pointcut" specifications are commonly expressed using queries written in a dedicated, declarative language. We present *JTL*, the Java Tools Language, as a new query language. JTL provides a rich and powerful query-by-example mechanism which minimizes the *abstraction gap* between queries and the program elements they match. We further show how JTL can be extended to support *program transformations*, going as far as making JTL an AOP language in its own right.

Factories are presented as a new mechanism for controlling object instantiation, overcoming anomalies that can be found in the construction mechanisms of JAVA, C++, EIFFEL and similar object-oriented languages. Factories (not to be confused with the FACTORY METHOD or ABSTRACT FACTORY design patterns) provide classes with a complete control over their instantiation mechanism. In particular, classes can enforce the application of shakeins to all instances, without disturbing existing client code.

Finally, we allow shakeins to behave as *dynamic aspects*, i.e., aspects that can be applied to an object or removed from it at runtime. Because applying a shakein to a class generates a new class, we find that this implies the ability to change an object's class at runtime—i.e., object reclassification is required. As it turns out, reclassification using shakeins is part of a more general concept, which we call *Object Evolution*. Object evolution is a restriction of general reclassification by which dynamic changes to an object's class are *monotonic*: an object may gain, but never lose, externally-visible properties. We argue that there are many applications of monotonic evolution in practical systems. The monotonicity property makes it easier to maintain static type safety with object evolution than in general object reclassification.

Chapter 1

Introduction

There's no business like show business, but there are several businesses like accounting.

— David Letterman

Multivac [15] and Shalmaneser [43] are but two examples of a recurring motif in 20th-century science fiction: omnipotent computers that manage the everyday life of people. Although this vision is yet to materialize, it is clear today that software systems do control many aspects of our lives. *Enterprise applications*—the large-scale software programs used to operate and manage large organizations—run government agencies, banks, insurance companies, financial institutes, hospitals, etc. Their effect on our lives cannot be underestimated. Dependability, correctness, maintainability, and other such just causes are not worn-out buzzwords in the context of enterprise applications: they are an essential, even crucial, requirement of these software monsters. *This dissertation is about better tools for developing enterprise applications*, and better serving these causes—perhaps even contributing to a smoother run of these clockworks ticking behind the scenes of modern society.

An inherent paradox of enterprise applications is that they are so *illusively* simple. An enterprise application that runs a hospital is mostly concerned with the boring organization of records and moving of information around, with little, if any, clever algorithmic manipulation of the data. Likewise, the computations carried out in a banking application are almost always simple: calculating interest rates, or moving funds from one account to another.

For example, given a class `Account` that represents a bank account, how complex can be a method such as `transfer`, in charge of transferring funds from this account to another? The following code fragment[1] shows the entire relevant *business logic*:

```
void transfer(Account other, float amount) {
   if (balance < amount)
      throw new InsufficientFundsException();
   balance -= amount;
   other.balance += amount;
}
```

. . . totalling in exactly four imperative statements.

Yet in practice, developing enterprise applications is a daunting task, due to *orthogonal requirements* presented by most of these applications, including uncompromising reliability demands, unyielding security requirements, and complete trustworthiness that the applications must

[1] Here and henceforth, all code is in the JAVA [14] programming language, unless otherwise stated.

exhibit. This is why a method such as `transfer` will probably look a lot more like the code in Figure 1.1. Comprised of over a score of statements, even this version is greatly simplified compared with real enterprise application code.

```
1  void transfer(Account other, float amount) {
2     SecurityContext ctx = Security.getContext();
3     User user = ctx.getCurrentUser();
4     if (user == null) { // verify user is logged−in
5        Log.logSecurityViolation("Invalid transfer attempt.");
6        throw new SecurityException("No user is logged in.");
7     }
8     if (!user.canTransferFrom(this)) { // check credentials
9        Log.logSecurityViolation("Invalid transfer attempt.");
10       throw new SecurityException("Operation not allowed.");
11    }

13    Log.log("Transferring from " + this + " to " + other);
14    Log.log("Amount transferred: " + amount);

16    // Start the transaction (also locks the DB)
17    Transaction tx = getDbConnection().beginTransaction();

19    refreshFieldsFromDb(); // fetch most current field values (including balance)
20    other.refreshFieldsFromDb(); // ... for both objects

22    if (balance < amount) {
23       Log.logError("Transfer not possible");
24       tx.rollback(); // Abort transaction
25       throw new InsufficientFundsException();
26    }
27    balance -= amount;
28    other.balance += amount;

30    updateDbRecord(); // store updated balance in DB
31    other.updateDbRecord(); // ... for both objects

33    // Commit the transaction (and unlock the DB)
34    tx.commit();

36    Log.log("Transfer complete");
37 }
```

Figure 1.1: A `transfer` method catering to real-world considerations such as logging, security, transactions and persistence (simplified)

Examining the figure, we see that the business logic in the method is dwarfed by code that deals with *non-functional concerns*, including:

- Security (lines 2–11),
- Logging (lines 5, 9, 13, 14, 23 and 36),

- Data persistence (lines 19, 20, 30 and 31), and
- Transaction management (lines 17, 24 and 34).

The various concerns are *tangled* in a manner that bravely resists any attempt to separate them into distinct, clear-cut methods; and the code for handling each non-functional concern ends up being *scattered* across multiple, unrelated program modules. For example, a change of the transaction management policy to prevent nested transactions requires updates to numerous business methods such as `transfer`, rather than an update to a single program module.

Matters are further complicated by the fact that enterprise applications are often based on a heterogeneous platform configuration, connecting various independent systems (called *tiers*) into a coherent whole. The various tiers of an enterprise application may include, e.g., legacy mainframe servers, dedicated database servers, personal computers, departmental servers, and more.

The considerable complexity inherent in the development of enterprise applications, combined with the staggering demand for their rapid development, initiated a series of component-based *middleware architectures*, such as CORBA [221] and DCOM [172]. *Middleware frameworks*, used to implement such architectures, try to simplify enterprise application development by providing developers with various pre-defined *services*, such as security or load-balancing. These services are often externally applied to the code, helping untangle the various concerns to a certain extent. Modern middleware frameworks, such as *Java 2, Enterprise Edition* (J2EE) [203][2], provide a rich set of such services, including most of the non-functional concerns discussed above, and then some. These services are managed entirely by the framework; the programmer uses configuration files to direct their behavior, and they are hardly (or not at all) reflected in the program source code, which now focuses on business logic proper.

An alternative take on the problem of untangling and simplifying complex applications is the recently-introduced methodology known as *Aspect-Oriented Programming* (AOP) [151]. AOP allows developers to encapsulate the code relevant to any distinct non-functional concern in *aspect* modules. First and foremost in aspect-oriented technologies is the programming language ASPECTJ [150], an aspect-oriented extension of JAVA, though several other AOP languages and tools are also in use [34, 45, 174, 207]. The immense complexity of enterprise software makes it an appealing target for mass application of aspect-oriented development techniques [121, 158].

While it seems that we now have two solutions to a single problem, it turns out that neither solution, in its current form, is sufficient. The motivation for this work arises from the following double argument:

1. *The middleware framework approach has only limited and inflexible support for functional decomposition.* In cases where the canned solution is insufficient, application developers resort again to a tangled and highly scattered implementation of cross-cutting concerns. Part of the reason is that current middleware frameworks do not employ AOP in their implementation, and do not enable developers to decompose new non-functional concerns that show up during the development process.

   Section 1.1.1 elaborates on this claim.

2. Conversely, *the approach taken by currently-available aspect-oriented tools does not scale up to the demands of enterprise applications.* AOP languages such as ASPECTJ were conceived, designed, and implemented with the belief that the composition of a system from its aspects is an *implicit* process, carried out by some clever engine which should address the nitty-gritty details. One of the chief claims of this work is that this presumption does not

[2] With the introduction of version 5, the J2EE framework was recently renamed "Java EE".

carry to enterprise applications. Complex systems are composed of many different components in many different ways, but an automatic composition engine tends to make arbitrary decisions, whose combined effect is more likely to break larger systems and disturb existing, or legacy code.

This problem is examined in greater detail in Section 1.1.2.

A natural quest then is for a harmonious integration of middleware architectures and AOP, combining the benefits of each to provide a coherent solution. This can be done by replacing middleware services with more flexible aspects. Indeed, there were several works on an AOP-based implementation middleware frameworks (e.g., [56, 129, 190, 192, 206, 216, 222]). However, equally important is the quest to tame aspects and make them more manageable for large-scale projects.

An overwhelming majority of previous work on the integration of aspects and middleware are based on the JAVA programming language in general, and on J2EE, which is the standard JAVA middleware framework. This work will follow in using JAVA as our base programming language and J2EE as a "reference" middleware framework, although the lessons learned here are applicable to other languages and other middleware frameworks. The new approach and main contribution of this paper is in drawing from the lessons of J2EE and its implementation to design *a new AOP technology*, geared towards the generalized implementation of J2EE application servers and applications within this framework, and in middleware frameworks in general.

At the heart of this new approach reside *shakeins*, a new programming construct that we present as a more manageable alternative to aspects. Unlike ASPECTJ-style aspects, shakeins can be applied safely to legacy code, because they do not break the object model, and their application to other program modules is in a controlled and configurable manner. And, compared to existing J2EE services, shakeins offer *greater flexibility* as well as *extensibility*, since developers can introduce their own shakein-based services.

1.1 Why Existing Solutions Are Broken

This sickness doth infect
The very life-blood of our enterprise.

— William Shakespeare, *King Henry IV Part I*,
Act IV, Scene I

Although both middleware frameworks and aspect-oriented programming try to address the problem of code complexity in enterprise applications, we argue that neither of the two, in its current form, is sufficient. We now examine in some detail the limitations of each approach, concluding in the need for integrating both into a coherent solution.

1.1.1 Limitations of the Services-Based Solution

Ideally, with modern middleware frameworks (and in particular with the J2EE framework), the developer only has to implement the domain-specific business logic. This business logic is none other than what the AOP community calls *functional concerns*. Various *services* provided by the application server handle what are known as *non-functional concerns* in AOP jargon. Yet even though the J2EE framework simplifies enterprise application programming and reduces the amount of scattered and tangled code, there are limits to such benefits. The reason is that although

J2EE application servers (the programs that implement the middleware specification) are configurable, they are neither *extensible* nor *programmable*. Pichler, Ostermann, and Mezini [191] refer to the combination of these two problems as *lack of tailorability*.

The application server is *not extensible* in the sense that the set of services it offers is fixed, in other words, no new services can be introduced by the developer. Kim and Clarke [154] explain why supporting logging in the framework would require scattered and tangled code. In general, J2EE lacks support for introducing new services for non-functional concerns which are not part of its specification. Among these concerns, we mention logging, memoization, precondition testing, checkpointing, and profiling.

The application server is *not programmable* in the sense that the implementation of each of its services cannot be easily modified by the application developer. For example, most implementations of the J2EE persistence service rely on a rigid model for mapping data objects, called *Enterprise Java Beans* (EJBs or "beans" for short), to a relational database. The service is then useless in the case that data attributes of a bean are drawn from several tables; nor can it be used to define read-only beans that are mapped to a database view, rather than a table. The persistence service is also of no use when the persistent data is not stored in a relational database (e.g., when flat XML files are used).[3]

Any variation on the functionality of the persistence service is therefore by *re-implementation* of object persistence, using what is called "bean managed" persistence. This requires the introduction of callback methods (called *lifecycle methods* in J2EE parlance) in each bean. One lifecycle method, for example, is invoked whenever the bean in memory should be updated with the version stored in persistent storage, and conversely, a differently lifecycle method is invoked whenever the persistent storage should be updated to reflect changes in the bean.

The implication is that the pure business logic of bean classes is contaminated with unrelated I/O code. For example, the official J2EE tutorial [31, Chap. 5] includes many code examples with a mixup, in the same bean, of SQL queries and a JAVA implementation of functional concerns. At the same time, we find that the code in charge of persistence is *scattered* across all entity bean classes, rather than being encapsulated in a single cohesive module.

Bean-managed persistence may also lead to code *tangling*. Suppose for example that persistence is optimized by introducing a "dirty" flag for the object's state. Then, each business logic method which modifies state is tangled with code to update this flag.

Persistence is only one example; similar scattering and tangling issues rise with modifications to any other J2EE service. In our financial software example, a security policy may restrict a client to transfer funds only out of his own accounts. The funds-transfer method, which is accessible for both clients and tellers, acts differently depending on user authentication. Such a policy cannot be defined by setting configuration options, and the method code must explicitly refer to the non-functional concern of security.

To summarize, whenever the canned solutions provided by the J2EE platform are insufficient for our particular purpose, we find ourselves facing again the problems of scattered, tangled and cross-cutting implementation of non-functional concerns. As Duclos, Estublier and Morat [88] state: "*clearly, the 'component' technology introduced successfully by EJB for managing non-functional aspects reaches its limits*".

1.1.2 Limitations of Existing AOP Solutions

Plain aspects should make it easier to *add* services to middleware frameworks, if such frameworks are re-engineered to use AOP. They should also make it simpler to *replace* existing services with

[3] Persistent EJBs were replaced by *entities* in the EJB 3.0 specification [81] which is part of JAVA EE 5. However, entities are also limited to persistence in relational databases, and any alternative storage mechanism presents many of the same challenges discussed here.

alternative implementations. But of no less importance is the ability to *integrate* with existing program elements. When applied to existing middleware projects, an aspect-oriented solution must respect legacy code, and take great care not to break it. The huge investment in existing code within enterprise applications implies that new code must never be allowed to mistakenly alter working parts of the program. Yet such unintentional changes are an inherent risk in ASPECTJ-style aspects.

Aspects in ASPECTJ (and related languages) work by *changing existing code*, often on a program-wide scale. Aspects are advertised as a means for breaking the implementation into its orthogonal concerns; accordingly, an aspect may replace or refine the implementation of a class, and even change its type by introducing new members. Other mechanisms known to the development community do just the same; the most familiar example, in the world of object-oriented programming, is inheritance. However, unlike inheritance, the application of an aspect to a class does not define a new class, but rather changes existing classes *in situ*. This change is *destructive*, since classes touched by aspects cease to exist in their original form; only the modified form exists, and in particular, there cannot be instances of the original classes in memory. This can have dire effects on possibly unrelated parts of the program that rely on the current, tested behavior of certain classes. The larger the project, the more likely are aspects to break legacy code; which is why we believe that *current aspect mechanisms do not scale up to enterprise applications*.

Also unlike inheritance, ASPECTJ-style aspects can modify not only the directly-targeted class, but also its *clients*. Thus, even a simple aspect that appears to target a single, well-defined class can in fact modify an unbound number of classes, implicitly. This is the case, for example, when the aspect applies advice to field-access operations, or to method invocation operations; any client that accesses such a field (or method) can now be modified. As the project grows in size, it becomes increasingly difficult for the developers to have a clear concept of how the various aspects interact with the different classes. With enterprise applications, the simple question "which aspects modify this class" cannot be answered except by dedicated tools that examine the code. And when more than one aspect applies to the same class, the next self-evident question, "in what order (or precedence) do aspects modify this class," is similarly hard to answer.

The difference between aspects, which modify classes in an unbound and destructive manner, and inheritance, is what we call *The Aspects-Inheritance Schism*. It is discussed in greater detail in Section 2.1.2. As we shall see, the integration of aspects into an object-oriented system raises many questions and potential conflicts regarding the inclusion of these two mechanisms in a single software project.

The shakeins construct is a proposal to resolve the aspect-inheritance schism by making aspects which are a restricted form of inheritance. Shakeins use aspect-like semantics to modify existing classes; however, the application of a shakein is never global. A shakein is always applied to a specific class, and generates a new class from it, much like inheritance. (In fact, the natural approach for implementing shakeins is using subclassing.) The original classes are left unaltered, so that the application of a shakein cannot inadvertently break unrelated code. Further, clients of the target class are never modified by a shakein.

Shakeins present a form of *parameterized class modification* (Section 2.3), and differ from aspects in that their application is *selective* and *non-destructive*. Because the application is not global, developers can *control the shakein application order* on a case-by-case basis when multiple shakeins are applied to the same class. Shakeins can also be *composed* to create new shakeins. Finally, their parameterized nature gives shakeins the flexibility required by middleware services, without limiting developers to a pre-defined set of such services.

1.1.3 Marrying J2EE with AOP

And they lived happily ever after.

— Proverbial fairy-tale ending

Despite the complexity of middleware systems, and despite the similarity between AOP and the services that frameworks such as J2EE provide, it is evident that integrating this modern programming technique into the frameworks that need it, faces reluctance. For example, let us examine the manner in which some J2EE application server vendors have recently integrated minimal support for aspects into their products:

- Release 4.0 of JBoss [144], an open-source application server which implements the J2EE standard, supports aspects with no language extensions [45]. Aspects are defined as JAVA classes which implement a designated interface, while pointcuts are defined in an XML syntax. These can be employed to apply new aspects to existing beans without introducing scattered code. Standard services however are not implemented with this aspect support.

- WebLogic Server [23], a J2EE application server developed and marketed by BEA Systems, also includes support for aspects [33], using either the AspectWerkz [34] framework or the ASPECTJ language. Here, too, aspects can be applied to existing beans without scattered or tangled code, but they are not used to implement the core J2EE services.

- The latest version of the EJB standard (EJB 3.0) chose not to adopt a complete AOP solution, but rather a rudimentary mechanism that can only be used to apply advice of a limited kind to a limited set of methods in an inflexible manner. The basic services are not implemented using such advise, but rather remain on a different plane, unreachable and non-modifiable by the developer.

We see that in none of these solutions do aspects serve as a replacement for services, and none implement the core services using aspects. We believe that this reluctance to integrate AOP into J2EE is mainly a result of the scaling problems noted above.

In a proper and complete integration of the two approaches, each of the services that J2EE provides should be expressed as an aspect. The collection of these services will be the *core aspect library*, which relying on J2EE success, would not only be provably useful, but also highly customizable. Developers will be able to add their own aspects (e.g., logging) or modify existing ones, possibly using inheritance in order to re-use proven aspect code.

The resulting aspects could then be viewed as stand-alone modules that can be reused across projects. Another implication is that not all aspects must come from a single vendor; in the current J2EE market, all J2EE-standard services are provided by the J2EE application server vendor. If developers can choose which aspects to apply, regardless of the application server used, then aspects implemented by different vendors (or by the developers themselves) can all be used in the same project.

In this research, we propose a new approach to the successful marriage of J2EE and AOP, in which the design of a new AOP technology draws from the lessons of J2EE and its programming techniques. Using shakeins, which are more tamed, controllable and scalable than ASPECTJ-style aspects, we have designed ASPECTJ2EE, which is an aspect-based J2EE framework.

1.2 Contributions and Outline

This section lists our contributions towards the goal of integrating aspects in middleware frameworks, and lists, for each contribution, the relevant publications. The final part of this work,

Chapter 7, includes our concluding remarks and suggests directions for further research.

1.2.1 Shakeins

Chapter 2 introduces *shakeins*, and makes the case that much more can be offered to the client community by this novel programming construct, combining features of aspects with selected properties of generics and mixins [38]. In a nutshell, a shakein receives a class parameter and optional configuration parameters, and generates a new class which has the same type as the original, but with different data and code implementation. In a sense, shakeins are restricted, yet less chaotic, aspects. Like aspects, they can be used to modularize non-functional concerns without tangled and scattered code. Unlike traditional aspects, shakeins preserve the object model, present better management of aspect scope, and exhibit a more understandable and maintainable semantic model.

Shakeins should make it possible not only to *add* services, but also address two issues that current J2EE users face: *configuration* of existing services and *greater flexibility* in their application. Shakeins should also enjoy a smoother entrance into the domain because they can be made to *reuse* the existing J2EE architecture, and because they can simplify some of the daunting tasks (e.g., lifecycle methods) incurred in the process of developing EJBs.

Shakeins are more restricted than aspects in that there are limits to the change they can apply to code. However, unlike aspects, shakeins take configuration parameters, which make it possible to tailor the change to the modified class.

We argue that the explicit, intentional, configurable and parameterized application of shakeins to classes makes them very suitable to middleware frameworks and enterprise applications.

Publications. Shakeins are the subject of the paper *Shakeins: Non-Intrusive Aspects for Middleware Frameworks* [63], published in Transactions on Aspect-Oriented Software Development.

1.2.2 ASPECTJ2EE

Shakeins draw from the lessons of J2EE and its implementation. To demonstrate their applicability to this domain, Chapter 3 introduces the ASPECTJ2EE language, which shows how shakeins can be used to bring the blessings of AOP to the J2EE framework. ASPECTJ2EE is geared towards the generalized implementation of J2EE application servers and of applications within this framework.

As the name suggests, ASPECTJ2EE borrows much of the syntax of ASPECTJ. The semantics of ASPECTJ2EE is adopted from shakeins, while adapting these to ASPECTJ. The main syntactical differences are due to the fact that "aspects" (shakeins) in ASPECTJ2EE can be parameterized. However, in ASPECTJ2EE, parameter passing and the application of shakeins to classes are not strictly part of the language. They are governed mostly by external XML configuration files, a-la J2EE deployment descriptors.

A distinguishing advantage of this new language is that it can be smoothly integrated into J2EE implementations without breaking their architecture. This is achieved by generalizing the existing process of binding services to user applications in the J2EE application server into a novel *deploy-time* mechanism of weaving aspects. Deploy-time weaving is superior to traditional weaving mechanisms in that it preserves the object model, has a better management of aspect scope, and presents a more understandable and maintainable semantic model. Also, deploy time weaving stays away from specialized JVMs and bytecode manipulation for aspect-weaving.

Standing on the shoulders of the J2EE experience, we can argue that shakeins in general, and ASPECTJ2EE in particular, are suited to systematic development of enterprise applications.

Publications. ASPECTJ2EE was introduced in the paper *AspectJ2EE = AOP + J2EE: Towards an Aspect Based, Programmable and Extensible Middleware Framework* [61], included in the proceedings of the 18th European Conference on Object Oriented Programming (ECOOP '04).

1.2.3 JTL

Any aspect-oriented language or framework must provide developers with a mechanism for specifying a set of join points, i.e., program locations that should be modified by relevant aspects. Such specifications, known as *pointcuts*, can be stated in a procedural manner, but are more commonly expressed using queries written in a dedicated, declarative language. Thus, we find that every aspect-oriented technology includes a mechanism for querying language elements.

The best-known example for such a query language is the pointcut specification language embedded in ASPECTJ, which is also used elsewhere (e.g., in the CAESAR [171] programming language). However, during our work on shakeins and ASPECTJ2EE, we have come to realize that ASPECTJ's pointcut specification language has major shortcomings, and in particular limited expressive power. Indeed, we were not the first to note these limitations, and numerous works suggesting new program query languages have appeared in recent years, e.g., [92, 126, 128, 143, 187].

Our own shot at the program query problem is presented in Chapter 4: *JTL*, the Java Tools Language (pronounced "Gee-tel"). JTL relies on a simply-typed relational database for program representation, rather than an abstract syntax tree. The underlying semantics of the language is restricted to queries formulated in First Order Predicate Logic augmented with transitive closure (FOPL*).

Special effort was taken to ensure terse, yet readable expression of logical conditions. The JTL pattern `public abstract class`, for example, matches all abstract classes which are publicly accessible, while `class { public clone(); }` matches all classes in which method `clone()` is public. To this end, JTL relies on a DATALOG-like [50] syntax and semantics, enriched with quantifiers and pattern matching which all but eliminate the need for recursive calls.

Like all code-query languages, JTL is not limited in use to specifying pointcuts in aspect-oriented languages. Other applications include fixing type constraints (concepts [106, 124, 210]) for generic programming, specification of encapsulation policies, search tools in development environments, a definition language for detecting code defects and "code smells" [100], the specification and detection of micro-patterns [110], and more.

We have further shown how a novel, yet simple, extension to JTL makes it possible to use JTL for the general task of *program transformation*. With this extension, JTL can be used for a variety of program transformation tasks, such as the generation of database schema descriptions for persistent classes, refactoring, and more. In particular, JTL can be used as an AOP language in its own right, cheap and quick yet surprisingly powerful.

Publications. JTL was originally presented in *JTL—the Java Tools Language* [67], included in the proceedings of the ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '06). The program transformation extensions are described in the paper *Guarded Program Transformations with JTL* [65], currently under preparation. Additional uses of JTL, for the domain of reverse engineering (unrelated to AOP and not further discussed in this work) appear in the paper *JTL and the Annoying Subtleties of Precise μ-Pattern Definitions* [68], in the First International Workshop on Design Patterns Detection for Reverse Engineering (DPD4RE '06/WCRE '06).

1.2.4 Factories

The ASPECTJ2EE experiment indicates that the shakein semantics integrates well with current J2EE server architectures. However, the same experiment also led us to the conclusion that program modules should be provided with a better mechanism for controlling object instantiation. Therefore, in Chapter 5 we present *factories* as a new language construct, providing each class with complete control over its own instance-creation process.

Factories are methods which return new class instances. Syntactically, a factory is a method which overloads the **new** operator with respect to a certain class. The language extension requires no changes to the JVM.

The need for factories stems directly from anomalies that can be found in constructors (in JAVA, C++ [209], etc.). To overcome these anomalies, the J2EE framework uses *home objects* [202, Chap. 5] to control object instantiation; JAVA EE 5 and Hibernate [22] use runtime bytecode patching mechanisms; the Spring framework [146] uses a central factory object; and so forth. All these devices share in common an attempt to wrestle control over object instantiation from the programmer, so that standard classes can be replaced by augmented classes (with services or aspects applied). Our suggested factories (not to be confused with the FACTORY METHOD or ABSTRACT FACTORY design patterns [105]) allow any client of a class to control its creation in an orderly manner. Different clients can obtain instances of the same class decorated in a different manner, depending on each client's need, using *client-side factories*. And with *supplier-side factories*, the class itself can choose to provide only instances augmented in some specific manner.

The presentation of factories in Chapter 5 also includes *a taxonomy of class features*.

Publications. Factories were discussed as an appendix in the paper about shakeins [63]. The paper *Better Construction with Factories* [64], focused solely on factories and containing more details, was accepted for publication in the Journal of Object Technology.

1.2.5 Object Evolution

A current topic of interest in aspect-oriented programming is *dynamic aspects* [148, 190, 192], i.e., the ability to apply (or remove) different aspects to (from) a given object during the object's lifetime. Because applying a shakein to a class generates a new class, we find that dynamic aspects, with regard to shakeins, imply the ability to change an object's class at runtime—a facility known as *object reclassification* [86, 218].

As it turns out, reclassification using shakeins is part of a more general concept, which we call *object evolution*. Presented in Chapter 6, object evolution is a restriction of general reclassification by which dynamic changes to an object's class are *monotonic*: an object may gain, but never lose, externally-visible properties.

We argue that there are many applications of monotonic evolution in practical systems. The monotonicity property makes it easier to maintain static type safety with object evolution than in general object reclassification. Note that the monotonicity property may make evolution irreversible; this restriction is ameliorated by separating the notion of class from that of type, and in the case of shakeins-based evolution, we find that the mechanism can support repeated state changes, and even undo changes, under certain limitations. This allows object evolution to enable the use of shakeins as dynamic aspects.

We also show that object evolution requires less changes to the host object-oriented programming language and collects a reduced performance toll, compared to unrestricted reclassification facilities, mostly because all descends in the inheritance hierarchy are necessarily monotonic.

We describe three concrete variants of evolution, relying on inheritance, mixins, and shakeins,

and explain how any combination of these can be integrated into a concrete programming language.

Publications. Object evolution is discussed in the paper *Three Approaches to Object Evolution* [62], currently under consideration for publication in the journal Science of Computer Programming.

Chapter 2

Shakeins

Shaken, not stirred.

— James Bond, in *Goldfinger*

As detailed in the Introduction (Section 1.1), we find that the addition of aspect-oriented programming techniques to the realm of middleware frameworks faces reluctance. Even those few application server vendors that do offer AOP extensions to their systems, such as JBoss (JBoss Application Server) and BEA (WebLogic Server), do not implement the basic services as aspects. These services remain on a different plane, unreachable and non-modifiable by the developer.

Plain aspects should make it easier to *add* services to J2EE, if re-engineered to use AOP. They should also make it simpler to *replace* existing services with alternative implementations. This chapter that much more can be offered to the client community by *shakeins*, a novel programming construct, combining features of aspects with selected properties of generics and mixins [38]. Shakeins should make it possible not only to *add* services, but also address two issues that current J2EE users face: *configuration* of existing services and *greater flexibility* in their application. Shakeins should also enjoy a smoother entrance into the domain because they *reuse* the existing J2EE architecture, and because they simplify some of the daunting tasks (e.g., lifecycle methods) incurred in the process of developing EJBs.

In a nutshell, a shakein receives a class parameter and optional configuration parameters, and generates a new class which has the same type as the original, but with different data and code implementation. In a sense, shakeins are restricted, yet less chaotic, aspects. Like aspects, they can be used to modularize non-functional concerns without tangled and scattered code. Unlike traditional aspects, shakeins preserve the object model, present better management of aspect scope, and exhibit a more understandable and maintainable semantic model.

Shakeins are more restricted than aspects in that there are limits to the change they can apply to code. However, unlike aspects, shakeins take configuration parameters, which make it possible to tailor the change to the modified class.

We argue that the explicit, intentional, configurable and parameterized application of shakeins to classes makes them very suitable to middleware frameworks and enterprise applications, based of the following reasons:

- The architecture of *existing* middleware frameworks is such that in generating enterprise application from them, the system architect may selectively use any of the services that the framework provides. We have that *a chief functionality of middleware frameworks is in the precise management of non-functional concerns*, and that this objective is well served by the shakeins construct.

- The services that a middleware framework offers are applicable only to certain and very distinguishable components of the application, e.g., EJBs in J2EE. The need to isolate the application of aspects is aggravated by the fact that enterprise applications tend to use other software libraries, which should not be subjected to the framework modifications. For example, a business mathematics library that an application uses for amortizing loan payments, should not be affected by the underlying framework security management. Shakeins are suited to this domain because of its fundamental property by which *cross cutting concerns should not be allowed to cut through an entire system.*

- Further, for modularity's sake, it is important that these parts of the enterprise application, which are external to the middleware, are not allowed to interact with the services offered by the middleware. The fact shakeins are forbidden from creating new types helps in *isolation of the cross cutting concerns.*

- To an extent, shakeins generalize existing J2EE technology, presenting it in a more systematic, programming language-theoretical fashion. While doing so, shakeins make it possible to enhance J2EE services in directions that plain aspects fail to penetrate: The configurability of shakeins makes it possible to generalize existing J2EE services. Explicit order of application of shakeins to components adds another dimension of expressive power to the framework. And by using *factories* (Chapter 5), such enhancements can be applied with a minimal disturbance of existing code.

Chapter outline. Section 2.1 points at an inherent conflict between aspects and inheritance in languages that combine the object- and aspect-oriented paradigms. Section 2.2 presents the shakeins mechanism, and shows how it brings the benefit of AOP to OOP languages without breaking the object model. Next, Section 2.3 lists the benefits of the parameterized class modification approach offered by shakeins, and Section 2.4 compares shakeins with other AOP approaches used in middleware frameworks. Section 2.5 concludes and provides a road map for the following chapters.

2.1 Classes, Types, and Aspects

As explained above, a shakein S takes an existing class c as input, along with optional configuration parameters, and generates from it a new class $S\langle c\rangle$, such that the *type* of $S\langle c\rangle$ is the *same* as that of c. The *realization* of this type in $S\langle c\rangle$ may, and usually will, be different than in c. Further, the process of generating $S\langle c\rangle$ from c is *automatic*, *universal* and *uniform*. The generation process is universal and uniform [48] because the same shakein can be applied to a potentially infinite number of classes—as opposed to inheritance, which generates a new class from a given class in an ad-hoc, class-specific manner (see Section 2.3 for details).

Section 2.1.1, below, explains what we mean by making the distinction between the type defined by a class and its realization by the class definition. With this distinction, we explain in Section 2.1.2 why the mechanisms of aspects[1] and inheritance serve similar purposes but from different, foreign, and hard-to-reconcile perspectives. Section 2.2 presents the shakeins mechanism to resolve this schism.

2.1.1 The Five Facets of a Class

As early as 1971, Parnas [188] made the classical distinction between the *interface* and the *materialization* perspectives of a software component. It is only natural to apply this distinction to

[1] We refer here to ASPECTJ-style aspects. Most other aspect-oriented languages are variants of this same style.

class definitions in JAVA and other mainstream object-oriented languages.

We say that the interface and materialization are *purposes* that the class serves as a whole, and characterize its elements by this purpose. (The purpose is but one of several dimensions used in characterizing classes. Additional dimensions will be discussed in Chapter 5.)

Unlike the software components of the seventies, classes are instantiable. Accordingly, we break the interface of a class into two facets: the *forge* and the *type*. Similarly, we distinguish between three facets in the materialization: the *implementation* of the type, the *mill* behind the forge, and the *mold* into which instances are cast.

More specifically, the *forge* of the class is the collection of operations that can be used to create objects; the *type* is the set of messages that these instances may receive, along with their visibility specification; and the *implementation* is the body of the methods executed in response to these messages. There is a subtle distinction between the mill and the mold, which together realize the class's forge: The *mold* is the memory layout which instances of this class follow; it consists solely of field definitions. The *mill* is the set of constructor bodies.

To understand these terms better, consider class `Point`, shown in Figure 2.1.[2] The type of

```
class Point implements Shape {
   int x, y;

   public Point(){
      this(0,0);
   }

   public Point(Point other) {
      this.x = other.x;
      this.y = other.y;
   }

   public Point(int x, int y) {
      this.x = x;
      this.y = y;
   }

   public int getX() {
      return x;
   }
```

```
   // continued

   public int getY() {
      return y;
   }

   public void setX(int x) {
      this.x = x;
   }

   public void setY(int y) {
      this.y = y;
   }

   public void moveBy(int dx,
                      int dy) {
      x += dx;
      y += dy;
   }
}
```

Figure 2.1: The `Point` class definition

`Point`, containing methods such as `setX`, `moveBy`, and others, as well as the fields `x` and `y`, is shown in Figure 2.2(a). Superclasses and interfaces also add to the type; in this case, the type of `Point` includes methods and fields inherited from on superclass, namely `Object`, and one interface, `Shape`. Each superclass and superinterface also adds an upcast operator.

We see that the type includes the signature of all non-**`private`** fields and methods of the class. Thus what we call type here is in fact the class's structural type, to which JAVA applies a name, making it a nominal type.

The type does not include details such as a specification of the order by which methods may be invoked, pre- and post-conditions, or other classes with which the class may interact while

[2] A similarly structured class served as the running example in Kiczales and Mezini's [152] work on modularity and aspects.

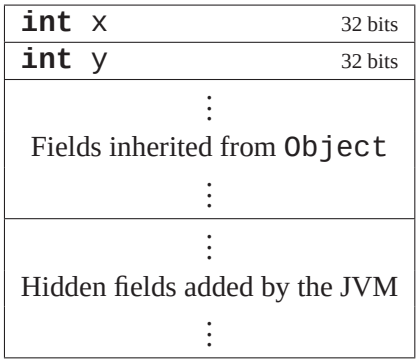
implementing each method. All these may be thought of as the class's *protocol*.

```
int x, y;
public int getX();
public int getY();
public void setX(int);
public void setY(int);

// From Shape:
public void moveBy(int, int);

// From java.lang.Object:
public boolean equals(Object);
public int hashCode();
// ... etc.

// Upcast operations:
public (Shape)Point;
public (Object)Point;
```

(a) The type of `Point`.

```
public Point(int x, int y);
public Point();
public Point(Point other);
```

(b) The class's forge.



(c) The class's mold.

Figure 2.2: The type, forge, and mold ensuing from the definition of class `Point`

The forge of `Point`, depicted in Figure 2.2(b), includes the signatures of the three constructors provided by the class: the zero-arguments constructor (`Point()`), the copy constructor (`Point(Point)`), and the variant that specifies the initial coordinates (`Point(int, int)`).

The *mold* for creating new objects is defined by the collection of all fields in this class and all of its supertypes. Specific languages or language implementations can include hidden fields in the mold, such as run-time type information, the Virtual Method Table [93] used in C++, etc. Figure 2.2(c) presents the mold defined by class `Point`. It includes fields defined in `Point` as well as any fields inherited from superclasses, along with any hidden field added by the JVM.

Finally, the *implementation* is the body of the methods defined by the class or any of its superclasses, while the *mill* is the body of the constructors defined in this class.

2.1.2 The Aspects-Inheritance Schism

A key feature of OOP is the ability to define new classes based on existing ones. The five facets described above can be used for explaining inheritance in JAVA and similar languages: Inheritance *extends* the type, mold and implementation of a class. By means of overriding, inheritance also makes it possible to *replace* or *refine* parts or all of the implementation. Inheritance can also refine the type in languages that allow variance. The mold, however, cannot be refined.

Interestingly, inheritance in mainstream programming languages *recreates* the forge from scratch, since constructor signatures are not inherited. Therefore, inheritance is allowed to make arbitrary and incompatible changes to the forge facet of a class. Still, the mill can only be refined, since constructors must invoke inherited versions.

Examining the notion of aspects from the perspective of modifications to the five facets, we see that they offer a similar repertoire of modifications. Aspects are advertised as means for breaking the implementation into its orthogonal concerns; accordingly, an aspect may replace or refine the implementation. In some aspect-oriented languages, aspects can also *extend* the type and the mold,

by introducing new members into a class. (This is known as *member introduction* in ASPECTJ. Conceptually, fields in an aspect that has per-object instantiation may also be viewed as fields added to the base object's mold.) Similarly, again using member introduction, aspects can *extend* the forge, but they cannot replace it.

Still, unlike inheritance, the application of an aspect does not define a new type, nor does it introduce a new mold or a new implementation; it changes the mold and the implementation of existing classes *in situ*. And, while member introduction makes it possible to modify the type of classes, it does not introduce a *new* type—because classes touched by aspects cease to exist in their original form; only the modified form exists. In particular, there cannot be instances of the original class in memory.

The similarity of aspects and inheritance raises intriguing questions pertaining to the interaction between the two mechanisms: Does the aspectualized class inherit from its base version? May aspects delete fields from the mold, or modify it by changing the type of fields? Likewise, can methods be removed from the type? Can their signature be changed? How does the application of an aspect to a class affect its subclasses?

The interaction of join points with inheritance raises still more questions. In ASPECTJ, two different join point types can be used to apply an advice to method execution: The **`execution`** join point places an advice at the method itself, whereas **`call`** places the advice in at the method's client, i.e., at the point of invocation rather than at its target. Now, suppose that class `B` inherits from `A`, with or without overriding method `m()`. Then, does the pointcut **`call`**`(A.m())` apply to an invocation of method `m()` on an object whose dynamic type is `B`? Conversely, does the pointcut **`call`**`(B.m())` ever apply if class `B` does not define a method `m()`, but inherits it from `A`? (These confusing semantics are analyzed in detail by Barzilay, Feldman, Tyszberowicz, and Yehudai [21].)

The reason that all these questions pop up is that aspects were not designed with inheritance in mind. The original description of aspects [151] did not dedicate this new construct to OOP languages. Similarly, the founding fathers of OOP did not foresee the advent of AOP, and in many occasions inheritance was used for some of the purposes AOP tries to serve. We witness what may be called *The Aspects-Inheritance Schism*.

Gradecki and Lesiecki [121, p. 220] speculate that "*mainstream aspect-oriented languages . . . will possess simpler hierarchies with more type-based behavior defined in terms of shallow, crosscutting interfaces.*" In other words, these two authors expect that this schism is resolved by programmers abandoning much of the benefits of inheritance. Our suggestion is that the resolution will be a re-definition of aspects cognizant of the class facets and of inheritance.

2.2 Shakeins as Class Re-Implementors

The shakeins construct is a proposal to resolve the aspect-inheritance schism by making *aspects which are a restricted form of inheritance*. Like inheritance, shakeins generate a new class from an existing one. Yet unlike inheritance, they cannot extend the class type. In fact, they cannot change the base class type at all.

Thus, we can say that *shakeins make a **re-implementation** of a class.*

Shakeins allow only specific changes to the class facets. Programming constructs which restrict one or more facets are not strange to the community: an *abstract class* is a type, a partial (possibly empty) implementation, and an incomplete mold. *Interfaces* are pure types, with no implementation, forge, mill or mold. Also, *traits* [199] have a fragment of a type and its implementation, but no forge, mill or mold.

Another familiar notion on which shakein rely is that of multiple implementations of the same (abstract) type, as manifested e.g., in the distinction between signatures and structures in

ML [175]. The major difference is, however, that a shakein assumes an existing implementation of a class, which it then modifies.

For concreteness, Figure 2.3 shows shakein `DisplayUpdating`, that can be used to modify class `Point` so that the display is refreshed after each coordinate change. The shakein works on

```
shakein DisplayUpdating {
  pointcut change() :
           execution(setX(int)) || execution(setY(int)) ||
           execution(moveBy(int,int));

  after() returning: change() {
    Display.update();
  }
}
```

Figure 2.3: A shakein for updating the display after each change to a `Point`

any class that defines methods `setX(int)`, `setX(int)`, and/or `moveBy(int,int)`. The desired effect is obtained by applying this shakein to `Point`. In the spirit of previous works that approached the display-updating aspect example [55, 152, 153], we assume that the there is a single, system-wide display which is globally accessible.

The syntax used to define shakeins (here and in the following examples) is an ASPECTJ-like syntax, using the keyword **`shakein`** rather than **`aspect`**. However, other syntactical choices can be made if shakeins are integrated into other programming languages, and in particular, Chapter 4 will discuss a suggested change to the pointcut specification language used here.

Figure 2.4 shows the `Confined` shakein. This shakein confines the range of valid values

```
shakein Confined {
  pointcut update(int v) :
           (set(int x) || set(int y)) && args(int v);

  before(int v): update(v) {
    if (v < 0)
      throw new IllegalArgumentException();
  }
}
```

Figure 2.4: A shakein for limiting the valid range of `Point`'s coordinates

for `x` and `y` to positive integers only. It is applicable to class `Point`, or any class with **`int`** fields `x` and `y`.[3]

In the process of re-implementing a class, a shakein may introduce methods and fields to the class. Such members must be **`private`**, accessible to the shakein but not to the external world (including inheriting classes, and classes in the same package). The reason is that such accessibility would have implied a change to the class type, which is not allowed to shakeins.

We will write `Confined<Point>` to denote the application of shakein `Confined` to `Point`, and `DisplayUpdating<Confined<Point>>` for the application of shakein `Disp-`

[3] While the update presented by this shakein can cause methods that update `x` and `y` to throw an exception, it is an unchecked ("runtime") exception, and therefore it does not alter the methods' signature—a forbidden change, since it implies an update to the class's type.

`layUpdating` to the result, etc.

The re-implementation property implies that although a class $S\langle c\rangle$ can (and usually will) be instantiated, it is not possible to define *variables* of this class. Instances of $S\langle c\rangle$ can be freely stored in variables of type c, as in the following JAVA-like pseudo-code:

```
c var = new S⟨c⟩();
```

In the `Point` example, we may then write:

```
Point p = new DisplayUpdating<Point>();
```

The type-preservation property of shakeins sets a clear semantics for the interaction of these with inheritance. As it turns out, the application of shakeins does not modify or break the inheritance structure. More formally, let C_1 and C_2 be two classes, and suppose that C_2 inherits from C_1. Then, we can write $C_2 \prec C_1$ to denote the fact that the type of C_2 is a subtype of C_1. Let S be a shakein. Then, we can also write $C_1 \simeq S\langle C_1\rangle$ to denote the fact that the type of $S\langle C_1\rangle$ is the same as C_1. Similarly, $C_2 \simeq S\langle C_2\rangle$. By substitution, we can obtain $S\langle C_2\rangle \prec S\langle C_1\rangle$, $C_2 \prec S\langle C_1\rangle$, and $S\langle C_2\rangle \prec C_1$. In fact, we have

Proposition 1. *For all classes C_1 and C_2 such that $C_2 \prec C_1$, and arbitrary shakeins S and S', $S'\langle C_2\rangle \prec S\langle C_1\rangle$.*

In our running example, shakein `DisplayUpdating` can be applied to any subclass of `Point`. If class `ColorPoint` extends `Point`, then the type of `DisplayUpdating<ColorPoint>` is a subtype of `Point`.

Figure 2.5 makes a graphical illustration of Proposition 1. It depicts a simple base class hierarchy consisting of classes C_1, C_2, C_3, and C_4, where $C_4 \prec C_2 \prec C_1$, and $C_3 \prec C_1$. There are

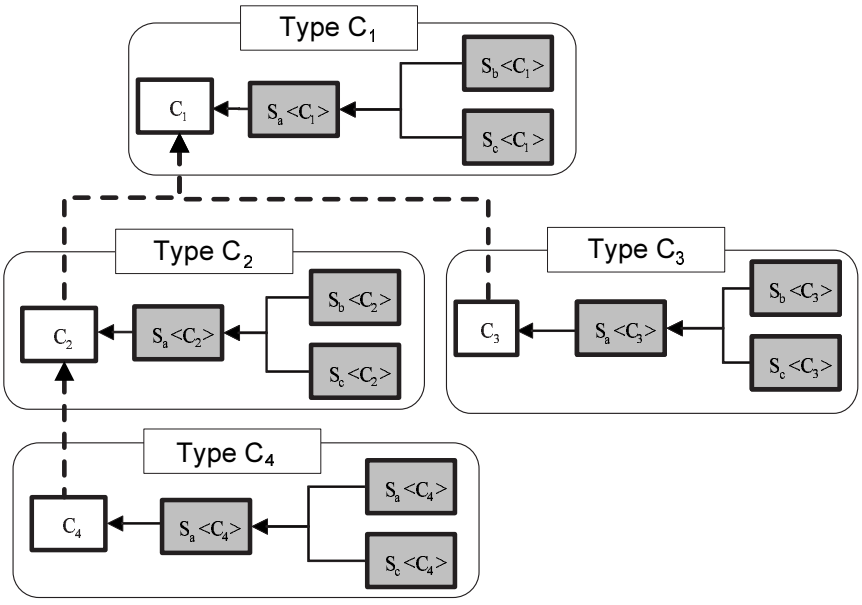


Figure 2.5: A class hierarchy subjected to shakeins. Each round-cornered box represents a type; each internal rectangle represents a class that implements this type

also three shakeins, S_a, S_b and S_c, where shakeins S_b and S_c are implemented using S_a, and each of the shakeins is applied to each of the classes.

We see in the figure that for all $i = 1, \ldots, 4$, the type of class C_i is the same as its three re-implementations $S_a \langle C_i \rangle$, $S_b \langle C_i \rangle$ and $S_c \langle C_i \rangle$. This common type is denoted by a round-cornered box labeled "Type C_i". As shown in the figure, the subtyping relationship is not changed by re-implementations; e.g., the type of class $S_a \langle C_4 \rangle$ is a subtype of $S_b \langle C_2 \rangle$'s type.

The figure should also make it clear how the type system of shakeins can be embedded in a standard JVM. Each shakein application results in the generation of a JAVA class, which compiles into a distinct `.class` file. Both vertical dashed arrows, representing type inheritance, and horizontal arrows, representing shakein application, are translated to class inheritance, i.e., an **`extends`** relationship between classes.

To see why Prop. 1 holds in this embedding, recall that the program does not have any variables (including parameters and fields) of the shakein classes: All instances of class $S \langle c \rangle$ are stored in variables of type c. In the figure, instances of type $S_a \langle C_4 \rangle$ are stored in variables of type C_4, which is upcastable to type C_2.

2.3 Parameterized Class Modification

Double, double, toil and trouble;
Fire burn, and Cauldron bubble.

— William Shakespeare, *The Tragedy of Macbeth*,
Act IV, Scene I

Aspects are distinguished from inheritance in that they can be automatically applied to multiple classes. In contrast, a subclass is defined with respect to a specific superclass; the nature of the extension (both of the interface and the materialization) is specific to every such instance. To apply the same kind of change to multiple base classes, the details of the change must be explicitly spelled out each time. Thus, although inheritance is a kind of what is known in the literature as *universal polymorphism* [48], it is not a *uniform* mechanism; each inheriting class modifies the base class in an *ad-hoc* manner.

It is therefore instructive to compare aspects to the parameterized version of inheritance, i.e., *mixins* [38]. A mixin, just like an aspect, makes it possible to apply the same kind of change (expressed in terms of inheritance) to multiple base classes.

Mixins were invented with the observation that there is a recurring need to extend several different classes in the exact same manner. They allow programmers to carry out such a change without rewriting it in each inheriting class. In languages such as MODULA-π [37] or JAM [8], the repeating change to the base class can be captured in a mixin M, which, given a class c, generates a class $M \langle c \rangle$ such that $M \langle c \rangle$ inherits from c. In languages with first-class genericity, such as C++ and NEXTGEN [5], mixins can be emulated by writing, e.g.,

```
template<typename c> class M: public c { ... }
```

in C++, or

```
class M<c> extends c { ... }
```

in NEXTGEN.

Generic structures are also a kind of a universal polymorphism, but their application is *uniform*. The above emulation of mixins by generics makes it clear that mixins are both *universal* and *uniform* kind of polymorphism.

Here, we enrich and simplify the aspects approach with ideas drawn from the work on mixins. We argue that the seemingly pedestrian notation of mixins, $M\,\langle c\rangle$, hides much expressive power. The reason is that parameter passing, a familiar mechanism of programming languages, exhibits several useful features which are missing in the current implicit and declarative mechanism of aspect application. These features are:

1. ***Selective Application.*** After a shakein was defined, it can be applied to selected classes without affecting others. In contrast, aspects have essentially two kinds of components: *global advices*, which, as the name suggests, apply to all classes in the universe of discourse; and *tailored advices*, which apply to a pre-defined list of specific classes.

   Thus, if we wish to apply a given aspect only to certain classes, we must use a tailored advice that will have to be modified each time a new class joins the list.[4]

   Conversely, an aspect with a global advice can affect an unbounded number of classes, potentially including classes that did not exist when the aspect was defined. This is often a desired benefit. However, it is not possible in general to change the applicability of an aspect without modifying it: a programmer introducing a new class into a project cannot choose to "opt out" and leave his class unaffected by existing aspects, even if he deems some of these aspects inappropriate for that specific class.

   The only way to prevent this effect is to modify the aspect itself, so that its advices will exclude the new class. If the aspect originated at an off-the-shelf library, its sources unavailable, then even this remedy will not be possible.

   For example, Figure 2.6(a) shows an attempt to implement the `Confined` shakein from Figure 2.4 as an ASPECTJ aspect using a global advice. The problem with this attempt is

```
aspect Confined {
  pointcut update(int v) :
    (set(int *.x) ||
     set(int *.y))
     && args(int v);

  before(int v): update(v) {
    ... // Perform confinement test
  }
}
```

(a) Using a global advice

```
aspect Confined {
  pointcut update(int v) :
    (set(int Point.x) ||
     set(int Point.y))
     && args(int v);

  before(int v): update(v) {
    ... // Perform confinement test
  }
}
```

(b) Using a tailored advice

Figure 2.6: Two attempts to define the `Confined` shakein as an ASPECTJ aspect

   that it could inadvertently affect unrelated classes with **`int`** fields named `x` and/or `y`, such as the standard-library class `java.awt.Event`. Figure 2.6(b) attempts to do the same using advice tailored specifically for `Point`. But now, if we *do* want the aspect to apply to any other class, we must explicitly modify its pointcut definition, which can quickly become unwieldy.

2. ***Non-destructive Application.*** The application of a shakein S to a class c does not destroy c, and both classes $S\,\langle c\rangle$ and c can be used. Instances of both may reside in memory simultaneously and even interact with each other.

[4] Abstract pointcuts can also be used to this end. This construct is discussed below in Section 2.3.3.

In contrast, in most AOP languages, the original class cannot be used once its aspectualized version was generated. For example, if we apply a caching aspect to some class, we cannot simultaneously use a non-caching version for accessing a local service.

3. ***Explicit and Flexible Ordering.*** Shakeins S_1 and S_2 can be applied in any order, to generate either $S_1 \langle S_2 \langle c \rangle \rangle$, $S_2 \langle S_1 \langle c \rangle \rangle$, or both. Further, it is possible to apply the same mixins in a different orders to different class.

   In comparison, the order of application of aspects, i.e., the *aspect precedence* as it called in the AOP jargon, is a global system property. It is practically impossible, for example, to cause class `A` to log its operations before security checks are applied, while class `B` logs its operations only after such checks are performed. With shakeins, this is achievable simply by generating `Log<Security<A>>` alongside `Security<Log<B>>`.

4. ***Composition.*** Shakeins, like mixins, can be conveniently thought of as functions, and as such it makes sense to compose them. In some languages supporting mixins one can define a new mixin by writing, e.g., $M := M_1 \circ M_2$, with the obvious and natural semantics. Languages with shakein support can offer a similar syntax for shakein composition.

   In contrast, the declarative nature of aspects complicates the semantics of their composition, to the extent that aspect inheritance is all but prohibited in ASPECTJ and most related languages.

5. ***Configuration Parameters.*** It is straightforward to generalize mixins (and shakeins) so that they take additional parameters which control and configure the way they extend (or re-implement) the base class. In the templates notation, one can write for example:

   ```
   template<typename c, char *log_file_name>
      class Log: public c {
         // Log into file log_file_name
      }
   ```

   Again, the implicit application and the declarative nature of aspects makes the notion of configuration awkward. For example, making two subsystems log their operations in different files is not easy with a single, non-parameterized aspect.

6. ***Repeated Application.*** One can write $S \langle S \langle c \rangle \rangle$, but it is not possible to apply the same aspect twice to a class. While it makes little sense to apply most shakeins more than once, in some cases (especially with parameterizes shakeins) the notion is very natural; for example, a construct like

   ```
   Log["pre.log"]<Security<Log["post.log"]<c>>>
   ```

   will generate two log files, one before and one after any security checks imposed by the `Security` shakein.

7. ***Parameter Checking.*** It is useful to declare constraints to the parameters of shakeins, and apply (meta-) type checking in passing actuals to these. Languages such as JAM offer this feature in mixins; for example, a mixin could require that it is applied only to classes that implement the `java.io.Serializable` interface. We see no equivalent in the domain of aspects.

Evidently, these differences are not a coincidence. AOP languages were designed with the belief that the composition of a system from its aspects is an implicit process, carried out by some clever engine which should address the nitty gritty details. The developers throw all the

components, including both classes and aspects, into the big cauldron that is the program; and these are stirred by the AOP engine to generate the resulting product (see Figure 2.7). One of the chief claims of this work is that this presumption does not carry to middleware applications, especially when we wish to preserve the investment in existing architectures. Complex systems are composed of many different components in many different ways. An automatic composition engine tends to make arbitrary decisions, whose combined effect is more likely to break larger systems, especially when legacy code is involved. Our support for this claim is by the detailed description of the aspect oriented re-structuring of J2EE, in the form of the ASPECTJ2EE language (Chapter 3).

Figure 2.7: Generating software systems by mixing classes and ASPECTJ-style aspects

Shakeins are similar to mixins in that they take parameters. As such they are a *universal* and *uniform* polymorphic programming construct. Shakeins are similar to generic structures in that they may take multiple parameters. However, whereas both mixins and generics suffer from their *oblivious* and *inflexible* mode of operation, i.e., they are unable to inspect the details of the definition of the base-, or argument- class, and generate specific code accordingly. As a result, mixins fail in tasks such as re-generating constructors with the same signature as the base, or applying the same change to multiple methods in the base class. Similar restrictions apply to generics. In contrast, shakeins use the same pointcut mechanism as aspects, and are therefore highly adaptable.[5]

Another issue that plagues mixins (and generics that inherit from their argument) is that of *accidental overloading* [8]. Shakeins overcome this problem using an automatic renaming mechanism, which is possible since no shakein-introduced member is publicly accessible.

[5]The component-system Jiazzi [166] uses a mixin-like mechanism to implement *open classes*, which in turn can be used for implementing cross-cutting concerns [167]. Jiazzi components are also parameterized, taking packages (sets of classes) as arguments. However, Jiazzi's open class approach differs from shakeins in that the base class is modified, both in its implementation and in its type; and such changes are propagated to all existing subclasses.

The chief parameter of a shakein is the class that this shakein re-implements. The definition of a shakein does not need to name or even mention this parameter, since it is implicit to all shakeins, in the same way that JAVA methods have a **`this`** parameter. We will nevertheless write this parameter occasionally, e.g., when we wish to perform parameter checking upon it. Additional parameters, if they exist, are used to configure or direct the re-implementationNote that aspects can also be thought of as taking implicit parameters; however, shakeins are distinguished from aspects in that their invocation is *explicit*, and that aspects take no configuration parameters.

More formally, a shakein S takes an existing class c as input along with other configuration parameters, $\mathcal{P}_1, \ldots, \mathcal{P}_n$, $n \geqslant 0$, and generates from it a new class $S[\mathcal{P}_1, \ldots, \mathcal{P}_n]\,\langle c \rangle$, such that the *type* of $S[\mathcal{P}_1, \ldots, \mathcal{P}_n]\,\langle c \rangle$ is the *same* as that of c. We use square brackets for passing the configuration parameters while angular brackets are used for passing class c, the mandatory parameter representing the class to be re-implemented. Thus, $S[\mathcal{P}_1, \ldots, \mathcal{P}_n]$ is the configured shakein, which can then be applied to the target.

Figure 2.8 shows how configuration parameters can enhance the functionality of shakein `Confined` (first shown in Figure 2.4). As shown in the figure, the updated version of `Confined`

```
shakein Confined[int minX, int maxX, int minY, int maxY] {
  pointcut updateX(int v) : set(int x) && args(int v);
  pointcut updateY(int v) : set(int y) && args(int v);

  before(int v): updateX(v) {
    if ((v < minX) || (v > maxX))
      throw new IllegalArgumentException();
  }

  before(int v): updateY(v) {
    if ((v < minY) || (v > maxY))
      throw new IllegalArgumentException();
  }
}
```

Figure 2.8: A parameterized version of `Confined` (from Figure 2.4)

is configured by four **`int`** parameters, specifying the minimal and maximal values for the `x` and `y` coordinates of its target class. To obtain an instance of `Point` restricted to the $[0, 1023] \times [0, 767]$ rectangle, one can write

```
Point p = Confined[0,1023,0,767]<Point>(511,383); //Initially at center.
```

Like generic structures, shakeins can also accept *types* as configuration parameters. These can be used, for example, to define method arguments, or new (private) fields, using parametric types. In fact, the first parameter to a shakein is *always* a type: this is the existing class which the shakein should re-implement.

2.3.1 Pointcut Parameters

A special kind of configuration parameter is the pointcut expression. Figure 2.9 shows a revised version of the `DisplayUpdating` shakein, which uses a pointcut parameter. The parameter `change` denotes the join points whose execution necessitates a display update. An actual value

```
shakein DisplayUpdating [pointcut change()] {
   after() returning: change() {
      Display.update();
   }
}
```

Figure 2.9: A parameterized version of `DisplayUpdating` (from Figure 2.3), accepting a pointcut specification as a parameter

of this parameter specifies a concrete such set. For example, the following:

```
DisplayUpdating[
  execution(setX(int)) || execution(setY(int))
    || execution(moveBy(int,int)
]<Point>
```

is an application of `DisplayUpdating` to class `Point`. The parameter value is a pointcut that matches all methods in `Point` which should cause a display update.

Consider now Figure 2.10, showing the class `Line`, which is implemented using two `Point`s.

```
class Line {
   private Point a, b;

   public Line(Point from,
               Point to) {
      a = new Point(from);
      b = new Point(to);
   }
```

```
   // continued

   public moveBy(int x, int y) {
      a.moveBy(x,y);
      b.moveBy(x,y);
   }

}
```

Figure 2.10: Class `Line`

An application of `DisplayUpdating` to `Line` is by writing

```
DisplayUpdating[execution(moveBy(int,int))]<Line>.
```

This re-implementation of `Line` does not suffer from the redundant display updates problem [152], which would have occurred in traditional AOP, i.e., display updates occurring both in the implementation of `Line` and its encapsulated `Point`s. Thanks to the non-destructive semantics of shakeins, these two `Point`s can be of the non-updating variant. This does not prohibit other `Point`s in the system (which are not part of `Line`s) to be display-updating.

In contrast, an aspect based solution to this problem should check that no `change` advice is in effect before executing this advice. This checking must be carried out *at runtime*, by examining the runtime stack with what is known in ASPECTJ as a **`cflowbelow`** condition.[6]

2.3.2 Shakein Composition and Repeated Application

Examining Figure 2.8, we see that the `Confined` shakein contains a certain amount of code duplication: It contains two advice, which are identical except for the pointcut to which they are attached and the parameters which they use for setting the valid range.

[6] Still, **`cflowbelow`**-based pointcuts can be used in shakeins where needed—for example, to prevent a call to `Point.moveBy()` from causing multiple display updates as it changes both point coordinate; see [152] for a discussion of this use of **`cflowbelow`**.

By using shakein composition, we can do away with this duplication. Figure 2.11 is a re-definition of the `Confined` shakein using pointcut parameters and *shakein composition*. By

```
// Auxiliary shakein, used to confine updates to one variable:
shakein ConfinedUpdate[pointcut setValue(int v),
                       int min, int max] {
  before(int v): setValue(v) {
    if ((v < min) || (v > max))
       throw new IllegalArgumentException();
  }
}

// Compose the auxiliary shakein twice, once per axis:
shakein Confined[int minX, int maxX, int minY, int maxY] :=
  ConfinedUpdate[set(int x) && args(int v), minX, maxX] ∘
  ConfinedUpdate[set(int y) && args(int v), minY, maxY];
```

Figure 2.11: A third version of `Confined` (Figure 2.8), using shakein composition. The auxiliary shakein `ConfinedUpdate`, which accepts a pointcut parameter, is composed twice to create the desired result

using shakein composition, we can now apply the same advice twice. In fact, we have here an example for *repeated application* of the auxiliary shakein `ConfinedUpdate`, first for the Y-axis, and then, on the result for the Y-axis.

While pointcut parameters can be simulated in ASPECTJ and similar languages using abstract pointcut definitions, other configuration parameters (such as the **`int`** values used to set the valid range) are not supported, nor is repeated application. Thus, there is no way to avoid code duplication when creating `Confined` as an ASPECTJ-style aspect.

With repeated application, pointcut parameters, and other parameter types, highly configurable shakeins can be defined. For example, we can define role-based security[7] using a shakein which accepts, in addition to its base class, two additional parameters: a pointcut definition $\mathcal{P}$, specifying which join points require a given role, and a role name $\mathcal{R}$. Given this definition, we can specify that certain methods in a bank account class, `Account`, require users to have the "`teller`" credentials:

```
Security[Pt,"teller"]<Account>
```

where P_t is a concrete pointcut definition. An additional application of the same shakein can then be used to specify which methods require client authorization:

```
Security[Pc,"client"]<Security[Pt,"teller"]<Account>>
```

where P_c is another concrete pointcut definition. This process can be repeated as many times as necessary, applying different configuration parameters to different classes. Shakein composition makes it possible to abbreviate the above, by writing, e.g.,

```
shakein MySecurity :=
  Security[Pc,"client"] ∘ Security[Pt,"teller"];
```
(2.1)

and then applying the result to multiple classes as needed: `MySecurity<Account>`, `MySecurity<Branch>`, etc.

[7] A security model in which permission to execute certain methods is granted only if the user belongs to some pre-defined group, or role.

2.3.3 A New Light on Aspect Terminology

By viewing shakeins are operators, taking classes as arguments and producing classes, we can clarify some of the illusive notions and terms used in traditional AOP jargon:

1. ***Aspect Instantiation.*** The semantics of instantiation of aspects, i.e., the reification of aspects at runtime, can be quite confusing. In ASPECTJ there are no less than five different modes of such instantiations: **`issingleton`**—a single instance serving all advised objects; **`pertarget`**—an instance for each advised object, and a very similar variant, **`perthis`**; **`percflow`**—an instance for each execution of an advice, accompanied with a close variant, **`percflowbelow`**. Some of these modes are further parameterized by a pointcut definition.

   In contrast, shakeins, just like mixins and generics, operate on code, and as such, they no longer exist at runtime. (Of course, the result of a shakein is a class which may have runtime instances.)

   Even the intriguing question of static fields defined in an aspect is simplified in the domain of shakeins. Since a shakein generates a re-implementation of a class, each such re-implementation will have its own copy of the static field—just as with static fields defined in mixins.[8]

2. ***Aspect Precedence.*** Different advices from different aspects may be applicable to the same join point. Yet in almost all cases, the order of advice application is significant. For example, if logging and security are applied in the wrong order, there will be no logs of any invocations that failed the security check, in breach of security.

   AOP languages make it possible to define global precedence rules for aspects. However, this declaration is never complete unless all participating aspects are known; and there is no possibility of applying a set of aspects to different classes in a different order.

   As operators, shakeins can easily be applied in a specific order as the need arises.

3. ***Abstract Pointcut.*** In ASPECTJ and other similar languages, an *abstract pointcut* is a named pointcut with no concrete specification. An aspect may include an abstract pointcut and apply advice to it. Such an aspect is said to be *abstract*, in the sense that it cannot be applied without effecting the missing pointcut definition. A *concrete* version of this aspect may be generated by defining a "sub-aspect" for it, which must provide a concrete value to the abstract pointcut.

   This mechanism provides a measure of flexibility; for example, using abstract pointcuts, an off-the-shelf aspect need not declare in advance the exact join points to which it will apply. Yet the terms are confusing when used in conjunction with OOP. An abstract pointcut does not offer dynamic binding to the concrete version, nor does it offer a signature that must be obeyed by all its implementors.

   Standing at the shakein point of view, abstract pointcuts are nothing more than a poor man's replacement for a pointcut parameters.

4. ***Abstract Aspects.*** An aspect is abstract not only when it contains abstract pointcuts, but also when it contains abstract methods. These methods can then be invoked from advice, effectively using the TEMPLATE METHOD [105] design pattern. By defining concrete sub-aspects that implement these methods, the programmer may define a different aspect each time.

[8] This of course applies only to static fields defined in the shakein itself; static fields defined in the base class still have only one copy.

A shakein may also define a method as abstract. Such a definition overrides the method definition in the shakein's class parameter. An application of such a shakein yields an abstract class, which can then be extended by other shakeins and provided with a concrete implementation for the missing methods. However, unlike in the case of abstract aspects, the implementation of the abstract methods in a shakein can optionally be different for each application of the shakein.

5. ***Aspect Inheritance.*** Except in the case of abstract super-aspects, the notion of aspect inheritance is ill-defined, and most aspect languages follow the footsteps of ASPECTJ in prohibiting this. The main reason is that a sub-aspect will clearly share the same pointcut definitions as its parent, and the same advice; does this imply that each advice should therefore be applied *twice* to each matching join point?

   For shakeins, no such problem exists. As shakeins are operators, shakein inheritance is nothing more than operator composition, as in `shakein S3<c> := S2<S1<c>>`.

It is therefore evident that while sharing the flexibility and expressive power of aspects, shakeins, by virtue of being parameterized operators on code, reconcile naturally with the concept of inheritance. They do not exhibit the confusing notions that accompany aspects, and their behavior is easy to understand within the domain of aspect-oriented programming.

In a one-sentence summary of Section 2.3, we may refine the description of shakeins originally presented at the beginning of Section 2.2: *Shakeins make a **configurable re-implementation** of a **class parameter**.*

2.4 Related Work

Obviously, we were not the first to observe the case for using aspects in middleware applications in general, and in J2EE in particular. Indeed, there is a large body of previous work in which aspects are applied in the middleware domain: JAC [190], Lasagne [222], PROSE [192], JAsCo [216], and others.

Unlike shakeins and other similar research work, the *JBoss Application Server* [144] and *Spring Application Framework* [146] are industrial-strength software artifacts employed in production code. The lessons that these two teach are therefore valuable in appreciating the merits of shakeins. We therefore begin by comparing shakeins to these two technologies. Section 2.4.1 compares shakeins with the dynamic aspects of the JBoss Application Server. A comparison with the AOP features of the Spring Application Framework is the subject of Section 2.4.2. Section 2.4.3 provides an overview of other related work.

2.4.1 JBoss Dynamic AOP

Version 4.0 of JBoss was the first implementation of J2EE to integrate AOP support. For technical reasons, the *JBoss AOP* [45] approach features advice- rather than aspect-level granularity, where each advice is encapsulated in what is called an *interceptor* class.

It is telling that the JBoss AOP enhancements of the aspect notion are similar to these of shakeins, including advice composition (*stacks* in the JBoss jargon), parameterized advice, selective and repeated application of advice, explicit and flexible (rather than global) ordering, and configuration parameters. We interpret this as a supporting empirical evidence to the claim that flat-oblivious aspects should be extended in certain ways.

Still, since the application of (standard) advice in JBoss is carried out in situ, destroying the original class, the JBoss approach suffers from the aspect/inheritance schism, instantiation complexity, etc. Perhaps in recognition of these difficulties, JBoss AOP also supports *dynamic AOP*—the ability to apply advice per instance rather than per class. A class must be "prepared" for such a dynamic application, by injecting into its code (at load time) hooks for the potential join points. The class loader consults an XML configuration file for the list of classes to prepare, and the hook locations. It is then possible, at run time, to add or remove interceptors.

In this extent, JBoss's flexibility and repertoire of features is greater than that of shakeins. JBoss offers the ability to *un-apply* an advice at runtime. This feature is missing in shakeins, but can be emulated by testing an on/off flag at the beginning of every advice. (And in Chapter 6 we will see how object re-classification can be used to achieve language-level dynamic AOP with shakeins.)

JBoss's dynamic AOP approach allows advised and unadvised instances to co-exist, yet it suffers from acute problems. Some of these problems are specific to the current version; for example, a dynamically-applied interceptor always intercepts *all* "prepared" join points in the target class, and must test explicitly, at runtime, to see if the current interception is of interest. For example, if two different pointcuts are used to "prepare" a class, for use by two different interceptors, then each join point will be tested twice: once by the interceptor that requires it, and once by the interceptor that does not. This shortcoming will hopefully be fixed in future releases. Other problems, however, are fundamental to the JBoss approach. For example, since interceptors are applied to existing objects, advice cannot be applied to constructors.

The most significant difference between JBoss AOP and shakeins is the approach taken for integration with the base technology. As explained above, shakeins are a language extension, which draws from principles of OO and genericity. In contrast, the variety of features in JBoss AOP is realized by a sophisticated combination of loaders, runtime libraries, and XML configuration files, and without any changes to the compiler (or the JVM). Thus, (probably in answer to Sun's J2EE certification requirements) JBoss AOP is an implementation of aspects through a software framework built over vanilla JAVA. Compliance with these requirements might also explain why standard J2EE services are not implemented as aspects in JBoss, and are therefore not as flexible as they might be.

Since JBoss aspects are not part of the language, a programmer who wishes to exploit aspect features is asked to master a variety of tools, while following the strict discipline dictated by the framework, with little if any compiler checking. For example, the runtime application of advice is achieved using a JAVA API; the compiler is unaware of the involved AOP semantics. As a result, many mistakes (e.g., an attempt to access an invalid method argument), can only be detected at runtime, possibly leading to runtime errors. Other problems, such as a mistyped pointcut (which matches no join points) will not be detected at all.

Thus, in a sense, the offerings of JBoss AOP can be compared to assembly programming: immense flexibility but with greater risks. However, unlike assembly code, performance in JBoss is not at its peak.

The clockwork driving the JBoss framework is visible to the programmer. The programmer *must* understand this mechanism in order to be able use it. This visibility has its advantages. For example, the illusive issue of aspect instantiation in ASPECTJ is clarified by JBoss: since interceptors must be instantiated explicitly prior to their application, the semantics of aspect instance management is left up to the programmer.

To illustrate some of the issues of a framework based implementation of aspects, consider Figure 2.12, which demonstrates how the `Confined` shakein (Figure 2.11) is implemented in JBoss. Figure 2.12(a) depicts a JAVA class which, by obeying the framework rules, can be used as an interceptor. The runtime content of argument `inv` to method `invoke` (lines 12–20) is the only

(a)

```
1 public class Confined implements Interceptor {
2   private int min, max;
3   private String fieldName;

5   public Confined(String fieldName, int min, int max) {
6      this.min = min; this.max = max;
7      this.fieldName = fieldName;
8   }

10  public String getName() { return "Confined"; }

12  public Object invoke(Invocation inv) throws Throwable {
13     FieldWriteInvocation fwi = (FieldWriteInvocation)inv;
14     int v = (Integer)(fwi.getValue());
15     if (fwi.getField().getName().equals(fieldName))
16        if (v < min || v > max)
17           throw new IllegalArgumentException();
18     // proceed to subsequent interceptors/base code:
19     return inv.invokeNext();
20  }
21 }
```

(b)

```
<aop>
  <prepare expr="set(int Point->x) OR set(int Point->y)" />
</aop>
```

Figure 2.12: (a) `Confined` as a JBoss interceptor, and (b) the supporting configuration file

information that the method has on the interception. The method assumes that the join point kind is a field-write, and uses a downcast operation (line 13) to access the specific features of a field-write join point, and in particular the newly-assigned value. This value is obtained (line 14) by downcasting an `Object` to the proper type. Both downcasts will fail if the interceptor is applied to an incorrect join point.

Figure 2.12(b) is the XML code that directs the injection of the hooks intended for this interceptor into class `Point`. To minimize the overhead, only the relevant join points in `Point` were prepared. The interceptor is intimately coupled with the XML, in making tacit assumptions on the join point kind and the argument type; violations of these assumptions are only detected at runtime, e.g. by downcast failures.

To create a shakein-typed instance of `Point`, one may write

```
Point p = Confined[0,1023,0,767]<Point>(); //Shakein version.
```

The JBoss equivalent is a bit longer:

```
Point p = new Point(); //JBoss dynamic AOP version.
InstanceAdvisor advisor = ((Advised)p)._getInstanceAdvisor();
advisor.appendInterceptor(new Confined("x", 0, 1023));
advisor.appendInterceptor(new Confined("y", 0, 767));
```

This code demonstrates more intricacies of implementing aspects by a software framework.

First, we see that advices are applied only after the object was constructed (no refinement of the constructors is possible). Second, since there is no explicit composition operator[9], two interceptors must be manually applied, one per axis. Third, we see that `p` must be casted to the interface type `Advised`. This interface type is implemented by the modified (prepared) version of `Point`; yet the compiler is not aware of this change. If the class was not prepared (e.g., an inconsistency in the XML file), then this cast attempt will fail. Finally, again due to compiler obliviousness, field names are represented as string literals (here, as arguments to the interceptor's constructor). Any mistake in the field name (e.g., writing "`X`" instead of "`x`") will go undetected and result in silent failure. By comparison, an empty pointcut argument to the auxiliary shakein from Figure 2.11 can trigger a compile-time warning.

Figure 2.13 compares the runtime performance of the JBoss and the shakein implementation of aspect `Confined`. The figure depicts the operations throughput of class `Point` in the base

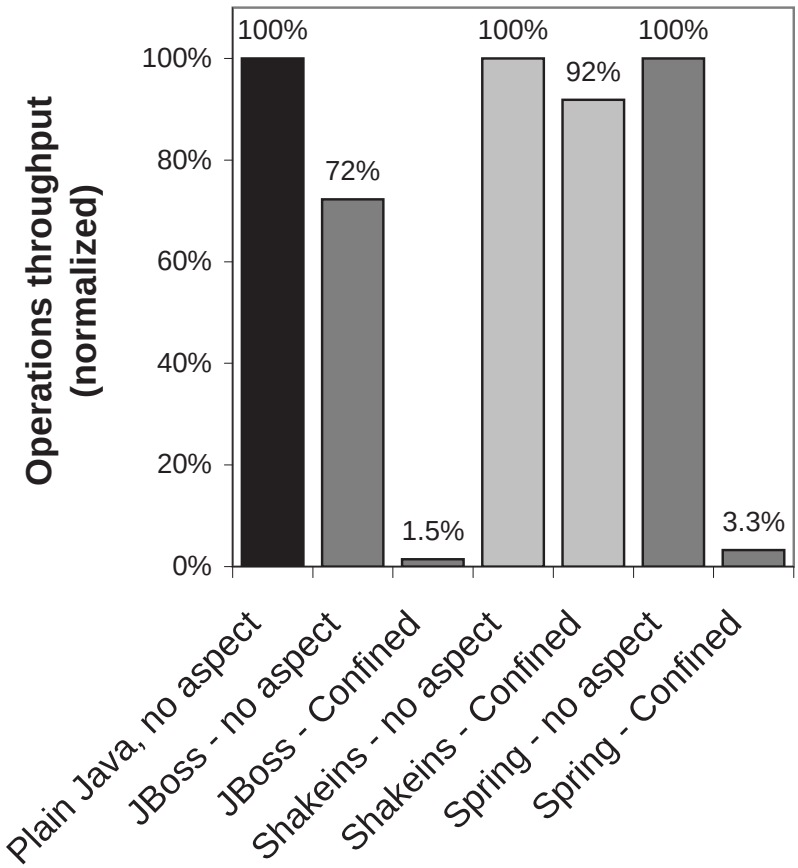


Figure 2.13: Performance degradation of class `Point` with different strategies of applying a "confinement" aspect

implementation and in the different aspectualized versions.[10] We see that the original class suffers no performance penalty in the shakein version. This was expected, because a shakein makes no modifications to the base class itself. The shakein-advised instance, generated by two subclassing operations, is about 8% slower. In contrast, while using JBoss AOP, instances of the original class

[9] Stacks cannot be used here.

[10] Specifically, we used the number of times the sequence of calls ⟨`setX`, `setY`, `moveBy`⟩ can be completed in a time unit.

suffer from a performance impact of about 28% before any advice is applied; this is the overhead introduced by the join point hooks. Once applied, the reflection-based interceptors slow the JBoss version to 1.5% of the original throughput.

There are two main sources of performance degradation in the JBoss implementation. *Time wise*, hooks slow down code, and this slowdown occurs even if no advices are applied to the receiver instance. (In class `Point`, this slowdown was by 28%.) Moreover, even a non-prepared class may be slowed down, if its code matches, e.g., a join point of a **`call`** to a prepared class.

Additional slowdown is caused by the advice having to use reflection-like objects in order to learn about the join point. The invocation object must be downcast to the specific invocation type (`FieldWriteInvocation` in Figure 2.12(a)), and arguments must be downcast from `Object` references to their specific types. As we have seen, this slowdown was by more than an order of magnitude in the case of `Point`. To be fair, we expect a more modest relative performance degradation in classes that do more substantial computation.

Space wise, the advice (whenever a join point is reached) is reified in a runtime object. The memory consumed by this object must be managed, thereby leading to additional slowdown. The invocation itself is reified in a number of objects (metadata, argument wrappers, arguments array, etc.) which add to the space and time overheads of the implementation. A quick inspection of Figure 2.12(a) reveals that there are at least four objects generated in reifying the join point.

2.4.2 Spring AOP

The Spring Application Framework is an "inversion of control" [101] container, used for applying services to standard JAVA objects. It is often used in conjunction with a J2EE server for developing enterprise applications; however, Spring provides alternatives to many of the J2EE services, in a more flexible and developer-friendly manner.

Objects in Spring are "beans", all obtained via a centralized factory which is configured using an XML file. This XML file specifies what properties should be set and what services applied to each bean type. Developers can choose from a wide range of pre-defined services (e.g., Hibernate-based persistence [22]) or define their own. New services (as well as existing ones) are defined using Spring's AOP facilities. Spring also supports the integration of standard ASPECTJ aspects, but we focus here on the framework's internal AOP support.

Much like shakeins, AOP in Spring is based on the generation of new classes. When advice is applied to a class, a new class is generated, which either implements the same interfaces as the base class or else extends it as a subclass. Thus, Spring enjoys several of the benefits of shakeins, including selective and non-destructive application, explicit and flexible ordering, configuration parameters (to an extent), and repeated application. In particular, due to the non-destructive application of aspects, there is no performance penalty to instances of the original class, which remains unmodified (see Figure 2.13).

However, beyond these similarity, there are several differences of note between the Spring and shakein approaches. Advice in Spring is manifested as an interceptor class, which is invoked whenever an advised method is executed. Pointcuts are also manifested as classes, and interrogated at runtime to find out which methods should be advised. Much as in JBoss, the mechanism relies on a sophisticated combination of libraries and configuration files, with no changes to the language itself. Therefore, Spring AOP shares much of the tolls noted for JBoss AOP, including similar space and time complexities (with the exception of hooks-induced slowdowns). Additional performance penalties are caused by the need to evaluate pointcuts at runtime, as well as the runtime generation of subclasses.

As a design decision, Spring AOP only support method invocation join points (and that, only for non-**`private`** methods). In our tests, the lack of support for field access join points implied that the `Confined` aspect had to be made explicitly aware of each of the `Point` methods that can

update the point's coordinates; in particular, the advice had to re-create the logic for the `moveBy` method, as shown in Figure 2.14 (lines 20–28). The Spring point of view contends that this would

```
1 public class Confined implements MethodBeforeAdvice {
2    private int minX, maxX, minY, maxY;

4    public void setMinX(int mX) { minX = mX; }
5    public void setMaxX(int mX) { maxX = mX; }
6    public void setMinY(int mY) { minY = mY; }
7    public void setMaxY(int mY) { maxY = mY; }

9    public void before(Method m, Object[] args, Object target) {
10      if (m.getName().equals("setX")) {
11         int newX = ((Integer)args[0]).intValue();
12         if ((newX > maxX) || (newX < minX))
13            throw new IllegalArgumentException();
14      }
15      else if (m.getName().equals("setY")) {
16         int newY = ((Integer)args[0]).intValue();
17         if ((newY > maxY) || (newY < minY))
18            throw new IllegalArgumentException();
19      }
20      else if (m.getName().equals("moveBy")) {
21         Point p = (Point)target;
22         int newX = p.getX() + ((Integer)args[0]).intValue();
23         if ((newX > maxX) || (newX < minX))
24            throw new IllegalArgumentException();
25         int newY = p.getY() + ((Integer)args[1]).intValue();
26         if ((newY > maxY) || (newY < minY))
27            throw new IllegalArgumentException();
28      }
29   }
30 }
```

Figure 2.14: `Confined` as a Spring advice class

not have been needed, had `moveBy` relied on `setX` and `setY` to update the fields, rather than using direct access (recall Figure 2.1). But from this very claim we must conclude that the Spring aspect is fragile with respect to changes in the implementation of `Point`; should `moveBy` be updated to rely on the setter methods, the advice must be accordingly updated. A non-fragile aspect implementation must rely on examining the control-flow at runtime, which a noticeable performance hit.

Like shakeins and JBoss aspects, Spring aspects support parameterization. However, the parameter values are passed in XML setup files that are used to initialize and configure the aspects (and other beans), rather than in source code. For example, the values for the four parameters of the `Confined` aspect appear in the XML fragment in Figure 2.15. This is a deliberate design choice, part of Spring's "inversion of control", or "dependency injection" philosophy. While this approach has many clear benefits (as detailed by Fowler [101]), it also means that any type mismatch in passing parameter values will only be detected at runtime. Still, such problems will be detected immediately upon program startup, when the configuration file is parsed, and are thus

```
<bean id="confinedInterceptor" class="Confined">
   <property name="minX"><value>0</value></property>
   <property name="maxX"><value>1023</value></property>
   <property name="minY"><value>0</value></property>
   <property name="maxY"><value>767</value></property>
</bean>
```

Figure 2.15: Configuring the Spring aspect (fragment)

unlikely to go undetected during development.

In our benchmarks (Figure 2.13), the Spring-based `Confined` aspect (which was not created using composition, due to its asymmetry with regard to `moveBy`) was over twice as fast as the JBoss version, but still much slower than the shakeins-based version.[11] The performance penalty was caused mainly by the need to analyze the method arguments and the join point (in particular, the method being executed) at runtime.

2.4.3 Other Related Work

The AspectWerkz [34] project, which was discontinued and integrated into ASPECTJ, introduces aspects to JAVA with no language change. Much like JBoss AOP, AspectWerkz AOP works by using XML files to dictate the weaving of hooks, at load time, to classes. These hooks function as runtime tests for the application of aspects. AspectWerks shares both the strengths and weaknesses of the JBoss AOP approach; aspects can be added or removed dynamically, on an object-based (rather than class-based) level, providing considerable flexibility. However, the performance of every instance of the original class, regardless of the actual use of aspects, suffers.

Several works suggest using *wrappers* [105], rather than subclasses, as a mechanism for intercepting method calls. For example, *JAC* (Java Aspect Components) [190] is a component-based framework aimed at enterprise software development with JAVA. JAC components, like Enterprise JavaBeans, reside in containers; however containers can also contain *aspect components*. Aspect components can be preloaded into the container or loaded at runtime.

The wrapper-based approach implies that aspects can be changed dynamically; the wrapper maintains a list of advice applied to the wrapped object, and this list can be modified at will. However, the approach also implies an increased memory footprint: each advised object is represented by $a + 2$ objects in memory, where a is the number of applied advice, plus one wrapper and the object itself.[12] A more significant limitation of the wrapper approach relates to possible advice targets. Only calls to public methods can be intercepted by a wrapper. While this limitation is shared, e.g., by the Spring framework as a design decision, the wrapper approach is further limited by the fact that only *external* calls to advised methods can be trapped. Whenever one public method of the wrapped object invokes another such method, the invocation is direct and does not pass via the wrapper, thereby skipping any advice applied to the target method. This happens because, while all external references to the object are in fact references to its wrapper, the **`this`** reference used internally remains a direct pointer. The subclassing approach suggested in this work does not suffer from this limitation.

Lasagne [222] is another wrapper-based approach for introducing aspects to middleware frameworks (and JAVA applications in general). The system presents a higher-level approach than

[11] The class shown in Figure 2.14 is somewhat simplified; the actual tests were performed using a more optimized version of this code. Testing was done using Spring version 1.2.6.

[12] In practice, the *list* of aspects maintained by the wrapper is also an object. However, several advice objects can be shared by several wrappers, if a given advice includes no object-specific state.

JAC. Since it is based on wrappers, Lasagne is non-invasive, works on a per-instance level, and allows for dynamic aspect application or removal. However, it shares the same limitations of the wrapper approach, and has been criticized for carrying a significant performance overhead [28].

2.5 Summary

Shakeins are a novel, aspect-like programming construct, with three distinguishing characterization: explicit application semantics, configuration parameters, and restriction on the changes to a class to be re-implementation only. We have seen that the shakein construct integrates well with the object model, by distinguishing the five facets of a class: type, forge, implementation, mold and mill, and explaining how these can be modified by shakeins. It was shown that if we adher to the principle that no variables of shaked classes are allowed, then the construct can be implemented with current JVMs, and using the existing inheritance model of JAVA.

Shakeins enjoy the advantages of the parameterized notation of mixins, while offering a simple answer to the accidental overriding problem. Thanks to the pointcuts semantics of aspects, shakeins become more expressive than mixins in the sense that they can "examine" the internals of their target classes. Conversely, thanks to the parameterized semantics and the object model integration, shakeins simplify some of the more subtle issues of aspects, including aspect inheritance, instantiation, and abstract aspects.

As an important application and prime motivation for actual use of this construct, the following chapter presents ASPECTJ2EE, an AOP programming language, similar in syntax to ASPECTJ, with aspects that have a shakein semantics. We will use the ASPECTJ2EE design to show that the shakeins semantics integrates well with the current architecture of J2EE servers.

Shakeins require a language for pointcut expressions. While the examples in this chapter and the next use the standard ASPECTJ pointcut language, a superior alternative, *JTL*, will be presented in Chapter 4. We will also see how JTL can be used to express limitations on the types to which a shakein can be applied.

To properly integrate shakeins with a programming framework, we must find a way to ensure that every obtained instance (or, in some cases, some obtained instances) of a given class is replaced by a different, shakein-generated class that implements the same type. For example, we might wish that any attempt to obtain an instance of class `Account` will yield an instance of `Secure<Account>`. In ASPECTJ2EE, we will use the standard J2EE mechanism of home objects to solve this problem. This mechanism is nothing but a variant of the FACTORY METHOD [105] design pattern. A more elegant solution, namely *factories*, will be presented in Chapter 5.

Finally, Chapter 6 suggests *object evolution* as a language extension that will enable shakeins to support dynamic aspect facilities.

Chapter 3

AspectJ2EE

High thoughts must have high language.

— Aristophanes, *The Frogs*

Having presented the advantages of shakeins, the time has come for evaluating their usefulness. To do so, we would like to go beyond the small examples presented in the previous chapter. The more thorough examination is in the context of a real life application, and in particular, with respect to the J2EE framework.

To demonstrate the applicability of shakeins to this domain, we introduce the ASPECTJ2EE language, which shows how they can be used to bring the blessings of AOP to the J2EE framework. ASPECTJ2EE is geared towards the generalized implementation of J2EE application servers and of applications within this framework.

As the name suggests, ASPECTJ2EE borrows much of the syntax of ASPECTJ. The semantics of ASPECTJ2EE is adopted from shakeins, while adapting these to ASPECTJ. To maintain maximal syntactical similarity, shakeins in ASPECTJ2EE are defined using the keyword **`aspect`**. There are, however, several syntactical differences, mostly due to the fact that "aspects" in ASPECTJ2EE can be parameterized.

In the initial design of ASPECTJ2EE, parameter passing and the application of shakeins are not strictly part of the language. They are governed mostly by external XML configuration files, a-la J2EE deployment descriptors.

A distinguishing advantage of this new language is that it can be smoothly integrated into J2EE implementations without breaking their architecture, thereby fulfilling the conceptual marriage discussed in Section 1.1.3. This smooth integration is achieved by generalizing the existing process of binding services to user applications in the J2EE application server into a novel *deploy-time* mechanism of weaving aspects. Deploy-time weaving is superior to traditional weaving mechanisms in that it preserves the object model, has a better management of aspect scope, and presents a more understandable and maintainable semantic model. Also, deploy time weaving stays away from specialized JVMs and bytecode manipulation for aspect-weaving.

Standing on the shoulders of the J2EE experience, we can argue that shakeins in general, and ASPECTJ2EE in particular, are suited to systematic development of enterprise applications. Unlike existing attempts to add AOP functionality to J2EE application servers, the ASPECTJ2EE approach is methodical. Rather than add aspects as an additional layer, unrelated to the existing services, our approach is that even the standard services should be implemented on top of the AOP groundwork. Using ASPECTJ2EE, the fixed set of standard J2EE services is replaced by a library of core aspects. These services can be augmented with new ones, such as logging or performance monitoring.

It should be noted that AspectJ2EE has its limitations compared to more traditional implementations of aspects; in particular, it is not at all suited to low-level debugging or nit-picking logging. For example, access to *non-private* fields by classes other than the defining class is not a valid join point in AspectJ2EE. However, it is not for these tasks that AspectJ2EE was designed, and it is highly suited for dealing with large systems and global aspect-oriented programming. In the kind of systems we are interested in, the field access restriction example does not mature into a hurdle; field management can be always be decomposed into getter and setter methods, and in fact *must* be decomposed this way in J2EE applications, where fields are realized as *attributes* with proper join points at their retrieval and setting.

We stress that unlike previous implementations of aspects within the standard object model, AspectJ2EE does not merely support "before" and "after" advices and "method execution" join points. AspectJ2EE supports "around" advices, and a rich set of join points, including control-flow based, conditional, exception handling, and object- and class-initialization.

In contrast to its lack of suitability for low-level tasks, AspectJ2EE boasts specific support for high-level tasks appropriate to enterprise applications. In particular, it has special support for the composition of aspects that are scattered across program tiers (*tier-cutting concerns*), such as encryption, data compression, and memoization.

Historically, the developers of enterprise applications are slow to adopt new technologies; a technology has to prove itself again and again, over a long period of time, before the maintainers of such large-scale applications will even consider adopting it for their needs. It is not a coincidence that many large organizations still use and maintain software developed using some technologies, such as COBOL , that other sectors of the software industry view as thoroughly outdated. The huge investment in legacy code slows the adoption of new technologies.

We believe that the fact that AspectJ2EE, by its reliance on shakeins, preserves the standard object model, while also relying on existing J2EE technologies, should contribute to widespread adoption of this new technology in the middleware software domain.

Chapter outline. Section 3.1 presents an overview of the AspectJ2EE language. Section 3.2 discusses the weaving process, which is part of any AOP system, and shows how the deployment step of J2EE applications can be extended into a weaving mechanism. Next, Section 3.3 presents the language in greater detail, including a discussion of implementing shakeins by means of subclassing, and an overview of the AspectJ2EE class library. Finally, Section 3.4 presents a few innovative uses for AspectJ2EE's unique support for the aspectualization of tier-cutting concerns. Section 3.5 concludes.

3.1 An Overview of AspectJ2EE

The AspectJ2EE language was designed as a shakeins-based alternative to AspectJ. To this end, its syntax parallels that of AspectJ to a large extent.[1] An `aspect` structure in AspectJ2EE defines not an AspectJ-style aspect, but rather a shakein. Thus, it cannot, for example, contain instantiation instructions; as detailed in Section 2.3.3, shakeins are never instantiated.

The main issues in which the AspectJ2EE language differs from AspectJ are:

1. *Aspect targets.* AspectJ can apply aspects to any class, whereas in AspectJ2EE aspects can be applied to *enterprise beans* only, i.e., those modules to which J2EE services are applied. (This selective application is made possible by the shakein semantics, which al-

[1] AspectJ2EE was designed to be syntactically similar to AspectJ version 1.2, the most recent version at the time of AspectJ2EE's design.

ways have a designated target.) In OOP terminology these beans are the core classes of the application, each of which represents one component of the underlying data model.

As demonstrated by the vast experience accumulated in J2EE, aspects have great efficacy precisely with these classes. We believe that the acceptance of aspects by the community may be improved by narrowing their domain of applicability, which should also benefit understandability and maintainability.

In this sense, ASPECTJ2EE follows the steps of the Spring Application Framework, which likewise treats beans as the core components in enterprise applications and allows aspects to be applied only to beans. Also like Spring, beans in ASPECTJ2EE are taken in a more liberal sense than the J2EE specification required; for example, ASPECTJ2EE beans, unlike standard EJBs, need not contain any lifecycle methods. Where needed, such methods can be automatically added by applying relevant aspects to the bean. In fact, most regular JAVA classes can be used as beans in ASPECTJ2EE applications.

Still, it should be stressed that the fact ASPECTJ2EE aspects can only be applied to beans is not a limitation of the shakein concept, but rather an ASPECTJ2EE design decision.

2. *Weaving method.* Weaving the base class together with its aspects in ASPECTJ2EE relies on the same mechanisms employed by J2EE application servers to combine services with the business logic of enterprise beans. This is carried out entirely within the dominion of object oriented programming, using the standard JAVA language, and an unmodified JVM. Again, this is made possible by the shakein semantics.

   In contrast, different versions of ASPECTJ used different weaving methods relying on pre-processing, specialized JVMs, and dedicated byte code generators, all of which deviate from the standard object model.

3. *Aspect parametrization.* Since the aspects in ASPECTJ2EE are shakeins, they take three kinds of parameters: pointcut definitions, types, and literal values. Parameterized aspects can be applied to EJBs by providing (in the EJBs deployment descriptor) a concrete value for each parameter, including concrete pointcut definitions. Pointcut parameters provide significant flexibility by removing undesired cohesion between aspects and their target beans, and enables the development of highly reusable aspects. It creates, in ASPECTJ2EE, the equivalent of Caesar's [171] much-touted separation between aspect implementation and aspect binding.

   Other aspect parameter types also greatly increase aspect reusability and broaden each aspect's applicability.

4. *Support for tier-cutting concerns.* ASPECTJ2EE is uniquely positioned to enable the localization of concerns that cross not only program modules, but program tiers as well. Such concerns include, for example, encrypting or compressing the flow of information between the client and the server (processing the data on one tier and reversing the process on another). Even with AOP, the handling of tier-cutting concerns requires scattering code across at least two distinct program modules. We show that using ASPECTJ2EE, many tier-cutting concerns can be localized into a single, coherent program module.

These are the key *conceptual* differences between ASPECTJ2EE and ASPECTJ. Other, mainly syntactical differences are discussed below in Section 3.3.

ASPECTJ2EE does not impose constraints on the base code, other than a subset of the dictations of the J2EE specification [82, 202] on what programmers must, and must not, do while defining EJBs. The dictations that are of importance to ASPECTJ2EE are that instances must

be obtained via the Home interface, rather than by directly invoking a constructor or any other user-defined method; and that business methods must not be **`final`** or **`static`**. Other requirements for EJBs, such as the prohibition on defining methods as **`synchronized`** or using file I/O mechanisms, are relaxed. In particular, the EJB 2.x requirement that EJB classes must be **`abstract`** is removed. If needed, specific services (aspects) can present specific limitations to their target classes. For example, a load-balancing aspect may require that the base class contains no **`synchronized`** business methods.

J2EE application servers offer the developer only minimal control over the generation of support classes. ASPECTJ2EE, however, gives a full AOP semantics to the deployment process. With deploy-time weaving, described next, the main code is unmodified, both at the source and the binary level. Further, the execution of this code is unchanged, and can be carried out on any standard JVM.

3.2 Weaving, Deployment and Deploy-Time Weaving

Now that the theoretical foundation of the shakeins construct was established, and that we understand how and why it may be useful in the context of middleware frameworks, the time has come to combine the two. The first step is in describing how deployment, a basic technique of J2EE, can be generalized for the process of weaving aspects (specifically, shakeins) into an application.

Section 3.2.1 explains weaving. Deployment is the subject of Section 3.2.2. Weaving of shakeins onto EJBs is discussed in Section 3.2.3, while Section 3.2.4 generalizes this process to arbitrary classes.

3.2.1 Weaving

Do you hear the wind? It's not dying,
It's singing, weaving a song ...

— John Ashbery, *Train Rising Out of the Sea*

Weaving is the process of inserting the relevant code from various aspects into designated locations, known as *join points*, in the main program. In their original presentation of ASPECTJ [150], Kiczales *et al.* enumerate a number of weaving strategies: "*aspect weaving can be done by a special pre-processor, during compilation, by a post-compile processor, at load time, as part of the virtual machine, using residual runtime instructions, or using some combination of these approaches.*" Each of these weaving mechanisms was employed in at least one AOP language implementation. As mentioned before, our implementation of shakeins use its own peculiar *deploy-time weaving* strategy. In this section we motivate this strategy and explain it in greater detail.

We first note that the weaving strategies mentioned in the above quote transgress the boundaries of the standard object model. Patching binaries, pre-processing, dedicated loaders or virtual machines, will confuse tools such as debuggers, profilers and static analyzers, and may have other adverse effects on generality and portability.

Further, weaving introduces a major *conceptual* bottleneck. As early as 1998, Walker, Baniassad and Murphy [227] noted the disconcert of programmers when realizing that merely reading a unit's source code is not sufficient for understanding its runtime behavior. Further, Laddad [158, p. 441] notes that in ASPECTJ, the runtime behavior cannot be deduced even by reading *all* modules, including both classes and aspects, since the application of aspects to the main code is governed by the command used to invoke the compiler.

The remedy suggested by Constantinides, Bader, and Fayad in the *Aspect Moderator* framework [70] was restricting weaving to the dominion of the OOP model. In their suggested framework, aspects and their weaving are realized using pure object-oriented constructs. Thus, every aspect-oriented program can be presented in terms of the familiar notions of inheritance, polymorphism and dynamic binding. Indeed, as Walker *et al.* conclude: "*programmers may be better able to understand an aspect-oriented program when the effect of aspect code has a well-defined scope*".

Aspect Moderator relies on the PROXY design pattern [105] to create components that can be enriched by aspects. Each core class has a proxy which manages a list of operations to be taken before and after every method invocation. As a result, join points are limited to method execution only, and only `before()` and `after()` advices can be offered. Another notable drawback of this weaving strategy is that it is *explicit*, in the sense that every advice has to be manually registered with the proxy. Registration is carried out by issuing a plain JAVA instruction—there are no external or non-JAVA elements that modify the program's behavior. Therefore, long, tiresome and error-prone sequences of registration instructions are typical to Aspect Moderator programs.

A better strategy of implementing explicit weaving is that this code is generated by an automatic tool from a concise specification. The shakein weaving mechanism gives in essence this tool. However, rather than generate explicit weaving code for a proxy, it generates a woven version of the code in a *newly generated subclass*. By replacing the proxy pattern with the notion of subclassing, it is also able to handle advice types other than `before()` and `after()`, and handle a richer gamut of join point types, as detailed below in Section 3.3.3.

Thus, shakeins do not use any of the obtrusive weaving strategies listed above. Rather, the mechanism employs a weaving strategy that *does not break* the object model. Instead of modifying binaries (directly, or by pre-processing the source code), the application of a shakein to a class results in an "under the hood" generation of a new class that inherits from, rather than replaces, the original. The new class provides an alternative realization, a re-implementation, of the same type; it does not introduce a new type, since there are no visible changes to the interface. This re-implementation is generated by advising the original one with the advice contained in the shakein.

Clearly, there are limitations to the approach of implementing shakeins as subclasses. The main such limitation is that Proposition 1 does not hold in the general case. Below we will show that in the particular case of EJBs in J2EE, this restriction does not arise, because access to EJBs is through interfaces.

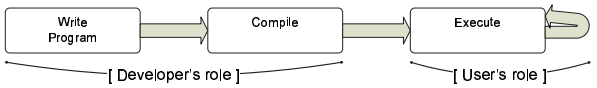
3.2.2 Deployment

J2EE offers a unique opportunity for generating the subclasses required for the weaving of shakeins. Figure 3.1 compares the development cycle of traditional and J2EE application. We see in the figure that *deployment* is a new stage in the program development process, which occurs after compilation but prior to execution. It is unique in that although new code is generated, it is not part of the development, but rather of user installation.

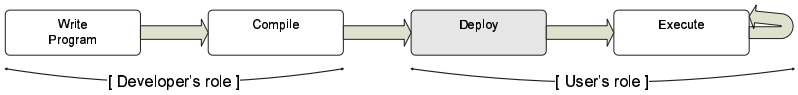
Deployment is the magic by which J2EE *services*, such as security and transaction management, are welded to applications. The generation of sub- and support classes is governed by *deployment descriptors* [202, Sec. 2.11.4], which are XML configuration files.

The idea behind deploy-time weaving is to extend this magic, by placing the shakein semantics in government of this process. As shakeins are based on straightforward inheritance, this extension also simplifies the structure of and inter-relationships between the generated support classes.

Technically, *deployment* is the process by which an application is installed on a J2EE application server. Having received the application binaries, deployment involves generating, compiling and adding additional support classes to the application. For example, the server generates *stub* and *tie* (skeleton) classes for all classes that can be remotely accessed, in a manner similar to, or even



(a) Traditional program development steps



(b) J2EE program development steps

Figure 3.1: Traditional and J2EE program development steps

based on, the remote method invocation (RMI) compiler, `rmic` [212]. Even though some J2EE application servers generate support class binaries directly (without going through the source), these always conform to the standard object model.

We must study some of the rather mundane details of deployment in order to understand how it can be generalized to do weaving. To do so, consider first Figure 3.2, which shows the initial hierarchy associated with an ACCOUNT bean, used to represent a bank account in a financial application.[2]

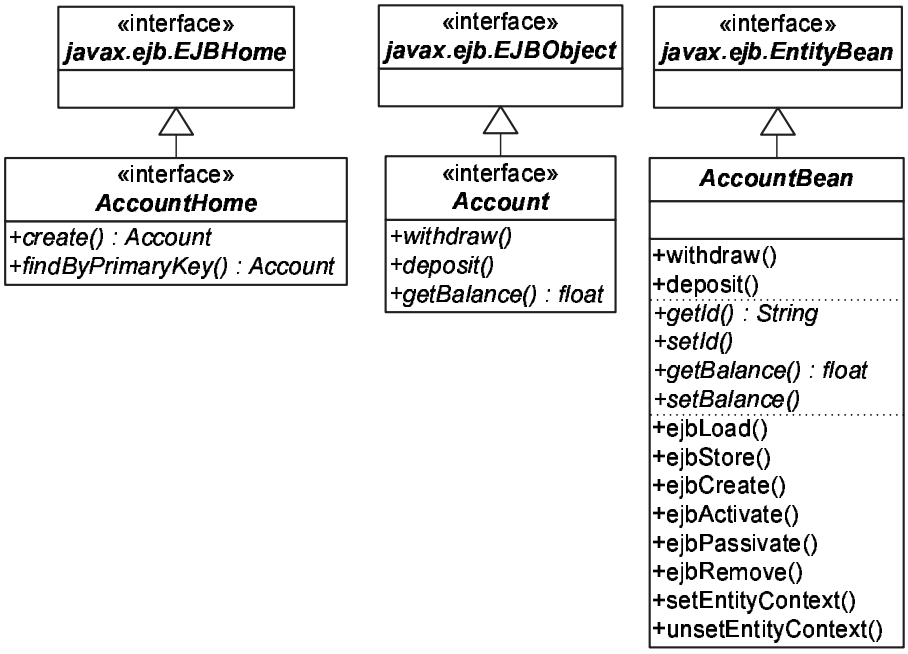


Figure 3.2: Classes created by the programmer for defining the ACCOUNT EJB

At the center of the figure, we see interface `Account`, which inherits from `javax.ejb.EJBObject`. This interface is written by the developer in support of the remote interface to the bean.[3] This interface is where all client-accessible methods are declared. In the example, there are three such methods: `withdraw()`, `deposit()`, and `getBalance()`. Interface `Account`

[2]The deployment process discussed here relates to versions 2.0 and 2.1 of the EJB specification [82].

[3]For the sake of simplicity, we assume that ACCOUNT has a remote interface only, even though since version 2.0 of the EJB specification, beans can have either a local interface, a remote interface, or both.

resides at the *client side*.

On the right hand side of the figure, we see abstract class `AccountBean`, inheriting from `javax.ejb.EntityBean`. The J2EE developer's main effort is in coding this class, which will reside at the *server side*. There are three groups of methods in the bean class:

1. *Business Logic.* The first group of methods in this class consists the implementation of business logic methods. These are `deposit()` and `withdraw()` in the example.

   It is convenient to think of class `AccountBean` as implementing interface `Account`. Technically, and as shown in the figure, no **`implements`** relationship needs to exist between the two. The reason is that the client and the server run different virtual machines, which usually reside on physically remote locations.

2. *Accessors of Attributes.* In addition to regular fields, an EJB has *attributes*, which are those fields of the class that will be governed by the persistence service in the J2EE server. Each attribute *`attr`* is represented by abstract setter and getter methods, called `set`*`Attr`*`()` and `get`*`Attr`*`()` respectively. Attributes are not necessarily client-accessible.

   In the example, there are four such accessors, indicating that the bean ACCOUNT has two attributes: `id` (the primary key) and `balance`. Examining the `Account` interface we learn that `id` is invisible to the client, while `balance` is read-only accessible.

3. *Lifecycle.* The third and last method group comprises a long list of mundane lifecycle methods, such as `ejbLoad()` and `ejbStore()`, most of which are normally empty when the persistence service is used. Even though sophisticated IDEs can produce a template implementation of these, they remain a developer's responsibility, contaminating the functional concern code. Later we shall see how deploy-time weaving can be used to remove this burden.

Finally, at the left hand side of Figure 3.2, we see interface `AccountHome`, which declares a FACTORY [105] of this bean. Clients can only generate or obtain instances of the bean by using this interface.

Concrete classes to implement `AccountHome`, `Acount` and `AccountBean` are generated at deployment time. The specifics of these classes vary with the J2EE implementation. Figure 3.3 shows some of the classes generated by IBM's WebSphere Application Server (WAS) [139] version 5.0 when deploying this bean. Examining the figure, we see that it is similar in structure to Figure 3.2, except for the classes, depicted in gray, that the deployment process created: `ConcreteAccount_b7e62f65` is the *concrete bean class*, implementing the abstract methods defined in `AccountBean` as setters and getters for the EJB attributes. Instances of this class are handed out by class `EJSRemoteCMPAccountHome_b7e62f65`, which implements the factory interface `AccountHome`.

Finally, `_Account_Stub`, residing at the client side, intercommunicates with `ConcreteAccount_b7e62f65`, which resides at the server side.

In support of the ACCOUNT bean, WAS deployment generates several additional classes which are not depicted in the figure: a stub for the home interface, ties for both stubs, and more. Together, the deployment classes realize various services that the EJB container provides to the bean: persistence, security, transaction management and so forth. However, as evident from the figure, all this support is provided within the standard object oriented programming model.

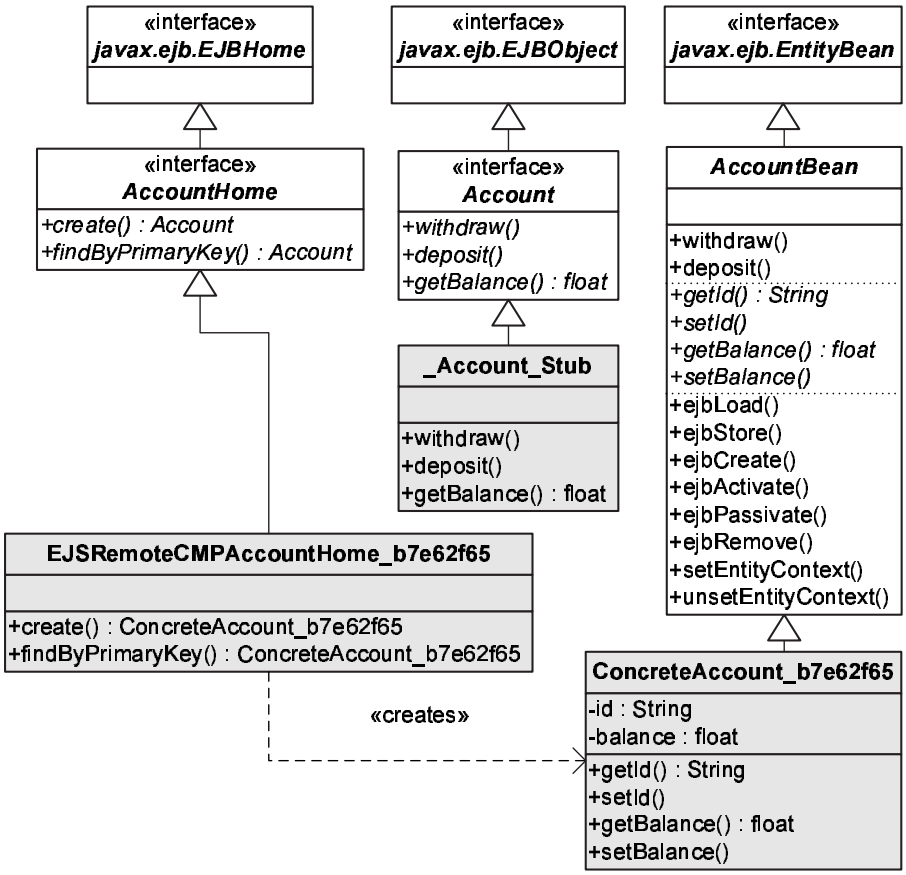


Figure 3.3: ACCOUNT classes defined by the programmer, and support classes (in gray) generated by WAS 5.0 during deployment

3.2.3 Deployment as a Weaving Process for EJBs

You know my methods. Apply them!

— Sherlock Holmes, in Arthur Conan Doyle's
The Hound of the Baskervilles, Chapter 1

Having understood the process of deployment and the generation of classes in it, we can now explain how deployment can be used as a weaving process. Consider first the ordered application of four standard ASPECTJ2EE shakeins (`Lifecycle`, `Persistence`, `Security`, and `Transactions`) to the bean ACCOUNT. (Such a case is easier than the more general case, in which the target class is not an EJB. We will discuss this issue below.)

Weaving by deployment generates, for each application of an aspect (i.e., a shakein) to a class, a subclass of the target. This subclass is called an *advised class*, since its generation is governed by the advices given in the aspect. Accordingly, the sequence of applications under consideration will generate four advised classes.

Figure 3.4 shows the class hierarchy after the deployment tool generated these four class in support of the shakein application expression

```
Transactions<Security<Persistence<Lifecycle<Account>>>>.
```

Comparing this figure to Figure 3.3, we see first that the class `AccountBean` was shortened

by moving the lifecycle methods to a newly defined class, `AdvAccount_Lifecycle`. The

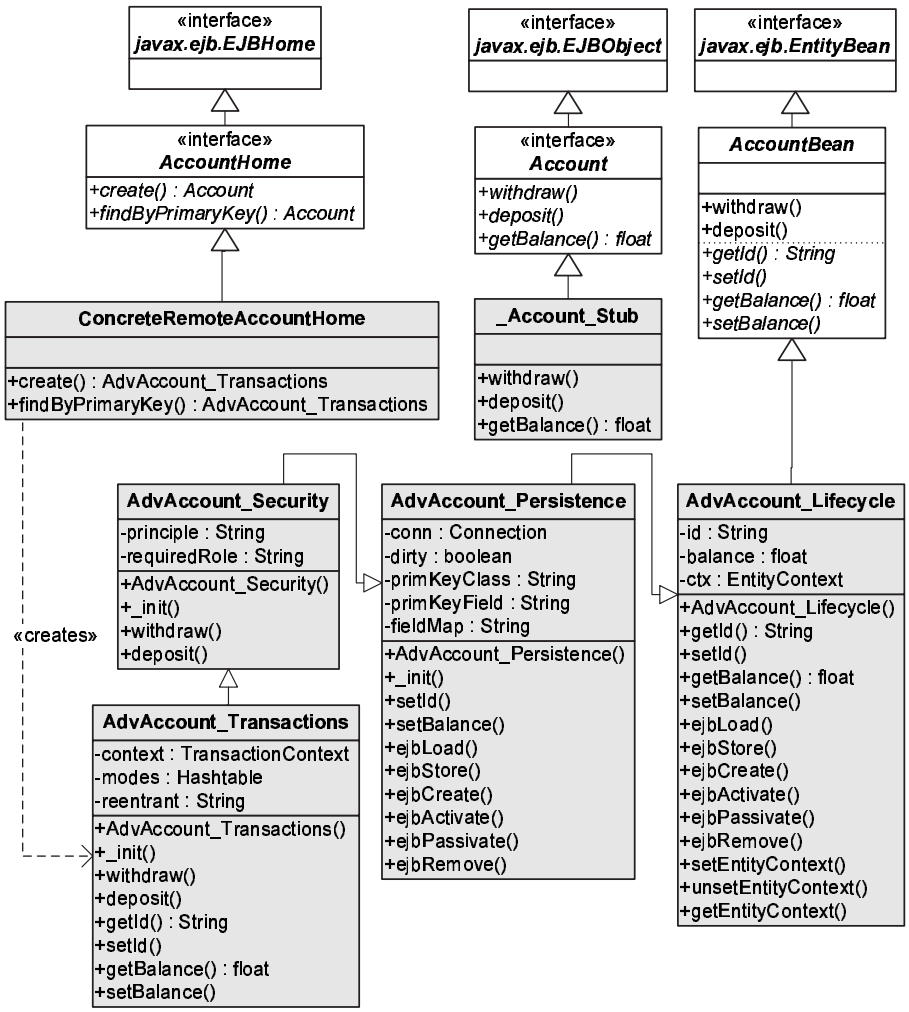


Figure 3.4: ACCOUNT classes defined by the programmer, and support classes (in gray) generated by ASPECTJ2EE during deployment

shakein `Lifecycle` made it possible to eliminate the tiring writing of token (and not always empty) implementations of the lifecycle methods in each bean. All these implementations are packaged together in a standard `Lifecycle` aspect.[4]

`AdvAccount_Lifecycle` is the advised class realizing the application of `Lifecycle` to ACCOUNT. There are three other advised classes in the figure: `AdvAccount_Persistence`, `AdvAccount_Security` and `AdvAccount_Transactions`, which correspond to the application of aspects `Persistence`, `Security` and `Transactions` to ACCOUNT.

The sequence of aspect applications is translated into a chain of inheritance of advised classes, starting at the main bean class. The *root advised class* is the first class in this chain (`AdvAccount_Lifecycle` in the example), whereas the last class (`AdvAccount_Transactions` in the example) is known as the *terminal advised class*. Fields, methods and inner classes defined in an aspect are copied to its advised class. *Advised methods* in this class are generated automatically based on the advices in the aspect.

[4]The lifecycle methods are declared in the interface `javax.ejb.EntityBean`. Hence, implementing them in a shakein does not change the type of class `AccountBean`.

We note that although all the advised classes are concrete, only instances of the terminal advised class are created by the bean factory (the generated EJB home). In the figure for example, class `ConcreteRemoteAccountHome` creates all ACCOUNTs, which are always instances of `AdvAccount_Transactions`.

It may be technically possible to construct instances of this bean in which fewer aspects are applied; there are, however, deep theoretical reasons for preventing this from happening. Suppose that a certain aspect applies to a software module such as a class or a routine, etc., in all but some exceptional incarnations of this module. Placing the tests for these exceptions at the point of incarnation (routine invocation or class instantiation) leads to scattered and tangled code, and defeats the very purpose of AOP. The bold statement that some accounts are exempt from security restrictions should be made right where it belongs—as part of the definition of the security aspect! Indeed, J2EE and other middleware frameworks do not support conditional application of services to the same business logic. A simple organization of classes in packages, together with JAVA accessibility rules, enforce this restriction and prevents clients from obtaining instances of non-terminal advised classes.

Still, some middleware frameworks, such as Spring, do allow developers to define different beans from the same base business logic. This can be done in ASPECTJ2EE as well, by defining different `<entity>` elements which apply different aspects to the same `<ejb-class>` (see the discussion of the deployment descriptor syntax in Section 3.3.2).

3.2.4 Deploy Time Weaving for General Classes

We just saw that deploy time weaving generates, at deployment time, an advised class for each application of an aspect of ASPECTJ2EE. Let us now consider the more general case, in which the target class is not an EJB.

It is instructive to compare the advising of EJBs (Figure 3.4) with the general structure of shakein classes, as depicted in Figure 2.5. We see that the diagrams are similar in making each aspect application into a JAVA class. However, Figure 3.4 adds two factors to the picture: First, the generation of instances of ACCOUNT is controlled by an external *factory class*. Second is the fact that the class `Account` is abstract.

Together these two make one of the key properties of EJBs, namely the fact that an EJB does not have a *forge facet* (Section 2.1.1). Instead, the framework imposes a requirement that all instances of the class are obtained from an external class, the *home object*, which follows the ABSTRACT FACTORY design pattern.

This property makes it possible to apply an aspect, a service, or a shakein to *all* instances of a certain class. When applying the deploy time weaving technique to non-EJB classes, one may chose to degenerate the forge facet of the target class, as in EJBs, or in the case that this is not possible, make sure that the correct constructors are invoked in the code.

3.3 The ASPECTJ2EE Programming Language

Having described the shakeins construct and deploy-time weaving, we are ready to describe the ASPECTJ2EE language.

The syntax of ASPECTJ2EE is a variant of ASPECTJ. The semantics of ASPECTJ2EE-aspects is based on a (limited) implementation of the shakein concept. Hence, aspects in ASPECTJ2EE, unlike in ASPECTJ, do not have a global effect, and are woven into the application at deployment time, rather than compile time.

When compared to shakeins, the main limitation of ASPECTJ2EE-*aspects* (henceforth just "aspects", unless noted otherwise), is that their application to classes is governed by an exter-

nal *deployment descriptor file*, written in XML. Accordingly, ASPECTJ2EE does not provide a syntax for explicitly applying aspects to classes. Consequently, the integration of aspects into ASPECTJ2EE is not complete. Indeed, ASPECTJ2EE suffers from two XML-JAVA coupling issues: (i) JAVA code using a class whose generation is governed by XML is coupled with this XML code, and (ii) XML file applying an aspect to a ASPECTJ2EE class is coupled with the ASPECTJ2EE names. However, in contrast with JBoss aspects, the detection of errors due to such coupling, i.e., using wrong class names or illegal or empty pointcut expressions, is not at run time, but rather at deployment time.

Comparing the ASPECTJ2EE version of shakeins with the theoretical description of the concept, we find that the benefits (see Section 2.3) are preserved:

1. *Selective application* is available; aspects are applied only to classes specified in the deployment descriptor.

2. *Non-destructive application* is preserved. However, instances are obtained using Home objects (factories) only. Therefore, the programmer, wearing the hat of an *application assembler* [82, Sect. 3.1.2], can dictate which combinations of aspect application are available. For example, it is possible to ensure that all instances of ACCOUNT are subjected to a security aspect.

3. *Explicit and Flexible Ordering* is provided by the XML binding language.

4. *Composition* is supported; special XML syntax can be used for composing two or more aspects. However, composed aspects can have no parameters.

5. *Configuration parameters* are available; the deployment descriptor is used for argument passing.

6. *Repeated application* is fully supported.

7. *Parameter checking* is supported in a very limited manner.

Section 3.3.1 presents the language syntax. The application of aspects through deployment descriptors is the subject of Section 3.3.2. Section 3.3.3 explains how deploy-time weaving can implement the various kinds of join points. Finally, Section 3.3.4 gives a brief overview of the standard aspect library.

3.3.1 Language Syntax

The major difference between ASPECTJ2EE and ASPECTJ is that ASPECTJ2EE supports parameterized aspects. For example, Figure 3.5 shows the definition of a role-based security shakein that accepts two parameters. The first parameter, `secured`, is a pointcut definition specifying which methods are subjected to a security check. The second one, `requiredRole`, is the user role-name that the check requires.

Parameter values must be known at deploy time. Accordingly, there are four kinds of parameters for aspects:

- *Type parameters*, preceded by the keyword **`class`**. The type can be restricted (like type parameters in JAVA generics) using the **`implements`** and **`extends`** keywords. These restrictions are the only parameter checking offered by ASPECTJ2EE.

- *Pointcut parameters*, preceded by the **`pointcut`** keyword.

```
aspect Security[pointcut secured(), String requiredRole] {
   before() : secured() {
      if (!userInRole(requiredRole)) {
         throw new RuntimeException("Security Violation");
      }
   }

   private boolean userInRole(String roleName) {
      // Check if the currently active user has the given role...
   }
}
```

Figure 3.5: The definition of a role-based `Security` aspect in AspectJ2EE

- *String parameters* and *primitive type parameters*, preceded by the type name (`String`, **`int`**, **`boolean`**, etc.). Arrays of these types are also supported.

In contrast with AspectJ, the scope of a specific aspect application in AspectJ2EE is limited to its target class. Therefore, any pointcut that refers to join points in other classes is meaningless. Accordingly, AspectJ2EE does not have a **`call`** join point, since it refers to the calling point, rather than the execution point, of a method. To apply advice to method execution, only an **`execution`** join point can be used. This restriction is a direct result of the shakein semantic model, and it eliminates the confusion associated with the **`call`** join point in relation to inheritance (as discussed in Section 2.1.2).

All other join point kinds are supported, but with the understanding that their scope is limited to the target class; for example, a field-set join point for a **`public`** field will not capture access to the field from outside its defining class. AspectJ2EE also introduces a new kind of join point for handling remote invocation of methods.

Since the application of aspects in AspectJ2EE is explicit, it does not recognize the AspectJ statement **`declare precedence`**. Introductions are also not supported, so neither are **`declare parents`** statements. Similarly, since AspectJ2EE aspects are never instantiated by themselves, the aspect instantiation keywords (**`issingleton`**, **`pertarget`**, **`perthis`**, **`percflow`**, and **`percflowbelow`**) are not part of the AspectJ2EE syntax.

Finally, there is a subtle syntactical difference due to the "individual target class" semantics of AspectJ2EE aspects: The definition of a pointcut should not include the target class name as part of method, field or constructor signatures. Only the member's name, type, access level, list of parameter types, etc. can be specified. For example, the signature matching any **`public void`** method accepting a single `String` argument is written AspectJ as **`public void`** `*.*(String)`. The same signature should be written as **`public void`** `*(String)` in AspectJ2EE. The AspectJ form applies to the methods with this signature in *all* classes, whereas the AspectJ2EE form applies only to such methods in the class to which the containing aspect is applied.

3.3.2 The Deployment Descriptor

In AspectJ, the application of aspects to classes is specified declaratively. Yet the process is not completely transparent: the application assembler must take explicit actions to make sure that the specified aspect application actually takes place. In particular, he must remember to compile each core module with all the aspects that may apply to it. (Or else, an aspect with global applicability may not apply to certain classes if these classes were not compiled with it.)

The order of application of aspects in ASPECTJ is governed by **`declare precedence`** statements; without explicit declarations, the precedence of aspects in ASPECTJ is undefined. Also, ASPECTJ does not provide any means for passing parameters to the application of aspects to modules.

In contrast, the shakein semantics in general, and ASPECTJ2EE in particular, require an explicit specification of each application of an aspect to a class, along with any configuration parameters. This specification could have been done as part of the programming language. But, following the conventions of J2EE, and in the sake of minimizing the syntactical differences between ASPECTJ2EE and ASPECTJ, we chose to place this specification in an *external* XML deployment descriptor.[5]

In fact, we shall see that the XML specification is in essence the abstract syntax tree, which would have been generated from parsing the same specification if written inside the programming langauge. Figure 3.6 gives an example, showing the sequence of application of aspects to ACCOUNT which generated the classes in Figure 3.4. Overall, four aspects are applied to the bean: `Lifecycle`, `Persistence`, `Security`, and `Transactions`. All of these are drawn from the `aspectj2ee.core` aspect library.

The figure shows the XML element describing bean ACCOUNT. (In general, the deployment descriptor contains such entities for each of the beans, along with other information.) We follow the J2EE convention, in that the bean is defined by the `<entity>` XML element (line 1.

Element `<entity>` has several internal elements. The first four, `<ejb-name>`, `<home>`, `<remote>`, and `<ejb-class>` (lines 2–5), specify the JAVA class names that make this bean. These are all part of standard J2EE and will not concern us here.

Following are elements of type `<apply>`, which are an ASPECTJ2EE extension. Each of these specifies an application of an aspect to the bean. Each `<apply>` element must have a property called `<aspect>`, which names the aspect to be applied. For example, line 6 in the figure specifies that the aspect `aspectj2ee.core.Lifecycle` is applied to ACCOUNT.

If the applied aspect is parameterized, then `<parameter>` sub-elements can be used. Each `<parameter>` element has a `name` property, and its body specifies the actual parameter value. Consider for example the `Security` aspect (Figure 3.5). In Figure 3.6 (line 17), we see that the actual value for the `secured` pointcut formal parameter is **`execution`**`(*(..))` (i.e., the execution of any method). Similarly, formal string parameter `requiredRole` was actualized with value *`"User"`* (line 18).

Thus, the third `<apply>` element is tantamount to configuring the aspect with the following pseudo-syntax used in the previous Chapter:

```
Security[execution(*(..)), "User"].        (3.1)
```

Array parameters are specified using any number of `<item>` elements. For example, the `fieldMap` property of the `Persistence` aspect is initialized (lines 11–14) as a two-items array.

In support of *explicit and flexible ordering*, the order of `<apply>` elements specifies the order by which aspects are applied to the bean. Intra-aspect precedence (where several advices from the same aspect apply to a single join point) is handled as in ASPECTJ, i.e., by order of appearance of advices.

We can generalize example (3.1) above to write the entire sequence of application of aspects to the bean, along with their parameters. In total, there are nine such parameters. These, together with the aspect names, would have made the programming language equivalent of the application sequence in Figure 3.6 cumbersome and error-prone. We found that the XML notation is a convenient replacement to developing syntax for dealing with this unwieldiness.

[5] In Java EE 5, deployment descriptors are optional, but still fully supported.

```
<entity id="Account">
   <ejb-name>Account</ejb-name>
   <home>aspectj2ee.demo.AccountHome</home>
   <remote>aspectj2ee.demo.Account</remote>
   <ejb-class>aspectj2ee.demo.AccountBean</ejb-class>
   <apply aspect="aspectj2ee.core.Lifecycle" />
   <apply aspect="aspectj2ee.core.Persistence">
      <parameter name="PKClass">java.lang.String</parameter>
      <parameter name="PKField">serialNumber</parameter>
      <parameter name="table">ACCOUNTS</parameter>
      <parameter name="fieldMap">
         <item>serialNumber:SERIAL</item>
         <item>balance:BALANCE</item>
      </parameter>
   </apply>
   <apply aspect="aspectj2ee.core.Security">
      <parameter name="secured">execution(*(..))</parameter>
      <parameter name="requiredRole">User</parameter>
   </apply>
   <apply aspect="aspectj2ee.core.Transactions">
      <parameter name="reentrant">false</parameter>
      <parameter name="requiresnew">
        execution(deposit(..)) || execution(withdraw(..))
      </parameter>
      <parameter name="required">
        execution(*(..)) && !requiresnew()
      </parameter>
   </apply>
</entity>
```

Figure 3.6: A fragment of an EJB's deployment descriptor specifying the application of aspects to the ACCOUNT bean.

Note that ACCOUNT can be viewed as an entity bean with container-managed persistence (CMP EJB [82, Chap. 10]) simply because it relies on the core persistence aspect, which parallels the standard J2EE persistence service. Should the developer decide to use a different persistence technique, that persistence system would itself be defined as an ASPECTJ2EE aspect, and applied to ACCOUNT in the same manner. This is parallel to bean-managed persistence beans (BMP EJBs) in the sense that the persistence logic is provided by the application programmer, independent of the services offered by the application server. However, it is completely unlike BMP EJBs in that the persistence code would not be tangled with the business logic and scattered across several bean and utility classes. In this respect, ASPECTJ2EE completely dissolves the distinction between BMP and CMP entity beans.

The ACCOUNT bean example does not use every feature supported by ASPECTJ2EE. In particular, it demonstrates selective application, non-destructive application (since the base class remains untouched), explicit and flexible ordering, and configuration parameters. *Repeated application* is not shown, but it can be brought into effect simply by having multiple `<apply>` elements specifying the same aspect, probably with different parameters each time. Support for *composition* is achieved using the `<define-aspect>` element. Figure 3.7 is an example that composes

two applications of the security aspect to define the `MySecurity` aspect (originally defined as fragment (2.1) in Section 2.3.2). Once defined, the `MySecurity` aspect can be used just like any

```
<define-aspect name="MySecurity">
   <apply aspect="aspectj2ee.core.Security">
      <parameter name="secured">P_c</parameter>
      <parameter name="requiredRole">Client</parameter>
   </apply>
   <apply aspect="aspectj2ee.core.Security">
      <parameter name="secured">P_t</parameter>
      <parameter name="requiredRole">Teller</parameter>
   </apply>
</define-aspect>
```

Figure 3.7: Defining the `MySecurity` aspect using aspect composition (P_c and P_t symbolize actual pointcut definitions)

other aspect, including in the composition of other aspects. A limitation of the aspect composition syntax in ASPECTJ2EE, however, is that it does not allow composed aspects to be parameterized; the composition must provide actual value for all parameters used by its constituent aspects.

In some ways, the use of XML configuration files in ASPECTJ2EE is reminiscent of the configuration method used by Spring (Section 2.4.2). However, unlike the Spring configuration files, the deployment descriptors in ASPECTJ2EE (and indeed, in standard J2EE as well) are processed during deployment, before the program is ever executed. Thus, any configuration errors, including type errors in parameter passing, are discovered ahead of execution (as opposed to run-time configuration parsing in Spring).

3.3.3 Implementing Advice by Sub-Classing

ASPECTJ2EE supports each of the join point kinds defined in ASPECTJ, except for **`call`**, since **`call`** advice is applied at the *client* (caller) site and not to the main class. We next describe how advice are woven into the entity bean code in each supported kind of join point.

Execution Join Points.

The **`execution`**(*`methodSignature`*) join point is defined when a method is invoked and control transfers to the target method. ASPECTJ2EE captures **`execution`** join points by generating advised methods in the advised class, overriding the inherited methods that match the execution join point. Consider for example the advice in Figure 3.8(a), whose pointcut refers to the execution of the `deposit()` method. This is a **`before`**`()` advice which prepends a printout line to matched join points. When applied to ACCOUNT, only one join point, the execution of `deposit()`, will match the specified pointcut. Hence, in the advised class, the `deposit()` method will be overridden, and the advice code will be inserted prior to invoking the original code. The resulting implementation of `deposit()` in the advised class appears in Figure 3.8(b).

Recall that only instances of the terminal advised class exist in the system, so every call to the advised method (`deposit()` in this example) would be intercepted by means of regular polymorphism. Overriding and refinement can be used to implement **`before`**`()`, **`after`**`()` (including **`after`**`()` **`returning`** and **`after`**`()` **`throwing`**), and **`around`**`()` advice. With **`around`**`()` advice, the **`proceed`** keyword indicates the location of the call to the inherited implementation.

The example in Figure 3.9 demonstrates the support for **`after`**`()` **`throwing`** advice. The advice, listed in part (a) of the figure, would generate a printout if the `withdraw()` method

```
before(float amount):
         execution(deposit(float)) && args(amount) {
   System.out.println("Depositing " + amount);
}
```

(a) Sample **`execution`** advice for the `deposit()` method

```
public void deposit(float amount) {
   System.out.println("Depositing " + amount);
   super.deposit(amount);
}
```

(b) The resulting, advised version of `deposit()`

Figure 3.8: An example for weaving an **`execution`** join point

resulted in an `InsufficientFundsException`. The exception itself is re-thrown, i.e., the advice does not swallow it. The resulting advised method appears in Figure 3.9(b). It shows how **`after()throwing`** advice are implemented by encapsulating the original implementation in a **`try`**/**`catch`** block.

```
after() throwing (InsufficientFundsException ex)
  throws InsufficientFundsException: execution(withdraw(..)) {
    System.out.println("Withdrawal failed: " + ex.getMessage());
    throw ex;
}
```

(a) Sample **`after throwing`** advice for an **`execution`** join point

```
public void withdraw(float amount)
         throws InsufficientFundsException {
  try {
    super.withdraw(amount);
  }
  catch (InsufficientFundsException ex) {
    System.out.println("Withdrawal failed: " + ex.getMessage());
    throw ex;
  }
}
```

(b) The resulting, advised version of `withdraw()`

Figure 3.9: An example for weaving **`after throwing`** advice in an **`execution`** join point

An **`execution`** join point may refer to **`private`** methods. Since such methods cannot be overridden in subclasses, the ASPECTJ2EE weaver generates a new, advised version of the method—and then overrides any method that *invokes* the private method, so that the callers will use the newly-generated version of the private callee rather than the original. The overriding version of the callers includes a complete re-implementation of each caller's code, rather than using refinement, so that only the new version of the private callee will be used. The only exception is where a private method is invoked by a constructor, which cannot be replaced by an overriding version. ASPECTJ2EE will issue a warning in such cases.

This technique is used not only with **execution** join points, but whenever an advice applies to code inside a **private** method (e.g., when a field access join point is matched by code inside one).

A similar problem occurs with **final** and **static** methods. However, such methods are disallowed by the J2EE specification and may not be included in EJB classes.

Constructor Execution Join Points.

The constructor execution join point in ASPECTJ is defined using the same keyword as regular method execution. The difference lies in the method signature, which uses the keyword **new** to indicate the class's constructor. For example, the pointcut **execution(new(..))** would match the execution of any constructor in the target class.

Unlike regular methods, constructors are limited with regard to the location in the code where the inherited implementation (**super()**) must be invoked. The invocation must be the first statement of the constructor, and in particular it must occur before any field access or virtual method invocation. Hence, join points that refer to constructor signatures can be advised, but any code that executes before the inherited constructor (**before()** advice, or parts of **around()** advice that appear prior to the invocation of **proceed()**) is invalid.

An **around()** advice for constructor execution that does not contain an invocation of **proceed()** would be the equivalent of a JAVA constructor that does not invoke **super()** (the inherited constructor). This is tantamount to having an implicit call to **super()**, and is valid only if the advised class contains a constructor that does not take any arguments.

Field Read and Write Access Join Points.

Field access join points match references to and assignments of fields. ASPECTJ2EE presents no limitations on advice that can be applied to these join points. However, if a field is visible *outside* of the class (e.g., a **public** field), then any *external* access will bypass the advice. It is therefore recommended that field access will be restricted to **private** fields and EJB *attributes* only. Recall that attributes are not declared as fields; rather, they are indicated by the programmer using **abstract** getter and setter methods. These methods are then implemented in the concrete bean class (in J2EE) or in the root advised class (in ASPECTJ2EE).

If no advice is provided for a given attribute's read or write access, the respective method implementation in the root advised class would simply read or update the class field. The field itself is also defined in the root advised class. However, an attribute can be advised using **before()**, **around()** and **after()** advice, which would affect the way the getter and setter method are implemented.

If an advice is applied to a field (which is not an attribute), all references to this field by method in the class itself are advised by generating overriding versions of these methods. However, since a **private** field is not visible even to subclasses, this might require generating a new version of the field, which *hides* [117, Sect. 8.3.3] the original declaration. In such cases, any method that accesses the field must be regenerated, even where the advice does not cause any code alteration, so that the overriding version will access the new field. In addition, the advised class's constructor initializes the new field by copying the value of the hidden one (which was initialized by the inherited constructor).

Exception Handler Join Points.

The **handler** join point can be used to introduce advice into **catch** blocks for specific exception types. Since the **catch** block per-se cannot be overridden, advising such a join point results in

a new, overriding version of the entire method containing the advised **`catch`** block. Most of the code remains unchanged from the original, but the code inside the **`catch`** block is altered in accordance with the advice.

Remote Call Join Points.

The **`remotecall`** join point designator is a new keyword introduced in ASPECTJ2EE. Semantically, it is similar to ASPECTJ's **`call`** join point designator, defining a join point at a method invocation site. However, it only applies to remote calls to various methods; local calls are unaffected.

Remote call join points are unique, in that their applied advice does not appear in the advised sub-class. Rather, they are implemented by affecting the stub generated at deploy time for use by EJB clients (such as `_Account_Stub` in Figure 3.4). For example, the **`around()`** advice from Figure 3.10(a) adds printout code both before and after the remote invocation of `Account.deposit()`. The generated stub class would include a `deposit()` method like the one shown in part (b) of that figure. Since the advised code appears in the stub, rather than in a server-side class, the output in this example will be generated by the client program.

```
around(): remotecall(deposit(..)) {
   System.out.println("About to perform transaction.");
   proceed();
   System.out.println("Transaction completed.");
}
```

(a) Sample advice for a method `deposit`'s **`remotecall`** join point

```
public void deposit(float arg0) {
   System.out.println("About to perform transaction.");
   // ... normal RMI/IIOP method invocation code ...
   System.out.println("Transaction completed.");
}
```

(b) The resulting, advised version of `deposit()`

Figure 3.10: (a) Sample advice for a method's **`remotecall`** join point, and (b) the resulting `deposit()` method generated in the RMI stub class.

Remote call join points can only refer to methods that are defined in the bean's remote interface. Advice using **`remotecall`** can be used to localize tier-cutting concerns, as detailed in Section 3.4.

Control-Flow Based Pointcuts.

ASPECTJ includes two special keywords, **`cflow`** and **`cflowbelow`**, for specifying control-flow based limitations on pointcuts. Such limitations are used, for example, to prevent recursive application of advice [152]. Both keywords are supported by ASPECTJ2EE.

The manner in which control-flow limitations are enforced relies on the fact that deployment can be done in a completely platform-specific manner, since at deploy time, the exact target platform (JVM implementation) is known. Different JVMs use different schemes for storing a stack snapshot in instances of the `java.lang.Throwable` class [53] (this information is used, for example, by the method `java.lang.Exception.printStackTrace()`). Such a stack

snapshot (obtained via an instance of `Throwable`, or any other JVM-specific means) can be examined in order to test for **`cflow`**/**`cflowbelow`** conditions at runtime.

An alternative implementation scheme relies on `ThreadLocal` objects [112]. The advice application would result in a `ThreadLocal` flag that will be turned on or off as various methods are entered or exited. At the target join point, the flag's value will be examined to determine if the **`cflowbelow`** condition holds, and the advice should be executed. The one-instance-per-thread nature of `ThreadLocal` objects ensures that this flag-based scheme will function properly in multi-threaded applications.

3.3.4 The Core Aspects Library

ASPECTJ2EE's definition includes a standard library of core aspects. Four of these aspects were used in the ACCOUNT example, as shown in Figure 3.4. Here is a brief overview of these four, and their effect on the advised classes:

1. The `aspectj2ee.core.Lifecycle` aspect (used to generated the root advised class) provides a default implementation to the J2EE lifecycle methods. The implementations of `setEntityContext()`, `unsetEntityContext`, and `getEntityContext()` maintain the entity context object; all other methods have an empty implementation. These easily-available common defaults make the development of EJBs somewhat easier (compared to standard J2EE development); the user-provided `AccountBean` class is now shorter, and contains strictly business logic methods. [6]

2. The `aspectj2ee.core.Persistence` aspect provides a CMP-like persistence service. The attribute-to-database mapping properties are detailed in the parameters passed to this aspect in the deployment descriptor. This aspect advises some of the lifecycle methods, as well as the attribute setters (for maintaining a "dirty" flag), hence these methods are all overridden in the advised class.

3. The `aspectj2ee.core.Security` aspect can be used to limit the access to various methods based on user authentication. This is a generic security solution, on par with the standard J2EE security service. More detailed security decisions, such as role-based variations on method behavior, can be defined using project-specific aspects without tangling security-related code with the functional concern code.

4. Finally, the `aspectj2ee.core.Transactions` aspect is used to provide transaction management capabilities to all business-logic methods. The parameters passed to it dictate what transactional behavior will be applied to each method. Transactional behaviors supported by the J2EE platform include methods that must execute within a transaction context, and will create a new transaction if none exists; methods that must execute within an existing transaction context; methods that are neutral to the existence of a transaction context; and methods that will fail to run within a transaction context. The list of methods that belong to each group is specified with a pointcut parameter passed to this aspect.

3.4 Innovative Uses for AOP in Multi-Tier Applications

The use of aspects in multi-tier enterprise applications can reduce the amount of cross-cutting concerns and tangled code. As discussed in Section 3.1, the core J2EE aspects were shown to

[6] The fact that the fields used to implement the attributes, and the concrete getter and setter method for these attributes, appear in `AdvAccount_Lifecycle` (in Figure 3.4) stems from the fact that this is the root advised class, and is not related to the `Lifecycle` aspect per se.

be highly effective to this end, and the ability to define additional aspects (as well as alternative implementations to existing ones) increases this effectiveness and enables better program modularization.

But ASPECTJ2EE also allows developers to confront a different kind of cross-cutting non-functional concerns: aspects of the software that are implemented in part on the client and in part on the server. Here, the cross-cutting is extremely acute as the concern is implemented not just across several classes and modules, but literally across programs. We call these *tier-cutting concerns*. In the context of ASPECTJ2EE, tier-cutting concerns are applied to the business methods of EJBs.

The notion of remote pointcuts was independently discovered by Nishizawa, Chiba, and Tatsubori [180].

The remainder of this section shows that several key tier-cutting concerns can be represented as single aspect by using the **`remotecall`** join point designator. In each of these examples, the client code is unaffected; it is the RMI stub, which acts as a proxy for the remote object, which is being modified.

3.4.1 Client-Side Checking of Preconditions

Method preconditions [169] are commonly presented as a natural candidate for non-functional concerns being expressed cleanly and neatly in aspects. This allows preconditions to be specified without littering the core program, and further allows precondition testing to be easily disabled.

Preconditions should normally be checked at the method execution point, i.e., in the case of multi-tier applications, on the server. However, a precondition defines a contract that binds whomever invokes the method. Hence, by definition, precondition violations can be detected and flagged at the invocation point, i.e., on the client. In a normal program, this matters very little; but in a multi-tier application, trapping failed preconditions on the client can prevent the round-trip of a remote method invocation, which incurs a heavy overhead (including communications, parameter marshaling and un-marshaling, etc.).

Figure 3.11 presents a simple precondition that can be applied to the ACCOUNT EJB: neither `withdraw()` nor `deposit()` are ever supposed to be called with a non-positive amount as a parameter. If such an occurrence is detected, a `PreconditionFailedException` is thrown. Using two named pointcut definitions, the test is applied both at the client and at the server.

In addition to providing a degree of safety, such aspects decrease the server load by blocking futile invocation attempts. In a trusted computing environment, if the preconditioned methods are invoked only by clients (and never by other server-side methods), the server load can be further reduced by completely disabling server-side tests.

When using aspects to implement preconditions, always bear in mind that preconditions test for logically flawed states, rather than states that are unacceptable from a business process point of view. Thus, preventing the withdrawal of excessive amounts should be part of `withdraw()`'s implementation rather than a precondition.

3.4.2 Symmetrical Data Processing

By adding code both at the sending and receiving ends of remotely-invoked methods, we are able to create what can be viewed as an additional layer in the communication stack. For example, we can add encryption at the stub and decryption at the remote tie, for increased security; or we can apply a compression scheme (compressing information at the sender, decompressing it at the receiver) to reduce the communications overhead; and so forth.

Consider an EJB representing a university course, with a method `register()` that accepts a `Vector` of student names (`String`s) to be registered to that course. The aspect in Figure 3.12

```
public aspect EnsurePositiveAmounts {
   pointcut clientSide(float amount):
      (remotecall(public void deposit(float)) ||
       remotecall(public void withdraw(float))) && args(amount);

   pointcut serverSide(float amount):
      (execution(public void deposit(float)) ||
       execution(public void withdraw(float))) && args(amount);

   before(float amount):
               clientSide(amount) || serverSide(amount) {
      if (amount <= 0.0)
         throw new PreconditionFailedException(
                           "Non-positive amount: "+amount);
   }
}
```

Figure 3.11: An aspect that can be used to apply precondition testing (both client- and server-side) to the ACCOUNT bean

shows how the remote invocation of this method can be made more effective by applying compression. Assume that the class `CompressedVector` represents a `Vector` in a compressed (space-efficient) manner. Applying this aspect to the COURSE EJB would result in a new method, `registerCompressed()`, added to the advised class. Unlike most non-public methods, this one would be represented in the class's RMI stub, since it is invoked by code that is included in the stub itself (that code would reside in the advised stub for the `register()` method).

```
public aspect CompressRegistrationList {
   around(Vector v): remotecall(public void register(Vector))
                                                  && args(v) {
      CompressedVector cv = new CompressedVector(v);
      registerCompressed(cv);
   }

   private void registerCompressed(CompressedVector cv) {
      Vector v = cv.decompress();
      register(v);
   }
}
```

Figure 3.12: An aspect for sending a compressed version of an argument over the communications line; can be applied to the bean COURSE which contains a `register` method

Less specific versions of the compression aspect (and of similar aspects, such as encryption) can be developed, relying on JAVA's reflection mechanism to collect information about the method arguments and then creating a compressed serialized version of the actual parameters. Additionally, compression and encryption can be applied not only for arguments, but also for return values. In this case, the aspect should use **`after() returning`** advice for both the **`remotecall`** and **`execution`** join points. Advice for **`after() throwing`** can be used for processing excep-

tions (which are often information-laden, due to the embedded call stack, and would hence benefit greatly from compression).

3.4.3 Memoization

Memoization (the practice of caching method results) is another classic use for aspects. When applied to a multi-tier application, this should be done with care, since in many cases the client tier has no way to know when the cached data becomes stale and should be replaced. Still, it is often both possible and practical, and using ASPECTJ2EE it can be done without changing any part of the client program.

For example, consider a session EJB that reports international currency exchange rates. These rates are changed on a daily basis; for the sake of simplicity, assume that they are changed every midnight. The aspect presented in Figure 3.13 can be used to enable client-side caching of rates.

3.5 Summary

We presented ASPECTJ2EE as an important application and prime motivation for actual use of the shakeins construct. ASPECTJ2EE is an aspect-oriented programming language, similar to ASPECTJ, whose aspects have a shakein semantics. The language shows that the shakeins semantics integrates well with the current architecture of J2EE servers. It also makes it possible to think of existing services as aspects, while unifying the deployment process of J2EE with aspect weaving as in AOP. Also, the langauge shows that the shakeins semantics allows existing services to be configured, and even applied multiple times. Such benefits are not possible with plain aspects.

By using deploy-time weaving, ASPECTJ2EE allows the programmer's code to be advised without being tampered with. Programmers can define methods that will provide business functionality while being oblivious to the various services (transaction management, security, etc.) applied to these methods. (It is the *code*, not the programmers, that is oblivious to the non-functional concerns—an important distinction, no doubt [97].) With the exception of the **`call`**, we showed that all join point kinds can be implemented using deploy-time weaving.

To the existing repertoire of join points ASPECTJ2EE adds a new join point kind, **`remotecall`**. This join point makes it possible to add to the familiar services provided by EJB containers. ASPECTJ2EE aspects using **`remotecall`** can be used to unscatter and untangle tier-cutting concerns, which in many cases can improve an application server's performance. We discussed in particular interesting such services, including client side checking of pre-conditions, symmetrical data processing, and memoization.

In using the shakeins semantics, aspects in ASPECTJ2EE are less general, and have a more defined target, than their ASPECTJ counterparts. Also, even though the same aspect can be applied (possibly with different parameters) to several EJBs, each such application can only affect its specific EJB target. Therefore, we expect ASPECTJ2EE aspects should be more understandable, and the woven programs more maintainable.

We believe that ASPECTJ2EE opens a new world of possibilities to developers of EJB-based applications, allowing them to extend, enhance and replace the standard services provided by EJB containers with services of their own. EJB services can be distributed and used across several projects; libraries of services can be defined and reused.

ASPECTJ2EE does not encompass the shakeins semantics in full. In particular, aspect application is external to the language and is specified by an XML deployment descriptor file. The file format is such that shakeins can be applied to EJBs only, and that un-shaked versions of such a bean are not available to clients. To ease interoperability with existing J2EE applications, the XML file format is an extension of the standard J2EE deployment descriptor format.

```
public aspect CacheExchangeRates {
   private static class CacheData {
      int year;
      int dayOfYear;
      float value;
   }

   private Hashtable<String, CacheData> cache
                     = new Hashtable<String, CacheData>();

   pointcut clientSide(String currencyName):
      remotecall(public float getExchangeRate(String))
      && args(currencyName);

   around(String currencyName): clientSide(currencyName) {
      Calendar now = Calendar.getInstance();
      int currentYear = now.get(Calendar.YEAR);
      int currentDayOfYear = now.get(Calendar.DAY_OF_YEAR);

      // First, try and find the value in the cache
      CacheData cacheData = cache.get(currencyName);
      if (cacheData != null && currentYear = cacheData.year &&
         currentDayOfYear == cacheData.dayOfYear)
            return cacheData.value; // Value is valid; no remote invocation

      // Value is not in cache: obtain normally
      float result = proceed(currencyName); // remote call

      // Cache the value for future reference
      cacheData = new CacheData();
      cacheData.year = currentYear;
      cacheData.dayOfYear = currentDayOfYear;
      cacheData.value = result;
      cache.put(currencyName, cacheData);
   }
}
```

Figure 3.13: A memoization aspect for caching results from a currency exchange-rates bean

The J2EE framework, and the ASPECTJ2EE language presented here, both use the ABSTRACT FACTORY design pattern to control object instantiation. However, there neither contains a mechanism that prevents developers from bypassing the home object (i.e., the factory class) and obtaining "raw", unadvised versions of the beans. This is possible because ASPECTJ2EE aspects, much like J2EE services, operate within the standard object model, and do not alter the base classes onto which the aspects/services are applied. To overcome this limitation, we suggest the factories mechanism, discussed in Chapter 5.

But first, the following chapter presents JTL. This embedded language can serve as a superior alternative to ASPECTJ's (and ASPECTJ2EE's) pointcut specification syntax—and much more.

Chapter 4

JTL

Hast thou found any likeness for thy vision?

— A.C. Swinburne, *Ave atque Vale*

JTL (the *Java Tools Language*, pronounced *Gee-tel*) is a declarative language, belonging in the logic-programming paradigm, designed for the task of selecting JAVA program elements. Two primary applications were in mind at the time when the language was first conceived:

(a) *Join-point selection for aspect-oriented programming*, where JTL can serve as a powerful substitute of ASPECTJ's pointcut syntax.

The pointcut expressions of ASPECTJ select the points in the code onto which an aspect is to be applied. The limited expressive power of the pointcut specification language of ASPECTJ (as used in the original definition of ASPECTJ2EE in Chapter 3) has been noted several times in the literature [126, 187]. JTL was designed to address these limitations, providing greater flexibility to the shakeins mechanism—although it can also be integrated into other AOP solutions as well.

(b) *Expressing the conditions making up concepts* for use in generic programming, including multi-type concepts.

Concepts [106, 124, 210] are a key issue in generic programming, since they make explicit the test of whether a given set of classes are legible as parameters for a given generic construct. As a concept specification language, JTL can be used with shakeins, allowing each shakein to explicitly limit the set of classes to which it can be applied.

JTL can also be used for other language extension tasks. The designers of frontier programming paradigms and constructs often choose, especially when static typing is an issue, to test-bed, experiment, and even fully implement the new idea by an extension to the JAVA programming language. A prime component in the interaction of such an extension with the language core is the mechanism for selecting program elements to which the extension applies. Examples that can benefit from a tool like JTL include JAM [8], Chai [205], OpenJava [220], the host of type systems supported by pluggable type systems [9], and many more.

JTL's design took a special effort to adapt the logic to the task at hand, and to provide a *Query By Example* [231] flavor to predicates, thereby minimizing the *abstraction gap* between the query language and the queried domain. A number of techniques were used to this end: first, many JAVA keywords were adopted by JTL, e.g., **`static`** program elements are matched by a predicate named **`static`**. Second, an effort was made to eliminate most of syntactical baggage of PROLOG [83] such as commas, parenthesis, multiple rules, etc. Third, to simplify the evaluation

semantics, the underlying computation is DATALOG [50]-based, i.e., there is no unification, no infamous cuts of any color, etc. Most importantly, JTL offers higher level abstraction mechanisms on top of the core logic programming—just as the looping constructs of high-level languages are built on top of machine-level conditional branches.

As JTL took shape and grew older it became clear that it can be used not only for language extension tasks, but also for other software engineering tasks, primarily as a tool to assist programmers understand the code they must modify. This particular problem of program understanding, even if it is far from being entirely solved by JTL, is dear to our hearts: First, software development activities in the industry include (and probably more and more so) the integration of new functionalities in existing code. Second, maintenance remains a major development cost factor.

JTL's focus is on the modules in which the code is organized: packages, classes, methods, variables and parameters (including their names), types, accessibility level and other attributes. JTL can also inspect the interrelations of these modules, including questions such as which classes exist in a given unit, which methods does a given method invoke, etc.

Going beyond JTL's core, two extensions were developed. One extension, developed by Itay Maman, allows JTL to inspect the imperative parts of the code by means of dataflow analysis [67]. Such analysis can be used, for example, in micro-pattern detection [68]. Another extension, presented in Section 4.6, provides JTL with the ability to generate output text based on matched program elements. This extends the use of JTL not only for search operations, but also for *search-and-replace* operations, and beyond that into a full-fledged program transformation tool.

4.0.1 Two Introductory Examples

JTL's syntax is terse and intuitive; just as in AWK [1], one-line programs are abundant, and are readily wrapped within a single string. In many cases, the JTL *pattern* for matching a JAVA program element looks exactly like the program element itself. For example, the JTL *predicate*[1]

```
public abstract void ()
```

matches all methods (of a given class) which are abstract, publicly accessible, return **`void`** and take no parameters. Thus, in a sense, JTL mimics the Query By Example idea.

As in the logic paradigm, a JTL *program* is a set of predicate definitions, one of which is marked as the program *goal*.

Even patterns which transcend the plain JAVA syntax should be readily understandable; for example,

```
abstract class {
    [long | int] field;
    no abstract method;
}
```

matches abstract classes in which there is a field whose type is either **`long`** or **`int`** and no abstract methods. The first line in the curly brackets is an *existential quantifier* ranging over all class members. The second line in the brackets is a *negation* of an existential quantifier, i.e., a *universal quantifier* in disguise, applied to this range.

4.0.2 The Underlying Model

Underlying JTL is a *conceptual* representation of a program in a simply-typed relational database. The JTL user can think of the interrogated JAVA program as a bunch of program elements stored in such a database.

[1] The terms "predicate" and "pattern" are used almost interchangeably; "pattern" usually refers to a unary predicate.

JTL is declarative, sporting the simple and terse syntax and semantics of logic programming for making database queries. JTL augments these with a number of enhancements that make it even more concise and readable. Predicates are the basic programming unit. The language features a set of native predicates (whose implementation is external to the language), with a library of pre-defined predicates built on top of the native ones. Many of the native and pre-defined predicates are conveniently named after JAVA keywords.

Thus, the scheme of this database is defined by the set of native JTL predicates. Standard library predicates and user-defined predicates, defined on top of the natives, can be thought of as database *views* or *pre-defined queries*.

Interestingly, JAVA (and many other software systems) are best modeled as *infinite* databases. The reason is that in JAVA and in almost all programming languages, one cannot hope to obtain all user code which uses a certain program element stored in a software library. Similarly, the list of classes that inherit from a given class is unbounded. This is quite the opposite of traditional database systems, which rely on a finite, closed-world model.

The JTL processor analyzes the predicates presented to it, determining whether they are open-ended, i.e., the size of the result they return is unbounded. In practice, only a finite approximation of the infinite database is stored; an open-ended predicate can be thought as a query whose size increases indefinitely with that of the approximation.

Note that this conceptual representation does not dictate any concrete representation for the JTL implementation. JTL is applicable to several formats of program representation, ranging from program source code, going through AST representations, JAVA reflection objects, BCEL [13] library entities, to strings representing the names of program elements. In fact, JTL's JAVA API is characterized by input- and output- data representation flexibility, in that JTL calls can accept and return data in a number of supported formats.

We stress that JTL can be implemented in principle on top of any source code parser, including the JAVA compiler itself.

Two central concerns in the language design were *scalability* and the *simplification* of the idiosyncracies of logic programming.

We found, in accordance with the experience reported by Hajiyev, Verbaere and More with their *CodeQuest* system [128], that the underlying relational model, together with combinations of bottom-up and top-down evaluation strategies that DATALOG makes possible, makes a major contribution to scalability.

For the sake of elegance and brevity of expression, JTL features specific constructs for set manipulation, quantification and other means that eliminate much of the need for loops (recursive calls in the logic programming world). As a result, unlike DATALOG and PROLOG queries, JTL predicates are defined by a single rule, written in a handful of lines, and often in a single line.

Underlying JTL's syntax and semantics is first order predicate logic with no function symbols and augmented with transitive closures, denoted FOPL*. The first order logic represented by JTL is restricted to finite structures (assuming a given database approximation). An inherent difficulty with FOPL** is that it allows one to make cyclic and senseless statements such as "*predicate p holds if and and only if the negation of p holds.*". The language pre-processor therefore restricts queries to to DATALOG with stratified negation [223]. This allows us to be enjoy the theoretical advantages of the formalism, including *modular specification*, *polynomial time complexity*, and a wealth of *query optimization techniques* [119]. Indeed, the JTL compiler (under development by Itay Maman and others) will generate DATALOG output for an industrial-strength DATALOG engine.

An interesting contribution of the work in JTL is in demonstrating that a simple query-by-example like syntax is possible for many tasks of querying OO programs, and in showing that this syntax stands on a solid theoretical ground. It may be possible to put together a JAVA-like

syntax for JAVA queries in an ad hoc fashion. The challenge we took upon ourselves was the combination of the sound underlying computational model and the query-by-example front end. Also, in using a DATALOG-based model (and in contrast with PROLOG as some other recent tools do) we achieve a termination guarantee, and the wealth of theory on database query optimization for concrete scalable implementation.

SQL, and more generally, the relational model was sometimes used for software query [208]. However, as Consens, Mendelzon and Ryman [69] observed, program analysis frequently requires transitive closure. This is the reason that JTL allows recursion, and is similar in its computational expressive power to Consents et al.'s *Graphing* system.

Chapter outline. Section 4.1 is a brief language tutorial, which shows how JTL can be used to inspect the non-imperative aspects of JAVA code, i.e., everything but the method bodies. An explanation of the semantics is then presented in Section 4.2.

The key uses of JTL in the domain of aspect-oriented programming are discussed in Section 4.3, followed by an overview of additional, unrelated uses (Section 4.4). Related work about query languages is discussed in Section 4.5.

Section 4.6 presents the language-transformation extensions, and Section 4.7 presents uses for this extension, in aspect-oriented development and elsewhere. Related work on program transformation is discussed in Section 4.8. Section 4.9 concludes.

Acknowledgement. The JTL language was co-developed with Itay Maman in the Technion faculty of Computer Science. This includes the language definition and its theoretical foundation (Sections 4.1 and 4.2), the survey of alternative query languages (Section 4.5), and the discussion of additional uses for the language (Section 4.4). All other sections of this chapter represent original research by the current author.

4.1 The JTL Language

This section gives a brief tutorial of JTL, assuming some basic familiarity with logic programming. The main issues to note here are the language *syntax*, in which a JAVA program element is matched by a JTL pattern which is very similar in structure to that element (see Section 4.1.1), and the *extensions* to the logic paradigm, such as argument list patterns, transitive closure standard predicates, and quantifiers which make it possible to achieve many programming tasks without recursion.

The two most important data types, what we call *kinds*, of JTL are (i) **`MEMBER`**, which represents all sorts of class and interface members, including function members, data members, constructors, initializers and static initializers; and (ii) **`TYPE`**, which stands for JAVA **`class`**es, **`interface`**s, and **`enum`**s, as well as JAVA's primitive types such as **`int`**. Additional types include **`PACKAGE`** and **`STRING`**. Compound kinds include **`MEMBERS`** (a list of elements of kind **`MEMBER`**), **`TYPES`** (a list of **`TYPE`** elements), etc.

A JTL program is a set of definitions of named logical *predicates*. Execution begins by selecting a predicate to execute as a *goal*.

As in PROLOG, predicate names start with a lower-case letter, while *variables* and parameters names start with a capital letter. Identifiers may contain letters, digits, or an underscore. Additionally, the final characters of an identifier name may be "`+`" (plus), "`*`" (asterisk), or "`'`" (single quote).

4.1.1 Simple Patterns

Many JAVA keywords are native patterns in JTL, carrying essentially the same semantics. For example, the keyword **`int`** is also a JTL pattern **`int`**, which matches either fields of type **`int`** or methods whose return type is **`int`**. The pattern **`public`** matches all **`public`** program elements, including public class members (e.g., fields) and public classes. Henceforth, our examples shall use these keywords freely.

Not all JTL natives are JAVA keywords. A simple example is `anonymous`, defined on **`TYPE`**, which matches anonymous classes.

Some patterns (like **`abstract`**) are overloaded, since they are applicable both to types and members. Others are monomorphic, e.g., **`class`** is applicable only to **`TYPE`**.

Another example is pattern `type`, defined only on **`TYPE`**, which matches all values of **`TYPE`**. This, and the similar pattern `member` (defined on **`MEMBER`**) can be used to break overloading ambiguity.

JTL has two kinds of predicates: *native* and *compound*. Native predicates are predicates whose implementation is external to the language. In other words, in order to evaluate native predicates, the JTL processor must use an external software library accessing the code. Native patterns are therefore declared (in a pre-loaded configuration file) but not defined by JTL.

In contrast, compound patterns are defined by a JTL expression using logical operators. The pattern

```
public, int                                                    (4.1)
```

matches all **`public`** fields of type **`int`** and all **`public`** methods whose return type is **`int`**. As in PROLOG, conjunction is denoted by a comma. In JTL however, the comma is optional; patterns separated by whitespace are conjuncted. Thus, (4.1) can also be written as **`public int`**. As a matter of style, the JTL code presented henceforth denotes conjunction primarily by whitespace; commas are used mainly for readability—breaking long conjugation sequences into subsequences of related predicates;

Disjunction is denoted by a vertical bar, while an exclamation mark stands for logical negation. Thus, the pattern

```
public | protected | private
```

matches JAVA program elements whose visibility is not default, whereas `!`**`public`** matches non-**`public`** elements.

Logical operators obey the usual precedence rules, i.e., negation has the highest priority and disjunction has the lowest. Square parenthesis may be used to override precedence, as in

```
!private [byte|short|int|long]
```

which matches non-**`private`**, integral-typed fields and methods.

A *pattern definition* names a pattern. After making the following two definitions,

```
integral := byte | short | int | long;
enumerable := boolean | char;
```

the newly defined patterns, `integral` and `enumerable`, can be used anywhere a native pattern can be, as in e.g.,

```
discrete := integral | enumerable
```

Beyond the natives, JTL has a rich set of pre-defined *standard* patterns, including patterns such as `integral`, `enumerable`, `discrete` (as defined above), `method`, `constructor` (both with the obvious semantics), the predicate

```
extendable := !final type
```

(matching classes and interfaces which may have heirs), predicate

```
overridable := !final !static method
```

(methods which may be overridden), and many more.

4.1.2 Signature Patterns

Signature patterns pertain to (a) the name of classes or members, (b) the type of members, (c) argument list, (d) declared thrown exceptions, and (e) annotations (meta-data).

Name Patterns

A *name pattern* is a regular expression preceded by a single quote, or a previously-declared name without the quote. Standard JAVA regular expressions (as defined by `java.util.regex.Pattern`) are used, except that the wildcard character is denoted by a question mark rather than a dot, since dots play an important role in JAVA program element names.

Name literals and regular expressions are quoted with single quotes; the closing quote can be omitted if there is no ambiguity.

For example, `void 'set[A-Z]?*' method` matches any **void** method whose name starts with "`set`" followed by an upper-case letter.

If the name pattern does not contain any regular expression operators, as in

```
toString_p := 'toString method;
```
(4.2)

then the pattern can be made clearer by using a **name** statement to declare `toString` as a member name and get rid of the quote. Thus, an alternative definition of (4.2) is

```
name toString;
toString_p := toString method;
```
(4.3)

(In truth, the above is redundant, since an implicit **name** statement pre-declares all methods of the JAVA root class `java.lang.Object`.)

Type Patterns

Type patterns make it possible to specify the JAVA type of a non-primitive class member. A type pattern is a regular expression preceded by a slash, e.g., pattern `/java.util.?*/ method` matches all methods with a return type from the `java.util` package or its sub-packages. The closing slash is optional.

The distinction between type patterns and name patterns only makes sense for members. In matching types, there is no such distinction, and both kinds of patterns or literals can be used. (There is a difference, however, in the resulting string baggage, discussed in Section 4.6.)

The slash is not necessary for type names which were previously declared as such by a **typename** declaration. For example,

```
typename java.io.PrintStream;
printstream_field := PrintStream field;
```
(4.4)

matches any field whose type is `java.io.PrintStream`. The **`typename`** statement in (4.4) declares `java.io.Serializable` as a name of a type, similarly to **`name`** statements for member names.

All the types (including classes, interfaces and enumerations) declared in the `java.lang` package are pre-declared as type names, including `Object`, `String`, `Comparable`, and the wrapper classes (`Integer`, `Byte`, `Void`, etc.).

Here is a redefinition of `toString_p` pattern (4.3), which ensures that the matched method returns a `String`:

```
toString_p := String toString method;
```
(4.5)

Argument List Patterns

JTL provides special constructs which all but eliminate recursion. An important example is *argument list patterns*, used for matching against elements of the list of arguments to a method. (Internally, such lists are stored in a linked list of elements of kind **`TYPE`**, using standard PROLOG-like `head` and `tail` relations.)

The most simple argument list is the empty list, which matches methods and constructors that accept no arguments. Here is a rewrite of (4.5) using such a list:

```
toString_p := String toString();
```
(4.6)

Note that (4.6) does not match fields, which have no argument list, nor constructors, which have no return type.

An asterisk ("`*`") in an argument list pattern matches a sequence of zero or more types. Thus, the standard pattern

```
invocable := (*);
```

matches members which may take any number of arguments, i.e., constructors and methods, but not fields, initializers, or static initializers. An underscore ("`_`") is a single-type wildcard, and can be used in either the argument list or in the return type. Hence,

```
public _ (_, String, *);
```
(4.7)

matches any public method that accepts a `String` as its second argument, and returns any type. (Again, constructors fail to match (4.7), since they have no return type.)

Argument list patterns are in fact an iteration construct, otherwise known as list queries. Section 4.1.6 explains the full semantics and syntax of list queries, which are not limited to argument list patterns.

Other Signature Patterns

There are patterns for matching the **`throws`** clause of the signature, e.g.,

```
io_method := method throws /java.io.IOException;
```

There are also patterns which test for the existence or absence of specific annotations in a class, a field or a method, and for annotation values. For example, the following pattern will match all methods that have the `@Override` annotation:

```
@Override method
```

These are not discussed in further detail in this brief tutorial.

4.1.3 Variables

It is often useful to examine the program element which is matched by a pattern. JTL employs variable binding, similar to that of PROLOG, for this purpose. For example, by using variable `X` twice, the following pattern makes the requirement that the two arguments of a method are of the same type:

```
firstEq2nd := method (X,X);
```

Similarly, the pattern

```
return_arg := RetType (*,RetType,*);
```
(4.8)

matches any method whose return type is the same as the type of one of its arguments.

The binary predicate **`is`** forces equality of two variables, as in the following rewrite of (4.8):

```
return_arg := RetType (*,ParamType,*), RetType is ParamType;
```

The **`is`** predicate proves particularly useful when queries are used (see e.g. Section 4.1.7).

4.1.4 Predicates

Patterns are parameterless predicates. In general, it is possible to define predicates taking any number of parameters. As usual in logic programming, *parameters* are nothing more than externally accessible variables. Consider for example the predicate

```
is_static[C] := static field C;
```
(4.9)

which takes parameter `C`. When invoked with a specific value for parameter `C`, pattern `is_static` matches only **`static`** fields of that exact type.

Conversely, if the predicate is invoked without setting a specific value for `C`, then it will assign to `C` the types of all **`static`** fields of the class against which it is matched. The semantics by which a parameter to a predicate can be used as *either* input or output is standard in logic programming; the different assignments to `C` are made by the evaluation engine.

Note however that since JTL uses a database-, DATALOG-like semantics, rather than the recursive evaluation engine of PROLOG, each type `C` satisfying (4.9) will show only once in the output, even if there two or more fields of that type.

(Note that because square brackets denote parameter passing, array types in JTL must be preceded by a slash or enclosed in a pair of slashes, even for arrays of primitives or pre-declared types. For example, **`/int[] field`** will match any field of type **`int[]`**.)

Native Predicates

JTL has several native parameterized predicates. The names of many of these are JAVA keywords. For example, predicate **`implements`**`[I]` holds for all classes which implement directly the parameter `I` (an **`interface`**).

This is the time to note that the predicates **`implements`**`[I]` and `is_static[C]`, just as all other patterns presented so far, have a hidden argument, the *receiver*, also called the *subject* of the pattern, which can be referenced as **`This`** or #.

Other native predicates of JTL include `members[M]` (true when `M` is one of **`This`**'s members, either inherited or defined), `defines[M]` (true when `M` is defined by **`This`**), `overriding[M]` (true when **`This`** is a method which overrides `M`), `inner[C]` (true when `C` is an inner class of **`This`**), and many more.

The following example shows how a reference to **`This`** is used to define a pattern matching what is known in C++ jargon as “copy constructors”:

```
copy_ctor := constructor[T], T.members[This];
```
(4.10)

This example also shows how a predicate can be applied to a subject which is not the default, by using a JAVA-like dot notation.

The `copy_ctor` predicate works like this: first, the pattern `constructor[T]` requires that the matched item, i.e., **`This`**, is a constructor, which accepts a single parameter of some type `T`. Next, `T.members[This]` requires that **`This`**—the matched constructor—is a member of its argument type `T`, or in other words, that the constructor’s accepted type is the very type that defines it.

Literals, just as variables, can be used as actual parameters. For example,

```
class implements[M]
```

matches any class that implements interface `M`, whereas

```
interface extends[/java.io.Serializable]
```
(4.11)

matches any interface that extends the `Serializable` interface.

The square brackets in an invocation of a predicate with a single parameter are optional. Thus, (4.11) could have also been written as:

```
interface extends /java.io.Serializable
```

Moreover, since there is a clear lexical distinction between parameters and predicates, even the dot notation is not essential for changing the default receiver. Thus,

```
copy_ctor := constructor[T], T members This;
```

is equivalent to (4.10).

Standard Predicates

JTL also has a library of standard predicates, many of which are defined as a transitive closure of the native predicates. Table 4.1 shows a sample of these.

```
1  inherits[M]      :=  members[M] !defines[M];
2  container[C]     :=  C.members[This];
3  precursor[M]     :=  M.overriding[This];
4  implementing[M]  :=  !abstract, overriding[M], M.abstract;
5  abstracting[M]   :=  abstract, overriding[M], !M.abstract;
6  extends+[C]      :=  extends[C] | extends[C'], C'.extends+[C];
7  extends*[C]      :=  C is This | extends+[C];
8  interfaceof[C]   :=  C.class & implements[This];
9  interfaceof+[C]  :=  C.implements+[This];
10 interfaceof*[C]  :=  C.implements*[This];
```

Table 4.1: Some of the standard predicates of JTL, and their definitions

The figure makes apparent the JTL naming convention by which the reflexive transitive closure of a predicate p is named p`*`, while the anti-reflexive variant is named p`+`. The myriad of recursive

definitions such as these saves much of the user's work; in particular it is rare that the programmer is required to employ recursion.

It is interesting to examine the "recursive" definition of one of these predicates, e.g., the definition of `extends+` (line 6). It may appear at first that with the absence of a halting condition, the recursion will never terminate. A moment's thought reveals that this is not the case. Since JTL uses a bottom-up construction of facts, starting at a fixed database, the semantics of this recursive definition is not of stack-based recursive calls, but rather a dynamic programming, or work-list, approach for generating facts.

The definition of **`instanceof`** (line 8) uses the *subject-chaining operator*, "&". The predicate that appears after this operator is applied to the same subject as the predicate that appears before it; thus, the definition is equivalent to

```
interfaceof[C] := C.class C.implements[This];
```

We find the version using subject-chaining to be more readable.

Predicate Name Aliases

The name `extends+` suggests that it is used as a verb connecting two nouns. As mentioned above, we can even write

```
C extends+ C'
```

But, the same predicate can be used in situations in which, given a class `C`, we want to generate the set of *all* classes that it extends. A more appropriate name for these situations is `ancestors`. (An example for the use of the alternative name `ancestors` appears in predicate (4.13) below.) It is possible to make another definition

```
ancestors[C] := extends+[C];
```

However, to promote meaningful predicate names, JTL offers what is known as *predicate name aliases*, by which the same predicate definition can introduce more than one name to the predicate. Aliases are written as a *definition annotation* which follows the main rule. The definition of `extends+`, for example, has such an alias:

```
extends+[C] := extends C | extends C', C'.extends+[C];
               Alias ancestors;
```

Native predicates can also have aliases, which are specified along with their declaration.

4.1.5 Set Queries

As mentioned previously, JTL's expressive power is that of FOPL*. Although it is possible to express universal and existential quantification with the constructs of logic programming, we found that the alternative presented in this section is more natural for the particular application domain.

Consider for example the task of checking whether a JAVA class has an **`int`** field. A straightforward, declarative way of doing that is to examine the set of all of the class fields, and then check whether this set has a field whose type is **`int`**.

The following pattern does precisely this, by employing a *query* mechanism:

```
has_int_field := class members: {
                   exists int field;
                 };
```
(4.12)

Here, the query `members: {` $Q_1; \cdots ; Q_n$ `}` generates first the set of all possible members `M`, such that `#.members[M]` holds. The "`members:`" portion of the query is called the *generator*.

This set is then passed to Q_1 through Q_n, the *quantifiers* embedded in the curly brackets. The entire query holds if all of these quantifiers hold for this set.

In (4.12), there was only one quantifier: the JTL statement **`exists int`** `field` is an existential quantifier which holds whenever the given set has an element which matches the pattern **`int`** `field`.

The next example shows two other kinds of quantifiers.

```
class ancestors: {
    all public;
    no abstract;
};
```
(4.13)

The evaluation of this pattern starts by computing the generator. In this case, the generator generates the set of all classes that the receiver **`extends`** directly or indirectly, i.e., all types `C` for which `#.ancestors[C]` holds. (Recall that `ancestors` is an alias for `extends+`, defined above.) The first quantifier checks whether all members of this set are **`public`**. The second quantifier succeeds only if this set contains no **`abstract`** classes. Thus, (4.13) matches classes whose superclasses are all public and concrete.

In addition to **`exists`**, **`all`**, and **`no`**, quantifiers in JTL include **`many`** p, which holds if the queried set has two or more elements for which pattern p holds, and **`one`** p, which holds if this set has precisely one such element.

The existential quantifier is the most common; hence the **`exists`** keyword is optional. Also, a missing generator (in predicates whose subject is a **`TYPE`**) defaults to the `members:` generator. Hence, a concise rewrite of (4.12) is

```
has_int_field := class {
                   int field;
                 };
```
(4.14)

The `members` generator produces all the fields, methods, constructors as well as the static initializer that are *recognized* within the body of a class, regardless of their visibility level. This includes members which either override or hide inherited members, as well as members which were inherited from any of its superclasses, but not overridden or hidden.

Other standard predicates which are useful as generators over class members include `defines`—all members that the class defined, either in the first time, or in overriding inherited members; `protocol`—all non-**`private`** members of a class, including inherited ones, that were not overridden (or hidden) due to inheritance; `holds`—all members (including **`private`**, inherited, overridden, and hidden members) that a class has; and `offers`—similar to `holds`, but excludes the members that were declared in `java.lang.Object`. Table 4.2 summarizes the differences between these five predicates. Of these, only `holds` is a primitive predicate; the others are part of the standard library, defined in terms of simpler predicates. For example, `defines` can be defined thus:

```
defines[M] := holds[M], ![extends*[T], T.holds[M]];
```

In all the examples shown here, the generator was a predicate with a single named parameter and an implicit receiver. In such cases, the generator generates a set of primitive values, which are the possible assignments to the argument. However, in general, the generator generates a relation of named tuples, and the quantifiers are applied to the set of these tuples. We discuss the underlying semantics of queries in greater detail in Section 4.2.

Generating predicate	Newly-defined members	Inherited members	Private members	Overridden & hidden members	Members from `Object`
`members`[a]	✓	✓	✓		✓
`defines`	✓		✓		Only for `Object` itself
`protocol`	✓	✓			✓
`holds`	✓	✓	✓	✓	✓
`offers`	✓	✓	✓	✓	

[a] Default generator.

Table 4.2: Predicates commonly used as generators for queries about class members

4.1.6 List Queries

List queries are very similar to set queries. They are specified using regular parenthesis, "`(`" and "`)`", rather than curly braces. Inside these parenthesis appear quantifiers (**`exists`**, **`all`**, etc.), just as in set queries. The difference in evaluation is that with list queries, JTL searches for a disjoint partitioning of the list into sublists, such that these sublists satisfy, as sets, the quantifiers, in order. Therefore, list queries are meaningful only for generators that provide an ordered list (an element of type **`MEMBERS`**, **`TYPES`**, etc.) rather than an unordered set.

One such generator is the primitive binary predicate `args`. Its semantics are such that `M.args[Ts]` holds if `M` is a method (and therefore of kind **`MEMBER`**) and `Ts`, of kind **`TYPES`**, is the list or argument types of that member. Elements of `Ts` can then be elicited using two binary primitive predicates: `head_is` and `tail_is`, which are the JTL equivalent of the standard, LISP-old, system of representing lists. Together with the unary predicate **`null`**, testing for emptiness, one may use `args` to write arbitrary recursive predicates for any desired iterative processing of the list of arguments of a method. Using list queries, however, provides a simpler alternative.

A list quantification pattern, such as

```
args: (many abstract,int,exists final,one public)
```

is evaluated in two steps: (a) list generation; and (b) application of the quantified conditions to the list—this is achieved by searching for a disjoint partitioning of the list into sublists that satisfy the quantifiers. In the above example, the list generation is carried out by searching for a variable `Ts` such that `#.args[Ts]` holds, and then applying the four quantifiers to it. The predicate holds if there are sublists $\mathtt{T}_1$, $\mathtt{T}_2$, $\mathtt{T}_3$, and $\mathtt{T}_4$, such that `Ts` is the concatenation of the four, and it holds that:

- There is more than one **`abstract`** type in $\mathtt{T}_1$,
- Sublist $\mathtt{T}_2$ has precisely one element, which matches **`int`**,
- There is at least one **`final`** type in $\mathtt{T}_3$, and
- There is exactly one **`public`** type in $\mathtt{T}_4$.

With list queries, there is no default quantifier; instead, a predicate expecting a list parameter is considered a quantifier, and a predicate expecting a *list element* parameter is the quantifier requiring that the respective sublist has exactly one element matching this pattern.

The default generator for list queries is `args:` (compared with `members:` for set queries). Now, the argument list pattern `()` is shorthand for `args:(empty)`, while `(*)` is shorthand

for `args:(all true)`. Similarly, the argument list pattern `(_,String,*)` from (4.7) is shorthand for `args:(one true,String,all true)`.

JTL's current specification does not support all of the extensions presented in JAVA 5. In particular, generic type parameters are currently not handled. However, processing of such arguments will probably be through a list enclosed in angular brackets.

4.1.7 Pedestrian Queries of Imperative Code

The executional aspect of JAVA code remained beyond the basic functionality of JTL. This aspect is primarily method bodies, but also other imperative code, including constructors, field initializers and static initializers.

Core JTL includes predicates for what we term *pedestrian* code queries, that make it possible to explore the fields and methods that executional code uses. A JTL extension which introduces a new type, **`SCRATCH`**, and enables rich data-flow analysis in JTL queries was developed by Gil and Maman [67]. (That work also explains how JTL can be extended to support abstract syntax tree (AST) queries, and why we chose not to create such an extension.) The remainder of this section focuses on pedestrian code queries, which prove useful in many AOP-related scenarios as well as in other cases.

In studying a given class, it is useful to know which methods use which fields. The JTL pattern in Figure 4.1, for example, implements one of Eclipse's [89] warning situations, in which a **`private`** field is defined but never used.

```
1 unused_private_member := private field,
2    This is F,
3    declared_in C, C inners*: {
4       all !access[F];
5    }
```

Figure 4.1: Detecting unused **`private`** fields using JTL

The pattern fetches the class `C` that defines the field (line 3), and then uses the reflexive and transitive closure of the `inner` relation to examine `C`, its inner classes, their inner classes, etc., to make sure that none of these reads or writes to this field. (The unification **`This is`** `F`, in line 2, is for making the receiver field accessible inside the curly brackets.)

The pattern `access` showing in line 4 of the Figure is defined in the JTL library. The definition, along with some of the other standard patterns that can be used in JTL for pedestrian code queries is shown in Table 4.3. Such queries model the method body as an unordered set of byte-code instructions, checking whether this set has certain instructions in it.

```
1 access[F]  :=  read F | write F; Alias accesses;
2 read[F]    :=  offers M, M read F; Alias reads;
3 write[F]   :=  offers M, M read F; Alias writes;
4 calls[M]   :=  invokes_interface[M] |
                 invokes_virtual[M] |
                 invokes_static[M] |
                 invokes_special[M];
                 Alias invokes, invoke;
5 use[X]     :=  access[M] | invoke[M]; Alias uses;
```

Table 4.3: Standard JTL predicates for pedestrian code queries

In the figure we see that the definition of `access` is based on the overloaded predicates `read` and `write`. The native predicate `read[F]` holds if the receiver is a method whose code reads the field `F`, whereas `write[F]` holds if the receiver is a method whose code writes to that field. The second line of the table overloads the native definition of `read`, so that it applies also to receivers whose kind is **`TYPE`**. The third line overloads `write` in a similar manner. It follows that the definition of `access` in the table is overloaded, and it applies to classes and methods.

The table also makes uses of the four other pedestrian natives for inspecting code: `invokes_interface`, `invokes_virtual`, `invokes_static`, and `invokes_special`. (These natives also have aliases identical to the bytecode mnemonics.)

With this minimal set of six natives and five standard predicates, several interesting patterns can be defined. For example, predicate

```
creates[T] := invokes_static[M], M.ctor & declared_in[T];
```

is true when the receiver creates an object of type `T`. Also, the following predicates test whether a method *refines* its precursor:

```
refines[M] := overrides[M], invokes_special[M];
refiner := refines[_];
```

And the following predicate checks whether a method is not empty:

```
does_something := !void | invokes[_] | writes[_] | native;
```

(If a method does not return a value, does not invoke any other method, nor write to a field, then it must have no meaningful effect.)

With the above, we implemented an interesting PMD [197] rule, signalling an unnecessary constructor, i.e., the case that there is only one constructor, it is public, has an empty body, and it takes no arguments:

```
unnecessary_constructor := class {
    constructor => public () !does_something;          (4.15)
}
```

4.2 Underlying Semantics

As stated above, JTL belongs to the logic programming paradigm. This section explains how the JTL constructs are mapped to familiar notions of the paradigm.

In a nutshell, JTL is a *simply typed* formalism whose underlying semantics is *first order predicate logic* augmented with *transitive closure* (FOPL*). Evaluation in JTL is similar to that of PROLOG (more precisely, DATALOG), with its built-in support for the "join" and "project" operations of relational databases. This section elaborates the language semantics a bit further.

Kinds and Predicates

The type system of JTL consists of a fixed finite set of primitive kinds (types) $\mathcal{T}$. There are no compound kinds.

A *predicate* is a boolean function of $T_1 \times \cdots \times T_n$, $n \geqslant 0$, where $T_i \in \mathcal{T}$ for $i = 1, \ldots, n$. A predicate can also be thought of as a *relation*, i.e., a *subset* of the cartesian product

$$T_1 \times \cdots \times T_n,$$

called the *domain* of the predicate. By convention, the first argument of a predicate is unnamed, while all other arguments are named. The unnamed arguments is called the *receiver* or the *subject*. It can be accessed using the keyword **`This`** or the symbol #.

Native Predicates

JTL has a number of native predicates, such as **`class`**—a unary predicate of **`TYPE`**, i.e., **`class`**$\subseteq$**`TYPE`**, **`synchronized`**$\subseteq$**`MEMBER`** (the predicate which holds only for synchronized methods), `members`$\subseteq$**`TYPE`**$\times$**`MEMBER`**, **`extends`**$\subseteq$**`TYPE`**$\times$**`TYPE`** (with the obvious semantics), and the *0-ary* predicates **`false`** (an empty 0-ary relation) and **`true`** (a 0-ary relation consisting of a single empty tuple). Built-in predicates are called in certain communities *Extensional Database* (EDB) predicates.

A run of the JTL processor starts by loading a declaration of arity and argument types of all native predicates from a configuration file. Native declarations are nothing more than definitions without body. For example, the following commands in a configuration file

```
MEMBER.int;
```

states that **`int`** is a unary predicate such that **`int`**$\subseteq$**`MEMBER`**.

Compound Predicates

Conjunction, disjunction and negation can be used to define *compound* predicates from the built-ins. Also permitted are *quantification*, as explained in Section 4.1, and *transitive closure*, i.e., recursion—as in:

```
extends+[X] := extends X | extends[Y], Y.extends+[X];
```

The language offers an extensive library of *pre-defined*, or *standard* compound predicates. Compound predicates are sometimes called *Intensional Database* (IDB) predicates.

Finite Databases

To run, a JTL program requires a database which *conforms* to the natives, i.e., it must have in its schema the relations or the EDBs as dictated by the set of natives defined by the JTL implementation at hand.

The simplest way to supply a database is by specifying to the JTL processor a finite set of classes and methods, e.g., a `.jar` file. Obviously, such a collection does not directly represent any EDBs. EDBs are realized on top of the collection by means of a bytecode analysis library.

Alternatively, a finite database can also be provided by supplying a finite set of legal JAVA source files. The native relations are then realized on top of these by a JAVA parser.

JTL predicates can also be run without a fixed input set. Such a situation, can be thought of as a DATALOG query over an infinite database.

Evaluation Order

Unlike PROLOG, the order of evaluation in JTL is unimportant. The output set of a pattern is the same regardless of the order by which its constituent predicates in it are invoked. Predicates have no side-effects, and all computations (on finite databases) terminate.

The simplest way to compute the output set is bottom-up, i.e., by using a work-list algorithm which uses the program rules to compute all tuples in all IDBs used by the program goal. This process, although guaranteed to terminate, can be very inefficient, both time- and space-wise. JTL instead analyzes predicates and applies, whenever possible, a more efficient top-down evaluation strategy

Overloading and Kind Inference

The JTL processor includes a *kind inference engine* which, based on the kind of arguments and arity of the native predicates, infers arity and arguments kinds of predicates defined on top of these. For example, the definition

```
real := double | float;
```
(4.16)

implies that `real`⊆**`MEMBER`**.

JAVA's overloading of keywords carries through to JTL, e.g., since the JAVA keyword **`final`** can be applied to classes and members, the built-in predicate **`final`** in JTL is overloaded, denoting two distinct relations: **`final`**$_1$ ⊆**`TYPE`** and **`final`**$_2$ ⊆**`MEMBER`**. Many native predicates are similarly overloaded; JTL infers overloading of compound predicates. For example, the conjunction of **`final`** and **`public`** is overloaded; the conjunction of **`final`** and **`interface`** is not.

The Default Receiver

As seen in the last examples, JTL sports an implicit mechanism of applying a predicate to receiver. For example, the definition of `real` in (4.16) could have been written as

```
real := #.double | #.float;
```
(4.17)

Named Arguments

The *signature* of a relation is an ordered pair $\langle R, \mathbf{A}\rangle$, whose first component, $R \in \mathcal{T}$, defines the type of the receiver, while the second component, $\mathbf{A} = \{\langle \ell_1, A_1\rangle, \ldots \langle \ell_m, A_m\rangle\}$, defines the names of the arguments (the labels ℓ_j, $j = 1, \ldots, m$, must be distinct) and their types ($A_j \in \mathcal{T}$ for $j = 1, \ldots, m$).

A row of a relation is in general a *named tuple*, i.e., a tuple of values, where all but the first carry labels, such that the types of these values and the labels they carry match exactly the signature of the predicate.

Predicates are characterized by signature, e.g., the signature of predicate `members` is $\big\langle$**`TYPE`**, $\{\langle$"`M`", **`MEMBER`**$\rangle\}\big\rangle$, while the definition

```
container[C] := C.members[#]
```

implies

$\big\langle$**`MEMBER`**, $\{\langle$"`C`", **`TYPE`**$\rangle\}\big\rangle$

as the signature of `container`. Overloaded predicates have multiple signatures, one for each meaning. For example, the built-in predicate **`final`** has two signatures, $\big\langle$**`MEMBER`**, $\emptyset\big\rangle$ and $\big\langle$**`TYPE`**, $\emptyset\big\rangle$.

Set and List Queries

JTL extends logic programming with what we call a *query*, which is a predicate whose evaluation involves the generation of a temporary relation, and then applying various *quantifiers* to this relation. A set query predicate is true if all the quantifies hold for the generated temporary relation; a list query predicate is true if the list can be decomposed into a series of sublists so that each quantifier holds for the corresponding sublist, in order.

```
1 classical := class members: {
2    has field;
3    many method;
4    no static;
5    method => public;
6    field => private;
7    disjoint setter, getter;
8 }
```

Figure 4.2: A JTL predicate for matching "classical class" notion.

The predicate defined in Figure 4.2 tries to check that a class is "classical," i.e., that it has at least one field, two or more methods, that all methods are public, all fields are private, that there are no static fields or methods, and that the sets of "setters" and "getters" of this class are disjoint. (The definition in the figure assumes that predicates `setter` and `getter` were previously defined.)

The essence of the example is the *generator* of the temporary relation, written as `members:`. The colon character (`:`) appended to predicate `members` makes it into a generator. JTL generates the set of all members `M`, such that `#.members[M]` holds. This set, which can be also thought of as a relation with only one unnamed column, is subjected to the set expressions inside the curly brackets.

Six conditions are applied to this set: the first (line 2) is an existential quantifier (**`has`** is synonymous to **`exists`**) requiring that at least one element in the generated set satisfies the `field` condition, i.e., that the class has at least one field. The second condition (line 3) similarly requires that `method` holds for two more members. The 3^{rd} condition (line 4), as should be obvious, requires that this set does not contain any static members.

The conditions in lines 5–6 are *set expressions*. The first requires that the predicate `method` $\Rightarrow$ **`public`** (read: "method implies public") holds in this set, i.e., that all method members are public. The next condition similarly states that the set of `field` members is a subset of the set of **`private`** members. Finally, the set expression **`disjoint`** `setter, getter` requires that the two subsets obtained by applying predicates `setter` and `getter` to the set of class members are disjoint.

4.2.1 Translating JTL into Datalog

As stated earlier, JTL provides high-level abstraction on top of a DATALOG core. We will now briefly illustrate how JTL source code can be translated into DATALOG.

The first translation step is that of the subject variable: the subject variable in JTL (either implicit or explicit) is translated into a standard DATALOG variable which prepends all other actual arguments. For example, the JTL expression

```
p1 := abstract, extends X, X abstract;
```

is equivalent to this DATALOG expression:

```
p1(This) :- abstract(This), extends(This,X), abstract(X).
```

(In a sense, this is reminiscent of how early C++ compilers worked by translating the code to standard C [149], and prepended the hidden variable **`this`** to class methods when represented as simple functions.)

Disjunctive expression are not as simple since DATALOG requires the introduction of a new rule for each branch of a disjunctive expression. Thus,

```
p2 := public [interface | class];
```

is translated into:

```
p2(This) :- public(This), aux(This).
aux(This) :- interface(This).
aux(This) :- class(This).
```

The following predicate poses a greater challenge:

```
p3[T] := public extends T [T abstract | interface];
```

Here, the parameter `T` appears in the **`extends`** invocation and also on the left-hand side of the disjunction, but not on the right-hand side. The translation into DATALOG requires the use of a special EDB predicate, `always(X)`, which holds for every possible `X` value:

```
p3(This) :- public(This), extends(This,T), aux(This,T).
aux(This,T) :- interface(This), always(T).
aux(This,T) :- abstract(T), always(This).
```

The translation of quantifiers relies on the natural semantics of DATALOG, where every predicate invocation induces an implicit existential quantifier. For example,

```
p4 := class members: { abstract; };
```

Is equivalent to this DATALOG definition:

```
p4(This) :- class(This), members(This,M), abstract(M).
```

By using negation, we can express the universal quantifier in terms of the existential one, the negative quantifier in terms of the universal one, etc.

The examples presented here highlight the fundamentals of the JTL to DATALOG mapping. The program-transformation extension requires certain changes to the translation algorithm, as discussed in Section 4.6.

4.3 Using JTL in Aspect-Oriented Systems

We now examine how JTL can be used to serve the two main purposes for which it was created, as presented in the preamble of the current chapter (page 63): *join-point selection for aspect-oriented programming* and *expressing the conditions making up concepts*. Both uses can be of great benefit not only to languages that integrate shakein semantics, but to other aspect-oriented solutions as well.

4.3.1 Specifying Pointcuts Using JTL

The limited expressive power of the pointcut specification language of ASPECTJ (and other related AOP languages, e.g., CAESAR [171] and ASPECTJ2EE), has been noted several times in the literature [91, 126, 128, 187].

We propose that JTL is integrated into AOP processors, taking charge of pointcut specification. To see the benefits of using a JTL component for this purpose, consider the following ASPECTJ pointcut specification:

```
call(public void *.set*(*));
```

JTL's full regular expressions syntax can be used instead, by first defining

```
setter := public void 'set[A-Z]?*'(_);                    (4.18)
```

and then writing `call(setter)`.

This simple example exhibits two distinct advantages. First, JTL (unlike ASPECTJ) uses proper regular expressions; thus, while the ASPECTJ variant (mistakenly) matches a method called, e.g., "`settle`", the JTL variant does not match it (since it expects an uppercase letter after the prefix "`set`"). Second, the pattern is *named*, making it *reusable* while also making the pointcut specification more expressive. Still, if brevity is desired, anonymous patterns can also be used directly in pointcut specifications.

To understand the benefit of named patterns, consider a case where the same set of program elements needs to be referred to more than once. Assume we wish to tap both read and write operations to all primitive-typed public fields. In ASPECTJ, this is a verbose pointcut, as shown in Figure 4.3. Not only tedious, it is also error prone, since a major part of the code is replicated across all definitions.

```
get(public boolean *)  || set(public boolean *)  ||
get(public byte *)     || set(public byte *)     ||
get(public char *)     || set(public char *)     ||
get(public double *)   || set(public double *)   ||
get(public float *)    || set(public float *)    ||
get(public int *)      || set(public int *)      ||
get(public long *)     || set(public long *)     ||
get(public short *)    || set(public short *);
```

Figure 4.3: An ASPECTJ pointcut definition for all read- and write-access operations of primitive public fields.

By using disjunction in JTL expressions, the ASPECTJ code from Figure 4.3 can be greatly simplified if we allow pointcuts to include JTL expressions:

```
primitive := boolean | byte | char | double |
             float | int | long | short;
ppf := public primitive field;

get(ppf) || set(ppf); //JTL−based AspectJ pointcut
```

(In truth, `primitive` is a standard library predicate.) The ability to name predicates, such as `ppf` in the example, makes it possible to turn the actual pointcut definition into a concise, readable statement.

But there is more to using JTL than conciseness. Figure 4.4 is an example of a condition that is impossible to specify in ASPECTJ. Condition `field_in_plain_class` holds for **`public`** fields in a class which has no getters or setters. This requirement is realized by predicate `container`, which captures in `C` the container class. A query is then used to examine the other members of the class.

```
setter := public void 'set[A-Z]?*'(_);
boolean_getter = boolean 'is[A-Z]?*'();
other_getter = !boolean !void 'get[A-Z]?*'();
getter := public [boolean_getter | other_getter];

field_in_plain_class := public field,
   declared_in[C], C.members: {
       no getter;
       no setter;
  };
```

Figure 4.4: A JTL pointcut that cannot be expressed in ASPECTJ

The example from Figure 4.4 could have been implemented in other suggested alternatives to the ASPECTJ pointcut specification language, but not without a loop or a recursive call.

Figure 4.4 belongs to a set of pointcut definitions that present requirements not only about the join points themselves, but also on related classes—here, the class in which the matched method or field is defined, but possibly on other classes (such as parameter types) as well. To understand the importance of this ability, consider the following example. Assume we wish to apply an aspect to all classes that extend class `Resource`. This aspect specifically advises the class constructors, making them work by managing an instance pool. In ASPECTJ, this advice will use a pointcut such as:

```
execution(Resource+.new(..))
```

However, let us further assume that we can only pool resources that implement the `Serializable` interface. This is a very simple condition on the class containing the matched constructors, and in fact it can still be expressed using ASPECTJ:

```
execution(Resource+.new(..)) && execution(Serializable+.new(..))
```

i.e., we match constructors that are both in a subclass of `Resource` and in an implementing class of `Serializable`. The JTL-based alternative is perhaps easier to understand:

```
ctorInSerializableResource := constructor in C,
   C extends* Resource & implements* Serializable;

execution(ctorInSerializableResource) //Pointcut in AspectJ using JTL
```

As the requirements we present for the containing class get more complicated, ASPECTJ is unable to express them. For example, we might require that the advice is applied only to `Resource` subclasses that have no reference-type fields (say, to prevent the serialization of large object graphs). The following JTL predicate expresses this requirement:

```
ctorInSerializableResource := constructor in C,
   C extends* Resource & implements* Serializable & {
      no !primitive field;
   };
```

In effect, we are using JTL to express a *concept* about the join-points' containing type. The use of JTL to define concepts is discussed in detail in the Section 4.3.2 below.

Our contribution puts the expressive power of JTL at the disposal of ASPECTJ and other aspect languages, replacing the sometimes ad-hoc pointcut definition language with JTL's systematic approach. JTL replaces only the member-selection syntax, so that each language can have its own pointcut semantics (e.g., support for control-flow pointcuts such as **`cflow`** in ASPECTJ). And while JTL itself only examines the static structure of the code, JTL pointcut expressions can be used in dynamic pointcut definitions as well. A variant of ASPECTJ that embeds JTL, for example, can use expressions such as

```
execution(someJTLPredicate) && !cflowbelow(someJTLPredicate)
```

to filter off recursive calls. Languages that do not support dynamic pointcut expressions (like **`cflowbelow`**) can generate advice that use JTL's library at runtime for any additional required tests.

4.3.2 Concepts for Generic Programming

In the context of generic programming, a *concept* [106, 124, 210] is a set of constraints which a given set of types must fulfil in order to be used by a generic module. As a simple example, consider the following C++ template:

```
template<typename T>
   class ElementPrinter {
   public:
   void print(T element) {
      element.print();
   }
}
```

The template assumes that the provided type parameter `T` has a method called `print`, which accepts no parameters. Viewing `T` as a single-type concept, we say that the template presents an implicit assumption regarding the concept it accepts as a parameter. Implicit concepts, however, present many problems, including hurdles for separate compilation, error messages that Stroustrup et al. term "of spectacular length and obscurity" [210], and more.

With Java generics, one would have to define a new interface:

```
interface Printable {
   void print();
}
```

and use it to confine the type parameter. While the concept is now explicit, this approach suffers from two limitations: first, due to the nominal subtyping of JAVA, generic parameters must explicitly implement interface `Printable`; and second, the interface places a "baggage" constraint on the return type of `print`, a constraint which is not required by the generic type.

Using JTL, we can express the concept explicitly and without needless complications, thus:

```
(class | interface) {
   print();
};
```

There are several advantages for doing that: First, the underlying syntax, semantics and evaluation engine are simple and need not be re-invented. Second, the JTL syntax makes it possible to make useful definitions currently not possible with JAVA standard generics and many of its extensions.

The importance of concepts is not limited to the field of generic programming. It is also of relevance to aspect-oriented programming in general, and to shakeins in particular, because, like generic types, shakeins (and aspects) also make assumptions about the classes to which advice are applied. By expressing the concepts in JTL, we can make these assumptions explicit.

The problem of expressing concepts is more thorny when multiple types are involved. A recent work [106] evaluated genericity support in 6 different programming languages (including JAVA, C# [133] and EIFFEL [141]) with respect to a large-scale, industrial-strength, generic graph algorithm library, reaching the conclusion that the lack of proper support for multi-type concepts resulted in awkward designs, poor maintainability, and unnecessary run-time checks.

JTL predicates can be used to express multi-type concepts, and in particular each of the concepts that the authors identified in this graph library.

As an example, consider the `memory_pool` concept, which is part of the challenging example the concepts treatise used by Garcia et al. A memory pool (sometimes called an *instance pool* or *object pool* [122]) is used when a program needs to use several objects of a certain type, but it is required that the number of instantiated objects will be minimal. In a typical implementation, the memory pool object will maintain a cache of unused instances. When an object is requested from the pool, the pool will return a previously cached instance. Only if the cache is empty, a new object is created by issuing a create request on an appropriate factory object.

More formally, the memory pool concept presented in Figure 4.5 takes three parameters: `E` (the type of elements which comprise the pool), `F` (the factory type used for the creation of new elements), and **`This`** (the pool type).

```
name create, instance, acquire, release;

factory[E] := (class | interface) {
  public constructor ();
  public E create();
};

memory_pool[F,E] := This is T, {
  public static T instance();
  public E acquire();
  public release(E);
}, F.factory[E];
```

Figure 4.5: The `memory_pool` multi-type concept

The body of the concept requires that **`This`** will provide `acquire()` and `release()` methods for the allocation and deallocation (respectively) of `E` objects, and a static `instance()` method to allow client code to gain access to a shared instance of the pool. Finally, it requires (by invoking the `Factory` predicate) that `F` provides a constructor with no arguments, and a `create()` method that returns objects of type `E`.

As shown by Garcia et al., the requirements presented in Figure 4.5 have no straightforward representation in JAVA, C# or EIFFEL. In particular, using an **`interface`** to express a concept presents extraneous limitations, such as imposing a return type on `release`, and it cannot express other requirements, such as the need for a zero-arguments constructor in a factory. Using an **`interface`** also limits the applicable types to those that implement it, whereas the concept itself places no such requirement.

In a language where JTL concept specifications are supported, a generic module parameterized by types `X`, `Y` and `Z` can declare, as part of its signature, that `X.memory_pool[Y,Z]` must

hold. This will ensure, at compile-time, that `X` is a memory pool of `Z` elements, using a factory of type `Y`. Thus, concepts may be regarded as the generic-programming equivalence of the *Design by Contract* [169] philosophy.

Concepts are not limited to templates and generic types. Mixins, too, sometimes have to present requirements to their type parameter. The famous `Undo` mixin example [8] requires a class that defines two methods, `setText` and `getText`, but does not define an `undo` method. The last requirement is particularly important, since it is used to prevent *accidental overloading*. However, it cannot be expressed using JAVA interfaces. The following JTL predicate clearly expresses the required concept:

```
undo_applicable := class {
   setText(String);
   String getText();
   no undo();
};
```

In summary, we propose that in introducing advanced support of genericity and concepts to JAVA, one shall use the JTL syntax as the underlying language for defining concepts. In addition to the two benefits listed above (simple semantics and evaluation, useful definitions not possible in standard JAVA), using JTL also puts intriguing questions of type theory in the familiar domain of logic, since, as mentioned earlier, JTL is based on FOPL*. For example, the question of one concept being contained in another can be thought of as logical implication. Using text book results [36], one can better understand the tradeoff between language expressiveness and computability or decidability. Work is currently underway for defining a JTL sub-language, restricting the use of quantifiers, which assures decidability of concept containment.

4.4 Additional Applications

As JTL took shape and grew older it became clear that it can be used not only for its two original uses, namely pointcuts and concepts, but also for other software engineering tasks, primarily as a tool to assist programmers understand the code they must maintain. This section provides a brief survey of some of these additional uses: integration in development environments, LINT-like tests, and more. One additional use not covered here is the detection of *micro-patterns* [110], which relies on a JTL extension developed by Gil and Maman for examining data-flow analysis inside methods [67, 68].

4.4.1 Integration in CASE Tools and IDEs

In their work on JQuery, Janzen and De Volder [143] make a strong case, including empirical evidence, for the need of a good software query tool as part of the development environment. We argue that the querying (but not the navigational) side of JQuery can be replaced and simplified by JTL.

To prove this claim, an Eclipse [89] plug-in that runs JTL queries and presents the result in a dedicated view was developed by Itay Maman and others. Figure 4.6 shows an example: the program (which appears above the results) found all classes in JAVA's standard library for which instances are obtained using a **`static`** method rather than a constructor.

Using JTL, many searches can be described intuitively. For example, to find all classes that share a certain annotation `@X`, the developer simply searches for `@X` **`class`**. The similarity between JTL syntax and JAVA declarations (which we refer to as the *minimal abstraction gap*) will

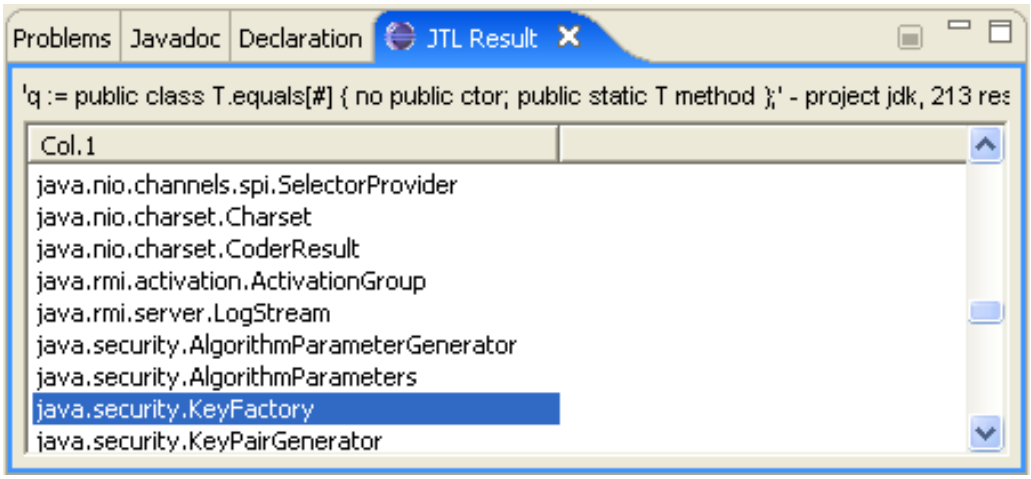


Figure 4.6: Screenshot of the result view of JTL's Eclipse plugin

allow even developers who are new to JTL to easily and effectively sift through the overwhelming number of classes and class members in the various JAVA libraries.

JTL can also be used to replace the hard-coded filtering mechanism found in many IDEs (e.g., a button for showing only **`public`** members of a class) with a free-form filter. Figure 4.7 is a mock screenshot that shows how JTL can be used for filtering in Eclipse.

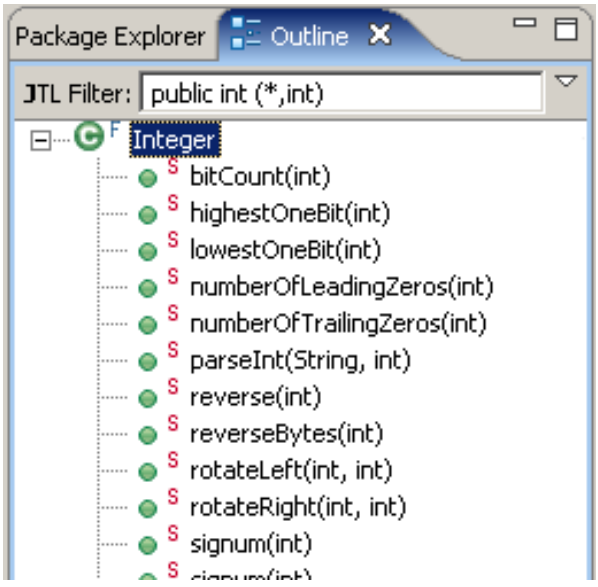


Figure 4.7: Using JTL for filtering class members (mock screenshot)

Finally, using the program transformation extension described in Section 4.6 below, JTL can be used for search-and-replace operations. Since the operation is context-sensitive, there is no risk of, e.g., changing text that appears in comments.

4.4.2 LINT-Like Tests

JTL can be used to express, and hence detect, many undesired programming constructs and habits. On the one hand, JTL's limitation with regard to the inspection of method bodies implies that it cannot detect everything that existing tools [99, 137, 198] can. In its current state, JTL cannot detect problematic constructs such as

```
if (C) return true else return false;
```

nor can it easily express numeric limitations (e.g., detecting classes with more than k methods for some constant k).

Yet on the other hand, JTL's expressiveness makes it easy for developers and project managers to improvise and quickly define new rules that are both enforceable and highly self-documenting. We have already seen a few examples for such tests: pattern (4.15) can be used to detect needlessly-defined constructors in classes; the pattern in Figure 4.1 detects unused **`private`** class members.

To further test this prospect, we wrote a collection of JTL patterns that implement the entire set of warnings issued by Eclipse and PMD [197] (a popular open-source LINT tool for JAVA). The only exceptions were those warnings that directly rely on the program source code (e.g., unused **`import`** statements), as these violations are not represented in the binary class file that we used.

For example, consider the PMD rule LOOSECOUPLING. It detects cases where the concrete collection types (e.g., `ArrayList` or `Vector`) are used instead of the abstract interfaces (such as `List`) for declaring fields, method parameters, or method return values—in violation of the library designers' recommendations. This rule is expressed as a 45-lines JAVA class, and includes a hard-coded (yet partial) list of the implementation classes. PMD does make a heroic effort, but it will mistakenly report (e.g.) fields of type `Vector` for some alien class `Vector` which is not a collection, and was declared outside of the `java.util` package.

The JTL equivalent is:

```
loose_coupling := (class|interface) {
   T method | T field | method(*, T, *);
}, T implements /java.util.Collection;
```

It is shorter, more precise, will detect improper uses of any class that implements any standard collection interface (without relying on an explicit list), and will not flag false positives.

4.4.3 Additional Applications

Several other potential uses for JTL include encapsulation policies and confined types, among others. *Encapsulation policies* were suggested by Scharli et al. [200] as a software construct for defining which services are available to which program modules. Using JTL, both the selection of services (methods) and the selection of modules (classes) can be more easily expressed.

Confined types [32] are another example in which JTL could be used, provided of course that confinement is represented in a form of annotation. We have not yet investigated the question of checking the imperative restrictions of confined types with JTL.

4.5 Related Work on Language Queries

Tools and research artifacts which rely on the analysis of program source code are abundant in the software world, including metrics [60] tools, reverse-engineering [20], smart CASE enhancements [135], configuration management [26], architecture discovery [116], requirement tracing [118], aspect-oriented programming [151], software porting and migration [157], program annotation [4], and many more.

The very task of code analysis per se is often peripheral to such products. It is therefore no wonder that many of these gravitate toward the classical and well-established techniques of formal language theory, parsing and compilation [2]. In particular, software is recurringly represented in these tools in an abstract syntax tree (AST).

JTL is different in that it relies of a flat relational model, which, as demonstrated in Section 4.5.2, can also represent an AST. (Curiously, there were recently two works [115, 164] in which relational queries were used in OO software engineering; however, these pertained to program execution trace, rather than to its static structure.)

JTL aspires to be a universal tool for tool writers, with applications such as specification of pointcuts in AOP, the expression of type constraints for generic type parameters and mixin parameters, selection of program elements for refactoring, patterns discovery, and more.

The community has already identified the need for a general-purpose tool or language for processing software. The literature describes a number of such products, ranging from dedicated languages embedded into larger systems to attempts to harness existing languages (such as SQL or XQUERY [30]) to this purpose. Yet, despite the vast amount of research invested in this area, no single industry standard has emerged.

A well-known example is REFINE [194], part of the *Software Refinery Toolset* by Reasoning Systems. With versions for C, FORTRAN , COBOL and ADA [217], Software Refinery generated an AST from source code and stored them in a database for later searches. The AST was then queried and transformed using the REFINE language, which included syntax-directed pattern matching and compiled into COMMON LISP, with pre- and post-conditions for code transformations. This meta-development tool was used to generate development tools such as compilers, IDEs, tools for detecting violations of coding standards, and more.

Earlier efforts include *Gandalf* [127], which generated a development environment based on language specifications provided by the developers. The generated systems were extended using the ARL language, which was tree-oriented for easing AST manipulations. Other systems that generated database information from programs and allowed user-developed tools to query this data included the *C Information Abstractor* [54], where queries were expressed in the INFOVIEW language, and its younger sibling *C++ Information Abstractor* [123], which used the DATASHARE language.

A common theme of all of these, and numerous others (including systems such as *GENOA* [84], *TAWK* [125], *Ponder* [16], *ASTLog* [73], *SCRUPLE* [189] and more) is the AST-centered approach. In fact, AST-based tools became so abundant in this field that a recent such product was entitled *YAAB*, for "Yet Another AST Browser" [10]. Another category of products is contains those which rely on a relational model. For example, the *Rigi* [177] reverse engineering tool, which translates a program into a stream of triplets, where each triplet associates two program entities with some relation.

Section 4.5.1 compares JTL syntax with other similar products. Section 4.5.2 then says a few words on the comparison of relational- rather than an AST- model, for the task of queering object-oriented languages.

4.5.1 Using Existing Query Languages

Reading a poem in translation is like kissing
your lover through a handkerchief.

— H. N. Bialik

Many tools use existing languages for making queries. YAAB, for example, uses the Object Constraint Language, OCL, by Rational Software, to express queries on the AST; the *Software*

Life Cycle Support Environment (SLCSE) [208] is an environment-generating tool where queries are written in SQL; Rigi's triples representation is intended to be further translated into a relational format, which can be queried with languages such as SQL and PROLOG; etc.

BDDBDDB [229] is similar to JTL in that it uses DATALOG for analyzing software. It is different from JTL in that it concentrates on the specific objective of code optimization, e.g., escape analysis, and does not further abstract the underlying language.

A more modern system is XIRC [92], where program meta-data is stored in an XML format, and queries are expressed in XQUERY. The JAVA standard reflection package (as well as other bytecode analyzers, such as BCEL) generate JAVA data structures which can be manipulated directly by the language. JQuery [143] is a PROLOG-based extension of Eclipse that allows the user to make queries about code. Finally, ALPHA [187] promotes the use of PROLOG queries for expressing pointcuts in aspect-oriented programming.

We next compare queries made with some of these languages with the JTL equivalent. Figure 4.8(a) depicts an example (due to the designers of XIRC) of using XQUERY to find Enterprise JavaBeans (EJBs) which implement `finalize()`, in violation of the EJB specification.

```
subtypes(/class[@name="javax.ejb.EnterpriseBean"])
  /method[
       @name = "finalize"
       and .//returns/@type = "void"
       and not(.//parameter)
  ]
```

(a) XIRC implementation of the query (from [92]).

```
class implements /javax.ejb.EnterpriseBean {
   void finalize();
};
```

(b) The equivalent query in JTL.

Figure 4.8: Searching for EJBs that implement `finalize` with XIRC (a) and with JTL (b)

In inspecting the figure, we find that in order to use this language the programmer must be intimately familiar not only with the XQUERY language, but also with the details of the XIRC encoding, e.g., the names of attributes where entity names, return type, and parameters are stored. A tool developer may be expected to do this, probably after climbing a steep learning curve, but its seems infeasible to demand that an IDE user will interactively type a query of this sort to search for similar bugs.

The JTL equivalent (Figure 4.8(b)) is a bit shorter, and perhaps less foreign to the JAVA programmer.

Figure 4.8 demonstrates what we call *the abstraction gap*, which occurs when the syntax of the queries is foreign to the queried items. Word-processing and other office automation applications present no (or minimal) abstraction gap. For example, the search string which a user enters in the a typical text editor's search box is usually identical to the strings which it matches. The database users community is accustomed to Query-be-Example. Our quest is in fact a request to bring this ideal to the world of language processing tools.

we see that the JTL code is not only shorter to type and is probably easier to learn, but it is also *obvious to any JAVA programmer*, exhibiting a minimal abstraction gap.

We next compare JTL syntax with that of JQuery [143], which also relies on logic programming for making source code queries. Table 4.4 compares the queries used in the JQuery case study (extraction of the user interface of a chess program) with their JTL counterparts. The table

Task	JQuery	JTL
Finding class "BoardManager"	`class(?C,name,BoardManager)`	`class BoardManager`
Finding all "main" methods	`method(?M,name,main)` `method(?M,modifier,[public,static])`	`public static main(*)`
Finding all methods taking a parameter whose type contains the string "`image`"	`method(?M,paramType,?PT)` `method(?PT,/image/)`	`method(*,/?*image?*/,*)`

Table 4.4: Rewriting JQuery examples [143] in JTL

shows that JTL queries are a bit shorter and resemble the code better.

The JTL pattern in the last row in is explained by the following: To find a method in which one of the type of parameters contains a certain word, we do a pattern match on its argument list, allowing any number of arguments before and after the argument we seek. The desired argument type itself is a regular expression (replacing, as in all JTL regular expressions, the dot with the question mark as the "any character" signifier).

The ASPECTJ sub-language for pointcut definition, just as the sub-language used in JAM [8] for setting the requirements for the base class of a mixin, exhibit minimal abstraction gap. The challenge that JTL tries to meet is to do achieve this objective with a more general language.

Figure 4.9 is an example of using JAVA's reflection API to implement a query—here, finding all **`public final`** methods (in a given class) that return an **`int`**.

```
public List<Method> findPublicFinalInt(Class c) {
   List<Method> result = new Vector<Method>();
   for (Method m : c.getMethods()) {
      int mod = m.getModifiers();
      if (m.getReturnType() == Integer.Type
               && Modifiers.isPublic(mod)
               && Modifiers.isFinal(mod))
         result.add(m);
   }
   return result;
}
```

Figure 4.9: Locating **`public final int`** methods using the JAVA reflection API

Comparing the use of the reflaction API with the XIRC approach, we can observe several things:

- Figure 4.9 uses JAVA's familiar syntax, but this comes at the cost of replacing the declarative syntax of XIRC with explicit control flow.
- Despite the use of plain JAVA, Figure 4.9 manifests an abstraction gap, by which the pattern of matching an entity is very different from the entity itself.
- The code still assumes familiarity with an API.
- As with XIRC, it is unreasonable to expect an interactive user to type in such code.

Again, the JTL equivalent, **`public final int(*)`**, is concise, avoids complicated control flow, and minimizes the abstraction gap.

We should also note that the *fragility* of a query language is in direct proportion to the extent by which it exposes the structure of the underlying representation. Changes to the queried language (i.e., JAVA in our examples), or deepening the information extracted from it, might dictate a change to the representation, and consequently to existing client code. By relying on many JAVA keywords as part of its syntax, the fragility of JTL is minimal.

There are, however, certain limits to the similarity, the most striking one being the fact that in JTL, an absence of a keyword means that its value is unspecified, whereas in JAVA, the absence of e.g., **`static`** means that this attribute is off. This is expressed as **`!static`** in JTL.

Another interesting comparison with JTL is given by considering ALPHA and Gybels and Brichau's [126] "crosscut" language, since both these languages rely on the logic paradigm. Both languages were designed solely for making pointcut definitions (Gybels and Brichau's work, just as ours, assumes a static model, while ALPHA allows definitions based on execution history). It is no wonder that both are more expressive in this than the reference ASPECTJ implementation.

Unfortunately, in doing so, both languages *broaden* rather than narrow the abstraction gap of ASPECTJ. This is a result of the strict adherence to the PROLOG syntax, which is very different than that of JAVA. Second, both languages make heavy use of recursive calls, potentially with "cuts", to implement set operations. Third, both languages are fragile in the sense described above.

We argue that even though JTL is not specific to the aspect-oriented domain, it can do a better job at specifying pointcuts. (Admittedly, dynamic execution information is external to our scope.) Beyond the issues just mentioned, by using the fixed-point model of computation rather than backtracking, JTL solves some of the open issues related to the integration of the logic paradigm with object-orientation that Gybels, Brichau, and Wuyts mention [41, Sec. 5.2]: The JTL API supports multiple results and there is no backtracking to deal with.

4.5.2 AST vs. Relational Model

We believe that the terse expression and the small abstraction gap offered by JTL is due to three factors: (i) the logic programming paradigm, notorious for its brevity, (ii) the effort taken in making the logic programming syntax even more readable in JTL, and (iii) the selection of a relational rather than a tree data model.

We now try to explain better the third factor. Examining the list of tools enumerated early in this section we see that many of these rely on the *abstract syntax tree* metaphor. The reason that ASTs are so popular is that they follow the BNF form used to define languages in which software is written. ASTs proved useful for tasks such as compilation, translation and optimization; they are also attractive for discovering the architecture of structured programs, which are in essence ordered trees.

We next offer several points of comparison between an AST based representation and the set-based, relational approach represented by JTL and other such tools. Note however that as demonstrated by Gil and Maman [67, Sec. 3.1], and as Crew's ASTLOG language [73] clearly shows, logic programming does not stand in contradiction with a tree representation.

- *Unordered Set Support.* In traditional programming paradigms, the central kind of modules were procedures, which are sequential in nature. In contrast, in JAVA (and other object-oriented languages) a recurring metaphor is the unordered *set*, rather than the *sequence*: A program has a set of packages, and there is no specific ordering in these. Similarly, a package has a set of classes, a class is characterized by a set of attributes and has a set of members, each member in turn has a set of attributes, a method may throw a set of exceptions, etc.

Although sets can be supported by a tree structure, i.e., the set of nodes of a certain kind, some programming work is required for set manipulation which is a bit more *natural and intrinsic* to relational structures.

On the other hand, the list of method arguments is sequential. Although possible with a relational model, ordered lists are not as simple. This is why JTL augments its relational model with built-ins for dealing with lists, as discussed in Section 4.1.6.

- *Recursive Structure.* One of the primary advantages of an AST is its support for the recursive structures so typical of structured programming, as manifested e.g., in Nassi-Shneiderman diagrams [179], or simple expression trees.

  Similar recursion of program information is less common in modern languages. JAVA does support class nesting (which are represented using the `inners` predicate of JTL) and methods may (but rarely do) define of nested classes. Also, a class cannot contain packages, etc.

- *Representation Granularity.* Even though recursively defined expressions and control statements still make the bodies of methods in object-oriented programs, they are abstracted away by our model.

  JTL has native predicates for extracting the parameters of a method, its local variables, and the external variables and methods which it may access, and as shown, even support for dataflow analysis. In contrast, ASTs make it easier to examine the control structure. Also, with suitable AST representation, a LINT-like tool can provide warnings that JTL cannot, e.g., a non-traditional ordering of method modifiers.

  It should be said that the importance of analyzing method bodies in object-oriented software is not so great, particularly, since methods tend to be small [60], and in contrast with the procedural approach, their structure does not reveal much about software architecture [116]. Also, in the object-oriented world, tools are not so concerned with the algorithmic structure, and architecture is considered to be a graph rather than a tree [135].

- *Data Model Complexity.* An AST is characterized by a variety of kinds of nodes, corresponding to the variety of syntactical elements that a modern programming language offers. A considerable mental effort must be dedicated for understanding the recursive relationships between the different nodes, e.g., which nodes might be found as children or descendants of a given node, what are the possible parent types, etc.

  The underlying complexity of the AST prevents a placement of a straightforward interface at the disposal of the user, be it a programmatic interface (API), a text query interface, or other. For example, in the *Hammurapi* [25] system, the rule "*Avoid hiding inherited instance fields*" is implemented by more than 30 lines of JAVA code, including two **`while`** loops and several **`if`** clauses. The corresponding JTL pattern is so short it can be written in one line:

  ```
  class { field overrides[_] }
  ```

  The terse expression is achieved by the uniformity of the relational structure, and the fact that looping constructs are implicit in JTL queries.

- *Representation Flexibility.* A statically typed approach (as in Jamoos [111]) can support the reasoning required for tasks such as iteration, lookup and modification of an AST. Such an approach yields a large and complex collection of types of tree nodes. Conversely, in a weakly-typed approach (as in REFINE), the complexity of these issues is manifested directly in the code.

Either way, changes in the requirements of the analysis, when reflected in changes to the kind of information that an AST stores, often require re-implementation of existing code, multiplying the complex reasoning toll. This predicament is intrinsic to the AST structure, since the search algorithm must be prepared to deal with all possible kinds of tree nodes, with a potentially different behavior in different such nodes. Therefore, the introduction of a new kind of node has the potential of affecting all existing code.

In contrast, a relational model is typically widened by adding new relations, without adding to the basic set of simple types. Such changes are not likely to break, or even affect most existing queries.

- *Caching and Query Optimization.* There is a huge body of solid work on query optimization for relational structures; the research on optimizing tree queries, e.g., XPATH queries, has only begun in recent years. Also, in a tree structure, it is tempting to store summarizing, cached information at internal nodes—a practice which complicates the implementation. In comparison, the well established notion of *views* in database theory saves the manual and confusing work of caching.

4.6 A JTL Extension for Program Transformation

A predicate in logic programming languages such as DATALOG or PROLOG can include variables to be used for output, their value being set as part of the standard process of evaluation. This is also true for JTL predicates. Because JTL supports string variables (the **`STRING`** kind), it is possible to harness such output-only variables for the generation of string output. To do so, all one needs to do is introduce an additional parameter of this kind to every relevant predicate, and appropriate terms inside the predicate to bind the value of this variable as part of the pattern-matching process. However, the process of managing this "baggage" variable is tiresome and error prone.

This section describes a JTL language extension which automatically manages this baggage information. Section 4.7 will give a number of detailed applications of the mechanism, ranging from program transformation tools to an AOP language.

The principle behind the extension is simple: every JTL predicate implicitly carries with it an unbounded array of baggage anonymous **`STRING`** variables, which are computed by the predicate. These variables are output-only—an invocation of a predicate cannot specify an initial value for any of them. The compilation process translates into DATALOG only baggage which is actually used. Thus, the examples seen so far, as well as their DATALOG equivalent, are not changed, since no baggage variables were used.

In most output-producing applications, only the first baggage variable, called the *standard output* or just the *output* of the predicate, is used. The output parameter is also called the *return value* of a predicate, in the context of output generation or program transformation.

The description begins (Section 4.6.1) with the assumption that there is indeed only one such baggage. Implementation considerations are discussed in Section 4.6.2. Then, Section 4.6.3 explains how multiple baggage variables are managed, and Section 4.6.4 shows how escaping inside string literals can be used for producing more expressive output. A key feature of baggage processing—the iterative production of output with quantifiers—is the subject of Section 4.6.5.

4.6.1 Simple Baggage Management

The essence of baggage extension is that the output of a compound predicate is constructed by default from the component predicates. Since the initial purpose of output is to production of JAVA code, we have that the output of JTL predicates that are also JAVA keywords (e.g.,

`synchronized`) return their own name on a successful match. Other primitive predicates (e.g., `method`) return an empty string. Type and name patterns return the matching type or name. The fundamental principle is that whenever possible, any predicate returns the text of a JAVA code fragment which can be used for specifying the match.

The returned value of the conjunction of two predicates is the concatenation of the components. By default, this concatenation trims white spaces on both ends of the concatenated components, and then injects a single space between these. Disjunction of two predicates returns the string returned by the first predicate that is satisfied. Thus, for example, the predicate

```
public static [int | long] field 'old?*'
```

can be applied to some field called `oldValue`, in which case it will generate an output such as:

```
public static int oldValue
```

String literals are valid predicates in JTL, except that they always succeed. They return as output their own value. By using strings, predicates can generate output which is different from echoing their building blocks. For example, the pattern

```
class _ "extends Number"                                    (4.19)
```

generates, when applied to class `Complex`, the output

```
class Complex extends Number
```

The string literal in the pattern does not present any requirement to the tested program element, and the string result need not be an echo of that element. Pattern (4.19), for example, will successfully match class `String`, which does *not* extend `Number`.

String literals are just one example of what we call *tautologies*: predicates which hold for any value of their parameters. Tautologies are used solely for producing output. The most simple tautology is the predicate `nothing`, which returns the empty string, i.e.,

```
nothing := "";.
```

With this language extension, the JTL library was enriched with many such tautologies, e.g., `visibility` and `multiplicity`, defined as

```
visibility := public | private | protected | nothing;
multiplicity := static | nothing;
```

The negation operator, "`!`", discards any output generated by the expression it negates. For example, `!static` will generate an empty string when successfully matched. Thus `multiplicity` can also be defined as

```
multiplicity := static | !static;
```

Some of the other tautologies in the library include:

- `modifiers`, returning the string of all modifiers used in the definition JAVA code element;
- `signature`, returning the type, name, and parameters of methods, or just the type and name of fields;
- `header`, including both the modifiers and signature;

- `torso`, a primitive tautology returning the body (without the head and embracing curly brackets) of a method or a class, or the initialization value of a field; and
- `preliminaries`, returning the package declaration of a class.

Tautology `declaration`, whose baggage is the full definition of a program element, is very useful for exact replication of the matched element. It is defined thus:

```
declaration :=
     [type | method] preliminaries header "{" torso "}"
   | field preliminaries header "=" torso;
```

The following demonstrates how tautology `header` and several other auxiliary tautologies are defined:

```
header := modifiers declarator '?*' parents;
modifiers := concreteness strictness visibility...
concreteness := abstract | final | nothing;
strictness := strictfp | nothing;
...                                                          (4.20)
declarator := class | interface | enum | @interface;
parents := superclass optional_interfaces;
superclass := extends T ![T is Object] | nothing;
...
```

(The declaration of `optional_interfaces` is shown below, in Section 4.6.5.) The actual definitions are a bit more involved, since they have to account for annotations and generic parameters, and must have overloaded versions for elements of kind **`MEMBER`**.

4.6.2 Implementation Issues

Although JTL can be implemented on top of the source code, the current implementation operates on the compiled `.class` representation. Therefore, predicate `torso` requires a decompilation process.

Note that even if source code is used as input, the predicates above do not result in a direct copy of the input; they "re-generate" the source. For example, a JTL program operating on JAVA code such as

```
public String a, b = "Hello";
```

cannot hope to reproduce it verbatim, but rather the more canonical form:

```
public java.lang.String a = null;
public java.lang.String b = "Hello";
```

The output mechanism does not require the introduction of any side-effect to JTL. Rather, when compiling JTL predicates to DATALOG, we have that the string output is presented as an additional "hidden", or implicit parameter to DATALOG queries. This parameter is used for output only. For example, the JTL predicate

```
pa := public abstract;
```

compiles to DATALOG as:

```
pa(This,Result) :-  public(This,Result1),
                    abstract(This,Result2),
                    concatenate(Result1,Result2,Result).
```

Finally, the extension requires that, for disjunction, output will be generated only for the first matched branch. To this end, each branch of the disjunction is considered true only if all previous branches are false; i.e., a pattern such as:

```
p_or_a := public | abstract;
```

is compiled to:

```
p_or_a(This,Result):- public(This,Result).
p_or_a(This,Result):- !public(This,_), abstract(This,Result).
```

Note that the operation remains commutative with regard to the question of which program elements match it; the pattern **`abstract | public`** will match exactly the same set of elements as **`public | abstract`** will. Commutativity is eliminated only with regard to the string output, where it is undesired.

4.6.3 Multiple Baggage

It is sometimes desirable to suppress the output of one or more constituents of a pattern, even if they are not negated. This can be done by prepending the percent character, "%", to the expression. For example, the predicate

```
%public %static %final _ '?*';
```
(4.21)

will match any public static final element, but print only its type and name, without the modifiers that were tested for. Predicate (4.21) can also be written using square brackets, thus:

```
%[public static final] _ '?*';
```
(4.22)

The suppression syntax is in fact one facet of a more complex mechanism, which allows predicates to generate multiple string results, directed to different "output streams". By default, any string output becomes part of string result 1, which is normally mapped to the standard output stream (`stdout` in Unix jargon). Also defined are string result 0, which discards its own content (`/dev/null`), and string result 2, the standard error stream (`stderr`).

To direct an expression's string output to a specific string result, prepend a percent sign and the desired string result's number to the expression. A percent sign with no number, as used above in (4.21) and (4.22), defaults to `%0`, i.e., a discarded string result.

For example, consider the following predicate:

```
testClassName := %2[ class '[a-z]?*' "begins with a lowercase letter."];
```

If matched by a class, it will send output to string result 2, i.e., the standard error stream; possible output can be:

```
class yourClass begins with a lowercase letter.
```

If, however, the expression is not matched (in this case, because the class does not begin with a lowercase letter), no output is generated.

By using disjunction, we can present an alternative output for those classes that do not match the expression; for example:

```
testClassName :=
 %2[ class '[a-z]?*' "begins with a lowercase letter."]
 | [class '?*' "is properly named."];
```
(4.23)

Because it is not directed to any specific string result, the string result of the second part of the predicate is directed to the "standard output". The disjunction operator is evaluated in such a manner that its right-hand operand can generate output only if its left-hand one yielded false. Thus, the predicate `testClassName` will generate exactly one of two possible messages, to one of two possible output streams, when applied to a class.

A configuration file binds any string result generated by a JTL program to specific destinations (such as files). The default destinations of string results 0, 1 and 2 can therefore be overridden, and additional string destinations (unlimited in number) can also be defined.

JTL also includes mechanisms for redirecting the string result generated by a subexpression into a different string result in the calling expression, or even to bind string results to variables. The syntax %n>m p will redirect the string result of predicate p in output stream n into output stream m of the caller. For example, the expression

```
[%2>1 p1] | "failed"
```
(4.24)

will yield the string result that `p1` sends to output stream 2, in output stream 1. If `p1` fails, (4.24) will generate the output failed. However, if `p1` succeeds without generating any output in stream 2 (e.g., it generates no output at all, or output to other streams only) then (4.24) will generate no output.

To bind string results to a variable, the syntax %n>V p can be used, binding the output of predicate p in output stream n to variable V of the caller. (On the DATALOG level, this implies passing V as the implicit baggage variable when predicate p is invoked.)

4.6.4 String Literals

Baggage programming often uses string literal tautologies. Escaping in these for special characters is just as within JAVA string literals. For example, "\n" can be used to generate a newline character. An easier way to generate multi-line strings, however, is by enclosing them in a \[... \] pair.

When output is generated, a padding space is normally added between every pair of strings. However, if a plus sign is added directly in front (following) a string literal, no padding space will be added before (after) that string. For example, the predicate

```
class "New"+ '?*'
```

will generate the output

```
class NewList
```

when applied to the class `List`.

The character # has a special meaning inside JTL strings; it can be used to output the value of variables as part of the string. For example, the predicate

```
class "ChildOf"+ '?*' is T "extends #T"
```
(4.25)

will yield

```
class ChildOfInteger extends Integer
```

when applied to `Integer`. The first appearance of `T` in predicate (4.25) captures the name of the current class into the variable; its second appearance, inside the string, outputs its value.

Recall that both name and type patterns can be used to match JAVA types. However, when applied to types, a name pattern returns (as a string value) the short name of a class, whereas a type pattern returns the fully-qualified class name. We can therefore write (4.25) as

```
class "ChildOf"+ '?*' "extends"_
```

to obtain

```
class ChildOfInteger extends java.lang.Integer
```

The sharp character itself can be generated using a backslash, i.e., "\#". To output the value of # (the current receiver) in a string, just write "#". For example, the following binary tautology, when applied to an element of kind **`MEMBER`**, outputs the name of that element with the parameter prepended to it:

```
prepend[Prefix] := "#Prefix"+"#";
```

In case of ambiguity, the identifier following the # character can be enclosed in square brackets. More generally, # followed by square brackets can be used to access not only variables, but also output of other JTL expressions. For example, a string literal such as

```
"#[public|private]"                                    (4.26)
```

it will embed in the string the output of **`public|private`**. As another example, Figure 4.10 shows the definition of a tautology that returns a renamed declaration of a JAVA method or field. Renaming is achieved by prefixing a string to the current name of the element.

```
rename[Prefix] :=  modifiers prepend[Prefix] [
                     method (*) throwsList "{ #[torso] }"
                   |
                     field "= #[torso];"
                   ];
```

Figure 4.10: A predicate for renaming methods and fields by adding a prefix

Note that using JTL expressions inside a string implies that this literal is no longer a tautology, since it can now fail—e.g., for expression (4.26), in case the element at hand is neither public nor private.

4.6.5 Baggage Management in Queries

In the `rename` predicate example (Figure 4.10), the term `(*)` outputs the list of all parameters of a method. Set and list queries generate output like any other compound predicates. Different quantifiers used inside the generated scope generate output differently. In particular, **`one`** will generate the output of the set or list member that was successfully matched; **`exist`**, **`many`** and **`all`** will generate the output of every successfully matched member; and **`no`** generates no output. The extension introduces one additional quantifier, which is a tautology: writing

```
optional p;
```

in a quantification context prints the output of p, but *only* if p is matched.

For example, the following predicate will generate a list of all fields and methods in a class that were named in violation of the JAVA coding convention:

```
badlyNamedClassMembers := %class {
 [field|method] '[A-Z]?*' "is badly named.";
}
```
(4.27)

By default, the opening and closing characters (`()` or `{}`) print themselves; their output can be suppressed (or redirected) by prepending a `%` to each character.

Two pseudo-quantifiers, **`first`** and **`last`**, are in charge of producing output at the beginning or the end of the quantification process. The separator between the output for each matched member (as generated by the different quantifiers) is a newline character in set quantifiers, or a comma in the case of list quantifiers. This can be changed using another pseudo-quantifier, **`between`**. The tautology `optional_interfaces` used in the above definition of `header` (4.20) requires precisely this mechanism:

```
optional_interfaces :=  implements: %{
                           first "implements";
                           exists _; --and names of all super interfaces
                           between ","; --separated by a comma
                           last nothing; --and no ending text
                        %}
                        | nothing;
```

Since we use the **`exists`** quantifier, the entire predicate in the curly bracket fails if the class implements no interfaces—in which case the "**`first`**" string "implements" is not printed; if this is the case, then the last line of the definition ensures that the predicate remains a tautology, printing `nothing` if need be.

4.7 Applications of the Program Transformation Extension

This section shows how JTL's "baggage" extension can be used for various tasks of program transformation. The description ignores the details of input and output management; the implicit assumption is that the transformation is governed by a build-configuration tool such as Ant [132], which directs the output to a dedicated directory (without overriding the originals), orchestrates the compilation of the resulting source files, etc. This makes it possible to apply a JTL program in certain cases to replace an existing class, and in others, to add code to an existing software base.

We are interested here in the following applications of this extension:

1. *Program Transformation.* Perhaps the most obvious application of the string result is to generate JAVA code. With the extension described here, JTL becomes a general-purpose transformation language. Each JTL program has thus two components: the *guard* [85], which describes the pre-requisites for the transformation, and the *transformer*, which is the clockwork behind the production of output out of the matched code.

   We argue that JTL can be used for general code transformations, e.g., it is straightforward to write a small JTL program that, given an interface, generates a class boilerplate that implements this interface, including method bodies to be refined by a programmer. The converse, a JTL program that elicits the class protocol and generates an interface out of it, is also easy. Many other refactoring [102] tasks are readily implemented as JTL transformations.

   Further, we show that JTL can be used to implement genericity in JAVA which does not suffer from the restrictions due to erasure semantics and is as powerful as MIXGEN and

NEXTGEN, rivaling even the rich expressive power of C++ templates. We further show that JTL can be used for implementing mixins [38], much like what is done in the JAM system.

2. *Using JTL as an AOP Language.* The relationship between program transformation and aspects is a fascinating subject [80, 98, 103, 159, 162]. The community is aware of this relationship ever since the original introduction of AOP [151], which stated that "*some transformations are aspectual in nature*". To an extent, the application of aspects to code belongs in the decades-old quest for *disciplined* program transformation [156]. Given a piece of code, an aspect is an operator that modifies it through the injection of advice.

   We give examples showing that JTL can be used to produce JAVA code that augments the original code, or even replaces it entirely, and that this code production supports aspect orientation. While in some senses, the code is not as elegant as in "pure" aspect-oriented programming languages, JTL does introduce discipline into program transformations, including the definition of pointcuts to which advice are applied. And because the pointcuts themselves are defined in JTL's powerful query language, we find that this toy AOP language has some capabilities that are unmatched even by ASPECTJ.

   Finally, JTL can also be used as a rapid prototyping or implementation mechanism for new AOP languages.

3. *A Generative Language for Aspects.* The META-ASPECTJ project [138] pioneered the notion of applying generative programming [75] to aspects. The idea here is that a META-ASPECTJ program reads the input software and then generates, hand-tailors if you will, aspects that can process this code.

   We argue that JTL can be used in the same fashion, and can generate ASPECTJ code instead of JAVA; the ASPECTJ code in turn modifies the JAVA program as part of the weaving process.

4. *Translation.* The literature [224, 226] distinguishes between *rephrasing* transformations and *translating* transformations. Two of the examples presented above (an AOP language and refactoring) are cases of rephrasing, while the last example (using JTL as a generative language for aspects) is a case of translation. There are even more exciting translation applications of JTL. We show here a JTL program that, given a set of JAVA class definitions, produces SQL statements that define a database schema to store instances of these classes. In the same fashion, it is not difficult to generate an XML datatype definition out of a class structure. One can further use JTL to generate the complementary JAVA code that stores objects in such an XML format (marshalling), or conversely, reads a matching XML document and loads it into memory organized in this class structure (unmarshalling).

   Some other translation applications which JTL can handle include a LINT-like tool for detecting coding convention violations and potential bugs ("code smells") [100] and generating a report as output, or a documentation tool which may elicit a JavaDoc skeleton out of class signatures.

In a sense, these abilities are not surprising: software that can generate textual output, can, in principle, generate programs in any desired programming language. There are however important factors that make JTL better for this task. The production of output is an *implicit* component of the pattern matching process. Programmer intervention is required only where the defaults are not appropriate.

Section 4.7.1 briefly describes some ways in which JTL's transformations can be used in a JAVA IDE such as the eclipse system. A more interesting application is using JTL, as is, as an

AOP language, as described in Section 4.7.2. In this application, each aspect is written as a JTL predicate. Just like aspects, generic classes, and mixins, the predicates take a class parameter, and generate another class out of it. In Section 4.7.3 we show how mixins and templates are implemented with JTL. The translation subcategoy of program transformation is the subject of Section 4.7.4, which shows how JTL can generate an SQL scheme definition out of a JAVA class definition.

4.7.1 Using JTL in an IDE and for Refactoring

We have previously described (Section 4.4.1) the JTL Eclipse plug-in, which can be used for making quick searches in a software base. We have also shown how JTL can be used to detect programming errors and potential bugs. It should also be obvious how baggage output makes it possible for JTL to not only detect such problems, but also provide useful error and warning messages. Pattern (4.27) in the previous section shows an example.

JTL can also be put to use in some refactoring services supplied by the IDE. The following pattern extracts the public protocol of a given class, generating an **`interface`** that the class may then implement:

```
elicit_interface := %class ––Guard: applicable to classes only
   preliminaries "interface" prepend["I_"] ––Produce header
      { ––iterate over all members
         optional %public !static method header ";";
      };
```

We see in this example the recurring JTL programming idiom of having a *guard* which checks for the applicability of the transformation rule, and a *transformer* which is a tautology. (Note that by convention, the output of guards is suppressed, using the percent character.) The interface is generated by simply printing the header declaration of all public, non-static methods. Its name equals the class name, with "`I_`" prepended.

The converse IDE task is also not difficult. Given an interface, the following JTL code shows how, as done in Eclipse, a prototype implementation is generated:

```
defaultVal :=  ––the default value of any given type
                 %boolean "false"
               | %[double | float] "0.0"
               | %char "'\0'"
               | %primitive "0"––all other primitive types are integral
               | %void nothing ––void methods return nothing
               | "null"; ––all reference types
gen_class := %interface ––Guard
   preliminaries "class" prepend["C_"] "implements #" {
      header \[ {
         return #[defaultVal];
      } \]
   }
```

The above demonstrates how JTL can be used much like output-directed languages such as PHP [160] and ASP [173]. The output is defined by a multi-line string literal, into which, at selected points, results of evaluation are injected.

Another common IDE/refactoring job is the addition of standard code fragments to classes. For example, in JAVA, the `equals` method is important for proper usage of classes in, e.g., the collection framework, yet it proves nontrivial to implement [59]. The JTL pattern in Figure 4.11 adds

a proper implementation of `equals` to its argument class, without changing anything else. The

```
1 addEquals := %class header %[# is T] declares: {
2   ![boolean equals (Object)] declaration;

4   last \[
5     @Override boolean equals(Object obj) {
6         if (obj == null) return false; if (obj == this) return true;
7         if (!obj.getClass().equals(this.getClass())) return false;
8         #T that = (#T)obj; // downcast the parameter to the current type
9         #[T.compareFields] // invoke helper predicate for field comparison
10        return true; // all field comparisons succeeded
11    }
12  \];

14  compareFields := { -- generate field-comparison code
15    %[primitive field, '?*' is Name] -- guard for primitive fields
16    "if (this.#Name != that.#Name) return false;";

18    %[!primitive field, '?*' is Name] -- guard for reference fields
19    "if (!this.#Name.equals(that.#Name)) return false;";
20  }
21 }
```

Figure 4.11: A JTL transformation that generates a proper **`equals`** method

term **`%class`** (line 1) is used as the first guard, filtering non-class elements. Next, the `header` tautology outputs the class's header. The expression "`%[# is T]`" captures the implicit parameter into the variable `T`, thereby making it accessible inside the generated scope.

We then use `declares` as a generator, and iterate over all members defined in this class. Members that do not match the signature of `equals` (as presented in line 2) will be copied without change, using the `declaration` predicate. In effect, this filters out any existing definition of the method we are about to add. Then, using the pseudo-quantifier **`last`**, code for the `equals` method is added at the end of the class body. This code meets the strict contract prescribed for `equals` by the JAVA language specification.[2]

In line 9, the predicate invokes another predicate, `compareFields`, defined in lines 14–20. This definition is located inside `addEquals` itself, making it a *local definition*, accessible only from within the defining scope. Looking at `compareFields`, we find that it provides two different possible outputs for fields, each with its own guard: one for primitive fields and one for reference-type fields, using the appropriate comparison mechanism in each case.

4.7.2 JTL as a Lightweight AOP Language

With its built-in mechanism for iterating over class members, and generate JAVA source code as output, it is possible to use JTL as a quick-and-dirty AOP language. The following JTL predicate is in fact an "aspect" which generates a new version of its class parameter. This new version is enriched with a simple logging mechanism, attached to all public methods by means of a simple

[2] However, this code will not correctly handle recursive references. A more elaborate version, for handling such cases, is also possible.

"before" advice.

```
loggingAspect := %class header declares: {
   targetMethod := public !abstract method; --pointcut definition
   %targetMethod header \[ {
       System.out.println("Entering method #");
       #[torso]
   } \]
   | declaration;
}
```

The local predicate `targetMethod` defines the kinds of methods which may be subjected to aspect application—in other words, it is a guard serving as a pointcut definition. The condition in the existential quantifier is a tautology; therefore, output will be generated for every element in the set. The first branch in the tautology, its guard, is the term `%targetMethod`.

If the member is matched against the guard, the method's header is printed, followed by a *modified* version of the method body. If, however, the member does not match the `targetMethod` pointcut, the disjunction alternative `declaration` will be used—i.e., class members that are not concrete public methods will be copied unchanged.

Having seen the basic building blocks used for applying aspects using JTL, we can now try to improve our logging aspect. For example, we can change the logging aspect so that it prints the actual arguments as well, in addition to the invoked method's name. To do so, we define the following tautology:

```
actualsAsString :=  %(
                        first \["(" + \];
                        last \[+ ")" \];
                        between \[+ ", " + \];
                        argName; --at least one; iterate as needed
                    %)
                    | "()"; --no arguments
```

Given a method signature with arguments list ($\texttt{type}_1\ \texttt{name}_1, \ldots \texttt{type}_n\ \texttt{name}_n$), this predicate will generate the output

"(" + name_1 + ", " + ... + name_n + ")"

which is exactly what we need to print the actual parameter values. The code generated is specific per method to which the advice is applied. Any attempt to implement an equivalent aspect with ASPECTJ requires the usage of runtime reflection in order to iterate over each actual parameter in a method-independent manner.

JTL AOP can be used to define not only **`before`**, but also **`around`**, **`after`**, **`after returning`** or **`after throwing`** advice. This is done by renaming the original method, and creating a new version which embeds a call to the original. Figure 4.12 introduces a version of the logging aspect which also reports returned values (or thrown exceptions). The predicate in the figure preforms two "iterations" over the members declared in a class. In the first iteration (lines 5–6), methods which match the pointcut (defined in line 2) are renamed. The second iteration (lines 9–30) regenerates matching methods with a new body that calls their renamed version, while adding the appropriate logging instructions.

It is interesting to see how guards and transformations are nested in the second iteration. At first, the phrase `%targetMethod header` has two components: the guard, which lets through

```
1  loggingAspect2 := %class header declares: {
2   targetMethod := public !abstract method; −− pointcut definition

4   −− rename matching methods:
5    %targetMethod rename["original_"];
6   | declaration; −− other elements copied unchanged.

8   −− And now, reiterate over matching methods:
9    %targetMethod header "{" −− regenerate header and ‘‘{’’
10   [ −− Regenerate torso, differently for void and non−void methods
11     %[ !void, _ is Return ] −− Guard for non−void methods
12       \[     System.out.println("Entering #" + #[actualsAsString]);
13              try { // Invoke the renamed original:
14                #Retrun result = #[prepend["original_"]] #[argNames];
15              } catch (Throwable e) {
16                System.out.println("Exception: " + e); throw e;
17              }
18              System.out.println("Returned " + result);
19              return result;
20       \]
21    | −− deal with void methods
22       \[     System.out.println("Entering #Name" + #[actualsAsString]);
23              try { // Invoke the renamed original:
24                #[prepend["original_"]] #[argNames];
25              } catch (Throwable e) {
26                System.out.println("Exception: " + e); throw e;
27              }
28       \]
29   ]
30   "}"; −− generate closing ‘‘}’’
31 }
```

Figure 4.12: A logging aspect defined as a JTL predicate

only members that match the pointcut, and a tautology which regenerates the header of matching items.

Subsequently, we see the compound predicate in lines 10–29, which is in charge of printing the method's new torso. This disjunction predicate has two parts: in the first (lines 11–20), a guard is applied to produce output for non-void methods; the second component (lines 22–28) comes into action if the first fails, and produces output solely for void methods. In the AOP terminology, we see that pointcuts and advices can be intermixed and nested.

Note how the arguments list is copied to create the method invocation, using `#[argNames]`. This tautology (for methods or constructors) is defined thus:

```
argNames := ( optional argName );
```

Because the default separator in list generators is a comma, the result is formatted exactly as we need it.

Unlike ASPECTJ aspects, which use `Object` references to capture return values in a type-independent manner and must therefore rely on the boxing and unboxing of primitive values, the aspect presented above involves no boxing of primitives. For example, if the method being

processed is of type **int**, then the generated replacement method will include a local variable `result` of the primitive type **int**.

The main limitation of writing aspects in JTL is that we have no way to traverse and modify the internals of method bodies. The JTL AOP language is therefore limited to **execution** pointcuts only. In particular, advice that should be applied to each access to a variable (**get** and **set** pointcuts), or advice that should be applied to exception catch blocks, etc., cannot be created with JTL. This limitation is not unique to JTL, however; several other AOP solutions take the same approach, some (such as the Spring framework) by a well-reasoned, explicit choice. Thus, JTL AOP can be classified as *black-box AOP* using Filman and Friedman's [97] terminology.

In the examples above, the generated class has the same name as the original class, i.e., weaving-by-replacement is used. However, JTL can just as easily be used for weaving-by-subclassing, and in particular can be used for implementing shakeins. Indeed, this is but one example of how JTL can be used to implement other AOP languages; Czarnecki and Eisenecker [75] explain that the "striking similarity" between code transformations aspects stems from the fact that both "may look for some specific code patterns in order to influence their semantics," and "[f]or this reason, transformations represent a suitable implementation technology for aspects or aspect languages".

The following section discusses additional uses for JTL, outside of AOP, that can be reached by replacing, augmenting, or subclassing existing classes.

4.7.3 Templates, Mixins and Generics

Since JTL can generate code based on a given JAVA type (or list of types), it can be easily used to implement generic types. A simple example is provided by the `singleton` predicate (Figure 4.13), which is a generic that generates a SINGLETON class [105] from a given base class.

```
singleton := "public" class "Singleton"+ '?*', %[_ is T] {
   %[ public constructor() ]
   | %2 "#T has no public zero−args constructor.";

   last \[
      private #T() { } // No public constructor

      private static #T instance = null;

      public static #T getInstance() {
         if (instance == null)
            instance = new #T();

         return instance;
      }
   \];
}
```

Figure 4.13: A JTL "generic" that generates SINGLETON classes

Given class, e.g., `Connection`, the predicate shown in the figure will generate a new class, `SingletonConnection`, with the regular singleton interface.[3] This seemingly trivial example

[3]It is a relatively simple change to make this JTL pattern generate a class that *replaces* the source class, rather than

cannot be implemented using JAVA's generics, because those rely on erasure [39]. It takes the power of NEXTGEN, with it's first-class genericity, to define such a generic type.

The JTL pattern is also superior to the C++ template approach, because the requirements presented by the class (its *concept* of the parameter) are expressed explicitly. The lack of concept specification mechanism is an acknowledged limitation of the C++ template mechanism [210, 211]. With the JTL example in Figure 4.13, in case the provided type argument does not include an appropriate constructor (i.e., does not match the concept), a straightforward error message is printed to `stderr`. This will be appreciated by anyone who had to cope with the error messages generated by C++ templates [210].

Because the generic parameter does not undergo erasure, JTL can also be used to define mixins [38]. Figure 4.14 is an example that implements the classic mixin `Undo` [8]. Here, too, the

```
undoMixin := "public" class "Undoable#T" extends T {
    %[ !private void setName(String) ]
    | %2 "#T has no matching setName method.";

    %[ !private String getName() ]
    | %2 "#T has no matching getName method.";

    all ![ !private undo() ]
          | %2 "Conflict with existing undo method.";

    last \[
        private String oldName;

        public void undo() {
            setName(oldName);
        }

        public void setName(String name) {
            oldName = getName();
            super.setName(name);
        }
    \];
}
```

Figure 4.14: The Undo mixin as a JTL pattern

pattern explicitly specifies its expectations from the type argument—including not only a list of those members that must be included, but also a list of members that must *not* be included (to prevent accidental overriding [8]).

4.7.4 Non-JAVA Output

There is nothing inherent in JTL that forces the generated output to be JAVA source code. Indeed, some of the most innovative uses generate non-JAVA textual output by applying JTL programs to JAVA code. This section presents one such example in detail.

A classic nonfunctional concern used in aspect-oriented systems is persistence, i.e., updating a class so that it can store instances of itself in a relational database, or load instances from it. In

generate an additional one.

most modern systems (such as Hibernate [22] and version 5 of JAVA EE), the mapping between JAVA classes and database tables is defined using annotations. For example, Figure 4.15 are two classes, each mapped to a different database table, with a foreign key relationship between them: In this simplified example, the annotation `@Table` marks a class as persistent, i.e., mapped to a

```
@Table class Account {
   @Id @Column long id; // Primary key
   @Column float balance;
   @ForeignKey
    @Column(name="OWNER_ID") Person owner;
}

@Table(name="OWNER") class Person {
   @Id @Column long id;
   @NotNull @Column String firstName;
   @NotNull @Column String lastName;
}
```

Figure 4.15: Two JAVA classes with annotations that detail their persistence mapping

database table. If the `name` element is not specified, the table name defaults to the class name. Similarly, the annotation `@Column` marks a persisted field; the column name defaults to the field's name, unless the `name` element is used to specify otherwise. The special annotation `@Id` is used to mark the primary-key column.

Given classes annotated in such a manner, we can use the `generateDDL` JTL program (Figure 4.16) to generate SQL DDL (Data Definition Langauge) statements, which can then be used to create a matching database schema. Using the **`first`**, **`last`**, and **`between`** directives, this query generates a comma-separated list of items, one per field in the class, enclosed in parenthesis. The program also includes error checking, e.g., to detect fields with no known matching SQL column type.

When applied to the two classes from Figure 4.15, `generateDDL` creates the output shown in Figure 4.17.

In much the same way, JTL can be used to generate an XML Schema [96] or DTD [40] specification, describing an XML file format that matches the structure of a given JAVA class. This is the reverse of the operation performed by the JAXB compiler [186], which generates class files given Schema or DTD definitions.

4.8 Related Work on Program Transformation

The work on program transformations is predated to at least D. E. Knuth's call for "program-manipulation systems" in his famous "Structured programming with go to statements" paper [156]. Shortly afterwards, Balzer, Goldman and Wile [19] presented the concept of *transformational implementation*, where an abstract program specification is converted into an optimized, concrete program by successive transformations.

By convention, program transformation in JTL has two components: *guards* (similar to a "pointcut" in the AOP terminology), which are logical predicates for deciding the applicability of a *transformer* (similar to "advices" of the jargon), which is a tautology predicate in charge of output production. The examples included in the previous section show that JTL is an expressive

```
generateDDL := %class "CREATE TABLE " tableName %{
   first "("; last ")"; between ",";

   %[ @Column field ] =>
      %sqlType
      | %2 ["Unsupported field type, field" '?*'];

   columnName sqlType sqlConstraints;
%}

sqlType := --simplified version
       %String "VARCHAR"
     | %integral "INTEGER"
     | %real "FLOAT"
     | %boolean "ENUM('Y','N')"
     | %BigDecimal "DECIMAL(32,2)"
     | %Date "DATE"
     | foreignKey;

sqlConstraints :=
    [ %@NotNull "NOT NULL" | nothing ]
    [ %@Id "PRIMARY KEY" | nothing ]
    [ %@Unique "UNIQUE" | nothing ];

foreignKey := %[ _ is FK ] --target class
    "FOREIGN KEY REFERENCES" FK.tableName;

tableName :=
     [ %@Table(name=TName:STRING) "#TName" ]
  | [ %@Table() '?*' ] --Default table name = class name
  | %2 "Class is not mapped to DB table.";

columnName :=
     [ %@Column(value=CName:STRING) "#CName" ]
  | [ %@Column() '?*' ]; --Default column name = field name
```

Figure 4.16: Patterns for generating SQL DDL statements for annotated persistent JAVA classes

tool for such output production—the transformation, or the process of aspect application, is syntax directed, much like syntax-directed code generation in compiler technology [2].

The JTL system can be categorized using Wijngaarden and Visser's taxonomy of transformation systems [224], consisting of three dimensions:

1. *Scope* pertains to the extent of the portion of an object program covered by a single transformation step, which can range from a single instruction to an entire program. The examples given here, including the application of JTL as an AOP language, are *local-local* transformations, since both input and output do not consult global information. This situation is typical to JTL programs, as a result of the the syntax-directed translation engine, but it is possible at least in principle to write programs with more global scope for either input or

```
CREATE TABLE Account (
   id INTEGER PRIMARY KEY,
   balance FLOAT,
   OWNER_ID FOREIGN KEY REFERENCES OWNER,
);
CREATE TABLE OWNER (
   id INTEGER PRIMARY KEY,
   firstName VARCHAR NOT NULL,
   lastName VARCHAR NOT NULL,
);
```

Figure 4.17: The DDL statements generated by applying the `generateDDL` pattern (Figure 4.16) to the classes from Figure 4.15 (shown pretty-printed for easier reading)

output. As a minor example, the SQL DDL generation program examines the annotations attached to classes other than the input class—as directed by the types of fields marked as foreign keys.

2. The *direction* of a transformation is either forward (source driven) or reverse (target driven). JTL is primarily source driven, in that the input structure orchestrates the generation of output. The reverse translation mode is one in which, just as being done in the ASP and PHP languages, the output (normally HTML text in these two languages) is a template, with placeholders ready to be filled by functions of the input. As some of the examples indicate, reverse direction transformation is possible in JTL. This is achieved by means of the ability to embed the output of predicates inside string literals.

3. Different transformation engines use a different number of *stages*. Wijngaarden and Visser make the distinction between *single-stage*, *multi-stage modify* and *multi-stage generate* techniques. JTL applications are a single-stage approach, since the target is generated in one single traversal over the source. It is future research to evaluate the benefits of using JTL in a multi-stage-generate approach, where every traversal generates a piece of output which is then merged to create the final output. A more interesting direction for future research is the multi-stage-modify approach, by which the target is generated incrementally by making several traversals over the source, which corresponds (if JTL is used as an AOP language) to famous questions of aspect interferences, priority, etc.

In as sense, our paper sides with the perspective by which aspects are thought of as transformations of a software base; aspect application is a transformation of the rephrasing kind, which also includes inlining, specialization, and refactoring. This perspective was presented earlier by Fradet and Südholt [103], whose work focused on "aspects which can be described as static, source-to-source program transformations". It was in fact one of the earliest attempts to answer the question, "what exactly *are* aspects?". Unlike JTL, the framework presented by Fradet and Südholt utilizes AST-based transformations, thereby offering a richer set of possible join-points, enabling the manipulation of method internals.

Lämmel [159] also represents aspects as program transformations, whereas the developers of LOGICAJ [195] go as far as claiming that "[t]he feature set of ASPECTJ can be completely mapped to a set of conditional program transformations".[4] LOGICAJ uses program transformations as a foundation for AOP, and in particular for extending standard AOP with generic aspects. More

[4] http://roots.iai.uni-bonn.de/research/tailor/aop

recently, Lopez-Herrejon, Batory and Lengauer [162] developed an algebraic model that relates aspects to program transformations (and used this model to explain several of the known problems of traditional AOP).

JTL is not the first system to use logic-based program transformation for weaving aspects. Indeed, De Volder and D'Hondt's [80] coin the term *aspect-oriented logic meta programming* (AOLMP) to describe logic programs that reason about aspect declarations. The system they present is based on TyRuBa [79], a simplified variant of Prolog with special devices for manipulating Java code. However, whereas JTL presents an open-ended and untamed system for manipulating Java code, De Volder and D'Hondt's system presents a very orderly alternative, where output generated not by free-form strings but rather using quoted code blocks. It is significantly easier to define and reason about aspects using TyRuBa.

We therefore find that, compared to other AOP-by-transformation systems, JTL is limited in the kind of transformations it can apply for weaving aspects, and the level of reasoning about aspects that it provides—which is why we view it as a "quick-and-dirty" AOP language. The windfall, however, is that program transformation in JTL is not limited to AOP alone, as evident from some of the examples provided in this paper—the generation of stub classes from interfaces, the generation of SQL DDL to match classes, the definition of generic classes, etc.

4.8.1 Output Validation

The trust we can put in any code-generation mechanism can be increased by the assurance that it will always generate valid code in the target language. Unfortunately, for arbitrary grammars G_1 and G_2, it is undecidable to determine whether $L(G_1)$ (i.e., the language defined by a grammar G_1) is a subset of $L(G_2)$ (see e.g., [204, p. 172]).

We must therefore infer that JTL is not "type safe", in the sense that a JTL program may generate output that is not a valid program in the target language, and no mechanism exists for proving the correctness of arbitrary JTL programs—or indeed, arbitrary code generation mechanisms of any kind; even the task of proving that a plain procedural program produces correct SQL is known to be difficult [57, 168] and, in its most general form, undecidable. However, given a *particular* JTL program p, there is still hope for proving that p always generates valid programs in a *specific* target language.

The problem of proving that a code-generating program p always generates valid code is in fact twofold:

(a) Finding the grammar G_p that describes all possible output texts, and

(b) Proving that $L(G_p) \subseteq L(G)$, where G is the grammar of the target language.

There are known solutions for some particular cases, e.g., Minamide and Tozawa [176] have shown practical algorithms for proving that PHP programs generate valid XHTML output.

Whereas in imperative programs the first step (finding the output grammar) is necessarily an approximation, the declarative nature of JTL renders the first step trivial: in JTL, *the program itself* reads as a context-free grammar of all possible outputs, i.e., $G_p \triangleq p$. For example, the predicate

```
[public | protected] static
```

has two possible outputs, public static or protected static.

This simplification of the output validation problem is *a key benefit* of the declarative approach. It also implies that human programmers will find it significantly easier to review the transformation program and verify that only valid code in the target language can result, compared to the same verification task given an imperative program transformation mechanism.

Previous results [176] show that it is possible (and practical) to decide if $L(G) \subseteq L(G_{\text{xml}})$, where G is a context-free grammar and G_{xml} is an XML grammar defined, e.g., using a DTD specification. It is therefore possible and practical to decide if a given JTL program generates valid XML for any given XML grammar.

The problem is more difficult, however, for target languages such of SQL or JAVA. Perhaps type safety in this case can be achieved by augmenting JTL with an auxiliary type system, which reflects the type system of the parse tree of the host language, so the process of type checking of predicate definitions in JTL will also prove that the output is correct—much as was done in the evolution of GOTECH into type-safe META-ASPECTJ. Still, even if the problem is decidable, with high-level languages, meeting the language grammar does not necessarily imply that the program is valid, since certain semantic demands must also be met (e.g., in JAVA, no two local variables in the same innermost scope can have the same name; this is not reflected in the language's CFG).

Compared to JTL, ASPECTJ and other AOP languages are type-safe in the sense that it is guaranteed that an application of an aspect to a syntactically correct JAVA program will yield a program which is still executable. Deep inside, this type safety is achieved by a general "proof system" (so to speak) that must automatically determine, for any predicates t and g, whether t follows from g, where t denotes the demands and assumptions that an advice makes of the advised code, and g the demands that the pointcut definition makes of the same code. Clearly, the difficulty of writing such a proof system increases with the expressive power of the language in which t and g are written. In the case that t and g use the full power of first order predicate logic, then the problem becomes undecidable, and it remains so even if we restrict the expressive power to that of context-free languages. "Type safety" in ASPECTJ is achieved by minimizing the expressiveness of both g and t; complex situations, e.g., iteration over parameters of advised methods, are deferred to runtime and must be implemented by JAVA code (*cf.* the JTL aspect in Figure 4.12, which performs this iteration ahead-of-time).

Unlike ASPECTJ code, code generated by a JTL program is never executed directly; it must first be compiled by the target language's compiler. When JTL is used as a JAVA-to-JAVA transformation mechanism, and in particular as an AOP language, its output is always processed by a JAVA compiler. If, in certain situations, the JTL program's output is illegal JAVA code, then these errors are still detected at compile time. We therefore conclude that the lack of output type safety in JTL will never lead to runtime errors due to misuse of types.

4.9 Summary

JTL is a novel, DATALOG-based query language designed for querying JAVA programs in binary format. It can also by extended to support program transformation, without breaking the logic programming paradigm.

Additional JTL-related research. Work by Maman [67, Sec. 6] shows that JTL, even when implemented as an interpreter, outperforms comparable solutions significantly, both in terms of runtime performance and in the size of the database that can be effectively searched.

As mentioned earlier, JTL has been developed with the JAVA language in mind. Nonetheless, our theoretical observations, as well as many of the implementation-level concerns, are applicable to other object-oriented programming languages. Additional work [66] shows how JTL's concepts can be ported to other programming languages. In particular, it was shown that JTL can be ported to C$^{\#}$ (including support for program elements that are alien to JAVA, such as delegates) simply by replacing the native predicates in JTL's standard library.

A paper by S. Cohen, Gil and Zarivach [58] investigates the termination question in JTL's underlying DATALOG model.

Using JTL with Shakeins. With regard to the key subject of this work, we believe that JTL can serve well for both of its originally stated purposes: expressing rich pointcuts for use with shakeins (and with AOP in general), and expressing concepts for restricting the application of shakeins to matched classes (and concepts for generic programming in general).

With the introduction of JTL as a pointcut- and concept-specification language, we believe our description of the shakein mechanism is complete. The following two chapters present language mechanism that can co-exist alongside shakeins, making the use of shakeins more natural as well as more powerful. Interestingly, both suggested mechanisms—factories (Chapter 5) and object evolution (Chapter 6)—prove to be of use and of interest in their own right, although their usefulness is magnified in the presence of the shakein language construct.

Chapter 5

Factories

The factory-owning class controls the means of production.

— Karl Marx, *Das Kapital*

The ASPECTJ2EE experiment indicates that the shakein semantics integrates well with current J2EE server architectures. However, the same experiment also led us to the conclusion that program modules should be provided with a better mechanism for controlling object instantiation. We therefore present *factories* as a new language construct, providing each class with complete control over its own instance-creation process.

But the motivation for factories is deeper than merely extending language support for enterprise applications; we argue that there are good programming-language theoretical reasons for adding the factories extension to all object-oriented programming languages. Good programming languages support, at the language level, the general principle of hiding implementation details from the client [188]. Indeed, most contemporary object-oriented programming languages let, sometime even force, the programmer to hide the implementation details of methods that a class offers. An inspiring case in point is Meyer's *principle of uniform access* [169, p.57], stating that

> *"All services offered by a module* [i.e., a class] *should be available through a uniform notation, which does not betray whether they are implemented through storage or through computation."*

However we observe that, despite the progress in language design, the *constructors* in object-oriented languages (called *creation procedures* in some of these) still reveal more than they should of their implementation secrets. Constructors are distinguished from the other services that a class may offer in that the client cannot apply them to a polymorphic object; instead the client is responsible for creating such an object, and therefore must know the precise name of the class that creates it.

The polymorphic nature of classes is advertised as means for separating interface from implementation. Object-oriented polymorphism means that a client may use instances of different subclasses to implement the same protocol. But the trouble is that, in order to be able to use such instances, one needs to create them somewhere, and the creation process is coupled with the name of the creating class. Breaking this coupling seems to be an intriguing chicken and egg riddle: Interface (or protocol) can be separated from implementation, but in order to select a particular implementation of a given protocol one must be familiar with at least one of these implementations.

Our solution to this cyclic dilemma is by making the selection of an implementation part of the interface. In the object-oriented terminology, this means that we allow a class to offer a set of

services, what we call *factories*, for generating instances of its various subclasses. Factories are first-class class members (alongside methods and constructors), but, unlike constructors, factories encapsulate instance management decisions without affecting the class's clients. Our contribution includes also a re-implementation of the JAVA compiler that supports factories; this implementation requires no changes to the JVM.

Factories directly attack the *change advertising problem*: Suppose that the implementation of a class (indeed, the internals of any software unit) is changed or specialized, but, as is the case with inheritance or shakeins, that the original version still remains. Then, the fact that there was a change must be advertised to the clients that wish to enjoy its benefits. Specifically, an instance of a class C' inheriting from C can be used anywhere an instance of C is used; but clients must be aware that C' exists, and be familiar with its name and its particular repertoire of constructors, in order to create such instances.

Existing solutions to the change advertising dilemma can be found in several popular frameworks, which act outside of the programming language. This includes, for example, the J2EE mechanism for obtaining instances of Enterprise JavaBeans. Clients must not directly invoke constructors for EJBs; rather, special methods of "home objects" [202, Chap. 5] must be used, effectively encapsulating the creation process and providing the platform with the ability to decide an instance of which (sub)class will be generated. The same is true for clients in the ASPECTJ2EE framework, as presented in Chapter 3.

Likewise, users of the Spring Application Framework should only obtain instances (of any class) by using special "bean factory" objects. The need for factories is further evident from the popularity and usefulness of design patterns that strive to emulate their functionality, including ABSTRACT FACTORY, FACTORY METHOD, SINGLETON [105], and OBJECT POOL [122]. However, both the frameworks and the design patterns introduce certain restrictions that the developers must adhere to (such as never invoking constructors directly). Just like these design patterns, factories are not compelled to return a *new* class instance. In not betraying the secret whether a new instance was generated or an existing one was fetched, they can be thought as applying the principle of uniform notation to instantiation. Much as with uniform access for "features" (attributes or functions) in EIFFEL, factories prevent upheaval in client classes whenever an internal implementation decision of the class is changed.

More concretely, we describe the design and implementation of an extension to the JAVA programming language to support factories. In this extension, factories act as methods that overload the **new** operator. But, unlike **new** overloading in C++, factories are not concerned with memory allocation but rather with instance creation and specific subclass selection decisions. We offer two varieties of factories:

- *Client-side factories* help localize instantiation statements, whereby a re-implementation can be selectively injected to certain clients.

- *Supplier-side factories* provide classes with fine control over their instantiation, and help in a global advertising of a change in the implementation.

Factories enable the encapsulated implementation of the "creational" design patterns listed above, either for all clients (using supplier-side factories) or for specific ones (using client-side factories). They provide a language-level solution to the change advertising dilemma, without presenting developers with any restrictions or complications. And they do away with the need for specialized mechanisms for obtaining instances of shakein-enhanced beans in ASPECTJ2EE.

Chapter outline. Section 5.1 starts by setting forth a common terminology for the discussion, and tries to unify some of the different perspectives offered in the literature to the class concept.

Using this terminology, Sections 5.2 and 5.3 expand on the motivation, by highlighting certain limitations of constructors. Factories are the subject of Section 5.4, which describes their JAVA syntax and some of the applications. This section also shows how factories support many classical design patterns. Section 5.5 describes how coupling between classes can be decreased using factories, and Section 5.6 describes the notion of client-side factories. Finally, Section 5.7 discusses the extension of the factories idea to other programming languages and concludes.

5.1 Terminology

There are many ways in which people perceive the notion of class: as a "*repository for* behavior *associated with an object*" [44, p.13], a "*unit of software decomposition*" and a "*type"* [169, pp.170–171], a "*tool for creating new types*" [209, p.223], a "*group* [of objects]" [142, p.50][1], a "*set of objects that share a common structure and a common behavior*" [35, p.93], etc. This section tries to unify these perspectives and propose a terminology (a conceptual framework if you will) for comparing and understanding the notion of a class in different programming languages.

We distinguish five, not entirely orthogonal, dimensions of class analysis: *purpose*, *commonality*, *encapsulation*, *morphability*, and *binding*. The most interesting dimension is *purpose*, already presented in Section 2.1.1, by which we identify, for each syntactical element of a class, a programming-language purpose. In Section 5.2 we shall argue that, judged by these dimension of evaluation, constructors make a bit of weird bird. Let us now describe in greater detail each of the five dimensions in turn.

1. **Purpose.** As detailed in Section 2.1.1, we characterize the constituents of a class into five different purposes: Two interface purposes, the *forge* and the *type*, and three materialization purposes, the *implementation*, the *mill*, and the *mold*.

   Specifically, the *forge* of the class is the collection of operations that can be used to create objects; the *type* is the set of messages that these instances may receive, along with their visibility specification; and the *implementation* is the body of the methods executed in response to these messages. The *mold* is the memory layout which instances of this class follow; it consists solely of field definitions. The *mill* is the set of constructor bodies.

2. **Commonality.** This dimension makes the distinction between *common* elements of the class notion (e.g., class variables and methods in SMALLTALK [113]) and *particular* such elements (e.g., instance variables and methods). More precisely, an element is common if its incarnation in different instances of the class is identical; otherwise, it is particular. Thus, particular elements may be used only in association with a specific class instance. Also, common elements cannot access particular elements.

3. **Encapsulation** (also known as *Visibility*.) A class may encapsulate (i.e., set the visibility of) its elements. C++'s three visibility levels, just as JAVA's four, are orthogonal to commonality.

4. **Morphability.** Morphability indicates the class element's ability to obtain a shape, or be re-shaped, in a subclass. In other words, morphability pertains to the kind of changes that a subclass may apply to components of the base class in the course of inheritance.

   There is a great variety in the morphability capabilities in different programming languages. For example, C++ allows a subclass to decrease the visibility of inherited members, OBERON [230] forbids overriding, JAVA sports `final` members and allows data members

[1] But also a "*template for several objects* . . . [a description of] *how these objects are structured internally.*"

to be hidden [117, Sect. 8.3.3], while EIFFEL allows re-implementation of a data member as a method, and method renaming. The analysis of this variety in full is beyond the scope of this work.

5. **Binding.** As the name suggest, in this dimension we make the distinction between statically-bound and dynamically-bound elements. Of course, this distinction can be made only for class elements which can be replaced or altered in a subclass. Non-**`virtual`** methods in C++ are famous for being statically bound.

   Observe that in most languages, commonality and binding are not orthogonal. Specifically, we find that *common elements are often statically bound*. The linkage between static binding and commonality is so entrenched that common methods and fields in languages such as JAVA, C# and C++ are marked with the **`static`** keyword.

   The phenomena can be explained by the reliance of dynamic binding on dispatching information associated with individual objects. Common elements are statically bound since they may exist even when there are no instances to the class.

5.2 Constructor Anomalies

Factories, the language extension proposed in this Chapter, are methods which return new class instances. Syntactically, a factory is a method which overloads the **`new`** operator with respect to a certain class.

In the terminology of the previous section, the signature of a factory belongs in the forge, while its body belongs in the mill. In this respect, factories are similar to constructors in mainstream object-oriented languages, the means by which a class' clients may obtain instances.

In analyzing constructors (in, e.g., JAVA or C++) with this terminology, we find that they exhibit three fundamental anomalies, which underline the need for the alternative approach that factories offer:

1. **Commonality.** In JAVA, the syntax for creating an instance of class `MyClass` is **`new`** `MyClass()`, i.e., it refers to the class name. In contrast, in EIFFEL the syntax is `!!myInstance`, i.e., referring to a variable. The difference between the languages is not a coincidence. Constructors are anomalous in that they are *simultaneously common and particular*: common—since they are invocable without an instance; particular—since they work on an object.

   This anomaly raises the dilemma of method binding inside constructor bodies. Method invocation from the mill follows a static binding scheme in C++ (even for **`virtual`** methods). in JAVA and C#, however, dynamic binding is used. Neither approach is without fault. Static binding can lead to illegal invocation of pure virtual methods. Dynamic binding prevents methods, invoked from within the mill, from assuming that all fields were properly initialized. Dynamic method binding in constructors leads, among other things, to difficulties in implementing non-nullable types, as described by Fähndrich and Leino [95]: during construction, fields of non-null types may contain null values.

2. **Morphability.** In examining the morphability of the five facets of a class purpose, we find that changes to four out of these are not arbitrary: The type definition of a subclass is an *extension* of the type definition of the superclass. Similarly, the mold of a subclass is an *extension* of the mold of the superclass. Also, the implementation can either *replace* or *extend* the implementation in the superclass, and the mill (constructor body of a subclass) must *extend* (i.e., invoke) the mill of the superclass.

In contrast, the forge of the subclass is *independent* of the forge of the superclass—the forge cannot be extended: it is not even inherited, and each class must define its own set of constructor signatures anew. The second constructor anomaly lies in the fact that the *the construction protocol is not inherited, yet, each constructor body **must** invoke a constructor of the base class.*

3. **Binding.** Third, it is mundane to see that a call to a constructor obeys a static binding scheme, and it takes just a bit of pondering to understand the difficulties that this scheme brings about. If a class C' inherits from C, then C' should be always substitutable for C. An annoying exception is made by constructor invocation sites in client code; these have to be manually fixed in switching from C to C'.

   The Gang of Four [105, p.24] place this predicament first in their list of causes for redesign, saying: "*Specifying a class name when you create an object commits you to a particular implementation instead of a particular interface*". With similar rationale, the very first item in Bloch's work on proper use of the JAVA language [29] recommends the use of static methods over constructors for obtaining instances of classes.

   Interestingly, in EIFFEL, although it has a strict dynamic binding policy, and although creation methods can be overridden, and although creation syntax is similar to method invocation, it is still the case that creation instructions such as `!!`x`.make` are statically bound.

5.3 Stages of Object Creation

Roma non fu fatta in un giorno.

— Italian proverb

Figure 5.1 demonstrates another issue with constructors. The figure depicts abstract class `Baby` whose constructor announces the baby's birth, and concrete class `NamedBaby` inheriting from it. Method `announce` is refined in `NamedBaby`, extending the announcement with details about the newborn's gender and name.

A client who has new baby boy, named "John", may then write

```
NamedBaby myBoy = new NamedBaby("John", true);
```

and be surprised by the printout "`New baby: Her name is null`", which is explained by the announcement being made before the subclass's fields are initialized. This lack of crisp separation between field initialization and the rest of the construction code, can even result in runtime exceptions, e.g., if `NamedBaby.announce` invokes `name.length()`.

C++ is not much better: The C++ equivalent of Figure 5.1 would print partial (albeit more sensible) output, "`New baby:`". Also, C++ would produce a runtime error if `announce()` is made abstract in class `Baby`.

This example motivates our distinction between three conceptual steps in an instance's birth process (later we shall argue that the separation between these is better served by factories): (a) *Creation*, in which the object's actual type is selected, memory is allocated and structured by the mold; (b) *Initialization*, in which fields are set to their initial values; and (c) *Setup*, in which the mill is executed.

These three steps correspond exactly to steps C1, C2 and C4 in the effects of a creation instruction `!!`x in EIFFEL [169, p.237]. The missing step, C3, is the attachment of the newly created object to the reference variable x; however, in languages such as JAVA and C++ the invocation

```
abstract class Baby {
   public Baby() {
      announce();
   }

   public void announce() {
      System.out.print("New baby: ");
   }
}

class NamedBaby extends Baby {
   String name;
   boolean isBoy;

   public NamedBaby(String name, boolean isBoy) {
      this.name = name;
      this.isBoy = isBoy;
   }

   public void announce() {
      super.announce();
      System.out.print(isBoy ? "His" : "Her");
      System.out.println(" name is " + name);
   }
}
```

Figure 5.1: Interwoven initialization and setup in JAVA constructors

of a constructor is an *expression* rather than a statement, and can be performed without assigning the result to a variable. (EIFFEL also supports the invocation of a creation procedure as an expression [90, Sec. 8.20.18], in which case step C3 is absent.)

The initialization step is realized in C++ by what is called the initialization list (written just after the constructor's signature). In JAVA and C# it is expressed using initializer values (or defaults) for fields, whereas the instance initializer block and the constructor bodies perform the setup. In EIFFEL, it is the assignment of standard default values to fields. As the example shows, however, initialization with default values is insufficient. Developers should be able to initialize all fields, across all levels of inheritance (i.e., complete step (b)) before setup code is being executed (step (c), the announcement in our example); *initialization* and *setup* should be unwoven.

We further note that none of these languages provides the developer with control over the *creation* step. Overloading the **new** operator in C++ grants us control over memory allocation, but not over the kind of object to be created, nor the decision if a new object has to be created at all.

We argue that good design of elaborate software systems often requires intervention in the creation step. Indeed, there are a number of successful design patterns, including ABSTRACT FACTORY, FACTORY METHOD, SINGLETON, and OBJECT POOL, which address precisely this need. The control that these "creational patterns" grant the programmer over the creation step is achieved by replacing constructor signatures from the forge facet with a different, statically-bound, common method (e.g., `getInstance`).[2]

[2]Such methods are sometimes called *factory methods*. While serving a similar purpose, they are different than our notion of *factories*.

Unfortunately, in contrast with most other patterns, the creational patterns cannot be implemented in object-oriented languages without revealing implementation details to the client: If class `T` is implemented as a SINGLETON, then clients of this class cannot write **`new`** `T()` and expect the correct instance to be returned; rather, they must be aware of the nonstandard creation mechanism, in violation of the uniform access principle. As a result, if a class evolves during development so that the new version employs (e.g.) an instance pool, all clients must be updated to use the `getInstance` method rather than the constructors; the use of creational patterns cannot be encapsulated as part of the class implementation.

Creational patterns often collide with inheritance. To enforce the use of a `getInstance` method and prevent accidental direct access to the constructors, all constructors can be made **`private`**, with the undesired implication that the class cannot be subclassed. The alternative of defining the constructor as **`protected`** is problematic in JAVA, since such constructors are visible to all classes in the same package.

Worse still, since method `getInstance` must be shared, it cannot be overridden in subclasses: If C' is a subclass of C, then the expression C'`.getInstance()` is valid—but returns an instance of C! This happens because `getInstance` is technically part of the type, while conceptually being part of the forge.

We shall see that factories enable a clear-cut separation between creation and initialization and setup, and allow for proper encapsulation of the creation step.

5.4 Factories

Class `STemplate` in Figure 5.2 shows how the SINGLETON design pattern can be realized by overriding **`new`** with the factory defined in lines 4–8. This factory is invoked whenever the expres-

```
1 class STemplate {
2    private static STemplate instance = null;

4    public static new() {
5       if (instance == null)
6          instance = this();
7       return instance;
8    }

10   STemplate() {
11      // setup code ...
12   }
13}
```

Figure 5.2: Using a factory to define a Singleton class

sion **`new`** `STemplate()` is evaluated, in class `STemplate` or any of its clients. Note that the factory is declared **`static`**, which stresses that it binds statically, and that (unlike constructors) it has no implicit **`this`** parameter. Examining the factory body we see that it always returns the same instance of the class. Thus, clients need not be explicitly aware of `STemplate` being a singleton, and will not be affected if this implementation decision is changed. (In the specific case of the SINGLETON design pattern, clients can compare instances to realize that only one exists. Other patterns, such as INSTANCE POOL, can be completely invisible to clients.)

A factory must either return a valid object of the class, or throw an exception. (Should the factory's return value be **`null`**, a `NullPointerException` results.)

Suppose that C' is a subclass of C. Then, a factory of C can return an instance of C'. This can be done by invoking any method which returns an instance of C', including a factory of C'—e.g., by a statement such as **return new** C'**(**$\cdots$**)**. If the factory however chooses to create an instance of class C, then it should invoke the constructor; yet writing **new** C**(**$\cdots$**)** (e.g., **new** `STemplate()` in the example) would recurse infinitely. Instead, the factory invokes the class constructor directly with the expression **this(**$\cdots$**)** (line 6 in the example).

We chose to overload the keyword **this**, particularly, its use for invoking a constructor. No ambiguity arises: In constructors, the function call **this(**$\cdots$**)** occurring in the first line can substitute the mandated call to **super** with a call to a different constructor in the same class (as in standard JAVA). Such a call does not create an instance, nor does it return a value, and it must appear only as the very first step in the constructor body. In a factory, **this(**$\cdots$**)** stands for a call to a constructor of the class. The call creates a new instance and returns a value; it may occur multiple times (or not at all), and in any location inside the factory body. The factory can choose to return the value generated by such a call. (In the case of the `STemplate` class, the value is cached to a static field, which is then returned.)

The constructor can only be called from a factory in the same class; any use of **new** C**(**$\cdots$**)**, either from outside class C or from inside it, will invoke a factory rather than a constructor.

While there are many different solutions to the specific issue of singletons, (e.g., declaring an object—rather than a class—in SCALA [184], or using prototype-based languages, such as CECIL [51]), the factory solution is not specific to singletons, and can be used for any creational design pattern. More examples will be presented below.

As usual with overloading, a factory may have parameters, which are matched against the actual parameters in the creation expression. A parameterized factory could be used for, e.g., implementing the FLYWEIGHT pattern: To do so, the factory returns, if possible, an existing object from its pool, and only creates an instance if no such object exists.

Like constructors, factories are not inherited. Had class C' inherited a factory **new()** from its superclass C, then the expression **new** C'**()** might yield an instance of C, contrary to common sense. Thus, the problem of C'`.getInstance()` yielding an instance of C, described in Section 5.2, does not occur with factories.

In contrast, when factories are employed, the expression **new** C**()** *can* yield an instance of C', since a subclass is always substitutable for its superclass.

Factories also allow developers to separate the initialization and setup stages of object construction. The mixup of Figure 5.1 is resolved by the factory based implementation in Figure 5.3, in which the call **new** `NamedBaby("John",true)` yields the expected "`New baby: His name is John`" output. The implementation in the figure adheres to the simple rule that field are initialized in constructors, and other setup is carried out by the factory. In particular, the announcement of the birth is made in the factory of `NamedBaby` (line 29).

5.4.1 Automatically Generated Factories

It's supposed to be automatic, but
actually you have to push this button.

— John Brunner, *Stand on Zanzibar*

A definition of a factory with a certain signature hides the constructor with the same signature. Such hidden constructors can only be invoked from the factory of a class, regardless of their access level. Let us now deal with the dual situation, i.e., a constructor without a factory. Backward compatibility of our extension is achieved by the following perspective: An expression of the form **new** S**(**$\cdots$**)** is always implemented by a factory whose signature matches the actual parameters.

```
1 abstract class Baby {
2    public Baby() {
3       // No fields that require initialization
4    }
5
6    public void announce() {
7       System.out.print("New baby: ");
8    }
9 }
10
11 class NamedBaby extends Baby {
12    String name;
13    boolean isBoy;
14
15    public NamedBaby(String name, boolean isBoy) {
16       // Initialization
17       this.name = name;
18       this.isBoy = isBoy;
19    }
20
21    public void announce() {
22       super.announce();
23       System.out.print(isBoy ? "His" : "Her");
24       System.out.println(" name is " + name);
25    }
26
27    public new(String name, boolean isBoy) {
28       NamedBaby result = this(name, isBoy); // Construction
29       result.announce(); // Setup
30       return result;
31    }
32 }
```

Figure 5.3: Re-implementation of Figure 5.1 using factories

This can be either a user-defined factory, or an *automatically generated factory* (AGFa). The automatic generation of factories is governed by:

> **The AGFa Rule:** Let c be a constructor with a signature σ in a non-abstract class S. Then, either (a) S has an explicit factory with signature σ, or (b) it has static AGFa with signature σ, which invokes c.

Figure 5.4 shows an example of the AGFa rule. The class defined in Figure 5.4(a) has a factory with no parameters. It also has a two-parameters constructor, with no matching factory. Figure 5.4(b) shows the AGFa that the compiler (internally) injects into the class.

Recall that in plain JAVA, instances of abstract classes cannot be created, even though such classes have constructors. The following argument uses the AGFa rule to explain this: *Instances can only be created by a* **new** *expression, which must have a matching factory. However, by the* AGFa *rule, abstract classes in plain* JAVA *do not have factories.*

```
class Complex {
   public static final Complex origin = new Complex(0,0);
   public Complex(double x, double y) {
      // instance setup ...
   }
   public static new() {
      return origin;
   }
}
```

(a) A class in which the zero-arguments factory returns a fixed instance

```
   public static new(double x, double y) {
      return this(x,y);
   }
```

(b) The factory added to the class by the AGFa rule

Figure 5.4: A class (a) with a constructor and its automatically-generated factory (b)

Conversely, if an abstract class S_a *does* define factories, then you can write **new** S_a`(`$\cdots$`)` in your code. Figure 5.5 shows an abstract class, `ScrollBar`, with a factory. This example is modelled after the famous example [105, p.87] of the ABSTRACT FACTORY design pattern. The code in the figure improves on the original implementation of the design pattern, in that the client is not aware that an abstract factory stands behind the scenes of the simple call **new** `ScrollBar()`. (As we shall see later, the internal implementation of the widget factory class itself can also be improved with factories.)

```
public abstract class ScrollBar {
   public static new() {
      WidgetFactory f = WidgetFactory.currentFactory();
      return f.CreateScrollBar(); // Select concrete subclass
   }
   // ... rest of the class omitted
}
```

Figure 5.5: An abstract class with a factory

As shown in Figure 5.6, interfaces may also have factories. The figure shows an interface, `DirectoryEntry`, whose factory makes it possible to obtain an instance of either of two implementing classes, `Folder` and `File`, depending on the parameter value.

5.5 Better Decoupling with Factories

The use of factories in interfaces can eliminate coupling between client code and library code. Consider, for example, the JAVA collection libraries. The standard library designers require, in very strong words, that interface types (like `List` and `Set`) will be used for method arguments:

> "*... it is of paramount importance that you declare the relevant parameter type to be one of the collection interface types.* ***Never*** *use an implementation type.*"
>
> – [46, p.526]; emphasis in the original.

```
public interface DirectoryEntry {
  public static new(String name) {
     if (FileSystem.isDirectory(name)) return new Folder(name);
     return new File(name);
  }
  // ... rest of the interface omitted
}
```

Figure 5.6: An interface with a factory

Similar recommendations apply to return types, field types, etc., all in spirit of Canning et al.'s original suggestions for separating the type and class notions using interfaces [47]. The coupling of client code to concrete implementation is indeed reduced by following this recommendation. But, such a coupling still remains, particularly at the point where a client is required to create an object.

Interfaces with factories can eliminate this coupling. In the case of the `List` interface, clients can generate instances of some default implementation by writing (say) **`new List()`**. The factory can choose the proper concrete implementation, possibly based on hints provided by the client. Figure 5.7 provides an example factory that can be used by the `List` class in JAVA's collections framework. Should new and improved implementations appear in future versions of the

```
public interface List {
  /**
   * @param synch indicates if a thread−safe list is needed
   * @param randomAccess indicates if O(1) element access is needed
   */
  public static new(boolean synch, boolean randomAccess) {
    if (synch) {
      if (randomAccess) return new Vector();
      return Collections.synchronizedList(new LinkedList());
    }

    // Else, synchronization is not needed.
    if (randomAccess)
       return new ArrayList();

    return new LinkedList();
  }
  // ... rest of the interface omitted.
}
```

Figure 5.7: One possible factory for the `List` interface

JAVA class libraries, this factory can be upgraded, and all clients will immediately benefit from the change. This solves the *change advertising dilemma* for new implementations of interfaces.

We would like to draw attention to the fact that following the recommendation of using interfaces rather than classes as method parameters, may in some situations *increase the burden* on clients rather than reducing it. Consider the learning effort of a user in search of a specific service in a software library. Suppose that this service is provided by a method m in an interface I. Then, before m can be invoked, the user must search for all the different implementation

of I, say classes $C_1, C_2, C_3, \ldots$, study them, and choose which of these to instantiate in order to generate an instance of I. Further, suppose that m takes a parameter of type interface I'. Then, the user must also search for all implementations of I', say classes $C'_1, C'_2, C'_3, \ldots$, study them all and choose the one appropriate for instantiation prior to invoking method m. If the constructor of the chosen class expects a third interface parameter I'', then, the user must further search for implementations $C''_1, C''_2, C''_3, \ldots$ of I'', etc.

A small example is method `Security.getProviders` in the JAVA standard library taking a `Map` as a parameter. In this parameter, the user can provide a set of selection criteria. Before the method may be used, even for testing or experimentation, the programmer must create an object representing such a test, and to do so, choose an implementation of the `Map` interface—but there are no less than seventeen such implementations in version 1.5 of the JDK.

Another example is method `JPanel.setBorder()` from the Swing GUI libraries, which expects a parameter of the `Border` interface. In order to use this method, the client must be spend time in studying the different implementations of this class, only to realize that yet a third class, `BorderFactory`, should be used to generate instances. With factories, the functionality of `BorderFactory` can be embedded in `Border`.

Searches for implementations of a given interface is usually not easy: implementations may be done by various different vendors, the list may change over time, and the selection between these may require a hefty learning effort. Interfaces (and abstract classes) with factories can therefore simplify the adoption of new, unfamiliar classes. Sometimes such a search is inevitable, but in many cases, it can be saved if the interface itself provides a reasonable selection of an implementation.

Writing a unit test code for a class whose methods take interface parameters is greatly simplified if these interfaces give ready-made instantiations. It is even conceivable that interfaces provide a stub implementation just for this purpose. For example, the standard JAVA interface `Runnable` can provide a stub implementation (perhaps defined as an inner class) in which the `run()` method does nothing.

5.6 Client-Side Factories

All examples so far defined factories in the same class on which the overload takes place. Factories of this sort are called *supplier-side factories*. It is also possible to define *client-side factories*, as demonstrated in Figure 5.8.

```
1 class Bank {
2    public static new Account(Customer c) {
3       if (c.hasBadCreditHistory())
4          return new LimitedAccount(c); // a subclass of Account
5       return Account.new Account(c);
6    }
7    // ... rest of the class omitted
8 }
```

Figure 5.8: A client-side factory for `Account`s in class `Bank`

Line 2 in the figure starts the definition of a factory. Unlike the previous examples, this definition specifies the returned type. The semantics is that the definition overloads **`new`** when used for creating `Account` objects from within class `Bank`. It is invoked in the evaluation of an expression of the form **`new`** `Account(c)` (where `c` is of type `Customer` or any of its subclasses) in

this context. This factory chooses (lines 2–6) an appropriate kind of `Account` depending on the particular business rules used by the enclosing class.

Unlike supplier-side factories, client-side factories *are* inherited by subclasses. Therefore, the factory from Figure 5.8 will also be used for evaluating expressions of the form **`new`** `Account(c)` in subclasses of `Bank`.

The client-side factory defined in `Bank` can be used by other classes as well, by writing

```
Bank.new Account(···) ,
```

or, after making a static **`import`** of class `Bank`, by simply writing

```
new Account(···) .
```

Figure 5.9 shows an implementation of the ABSTRACT FACTORY pattern with static binding. Classes `MotifWidgetFactory` and `PMWidgetFactory` each overload the **`new`** operator of all the GUI widgets.

```
class MotifWidgetFactory {
  public new ScrollBar() {
    return new MotifScrollBar();
  }

  public new Window() {
    return new MotifWindow();
  }

  // ... factories for other widget classes ...
}
```

```
class PMWidgetFactory {
  public new ScrollBar() {
    return new PMScrollBar();
  }

  public new Window() {
    return new PMWindow();
  }

  // ... factories for other widget classes ...
}
```

Figure 5.9: Widget-factory classes defined using client-side factories

A client wishing to use Motif, may write

```
import static MotifWidgetFactory.* .
```

This may be changed later to

```
import static PMWidgetFactory.*
```

should the GUI library need replacing.

The full semantics of a **`new`** call can now be explained as follows: Whenever a class is used in a **`new`** expression, its supplier-side factories enjoy an implicit **`import static`**. A client-side factory in scope can override this import.

The abstract widget factory example we have just described suffers from the problem that switching from Motif to PM requires a change to the client's **`import static`** statements. But there may be many such statements, in many source files. The remedy is to simply define an empty class,

```
class WidgetFactory extends PMWidgetFactory {}
```

and statically import it in all clients. This will direct all widget factory calls to `PMWidgetFactory`. The GUI can now be globally replaced with a single change, specifically replacing `WidgetFactory`'s superclass.

5.6.1 Dynamically Bound Factories

The above `WidgetFactory` can be thought of as a statically-bound implementation of the ABSTRACT FACTORY pattern, in that the decision on the concrete implementation is made at compile time. To make a dynamically-bound widget factory, we need *dynamically-bound factories*. These are defined, as the name suggests, without the **`static`** keyword. Figure 5.10 shows how such factories can be used in the classical implementation of the ABSTRACT FACTORY design pattern.

```
public abstract class WidgetFactory {
   public abstract new ScrollBar();
   public abstract new Window();
   // ... and other widgets.

   private static WidgetFactory f;
   public static new() {
      if (f != null) return f;
      if (GUI.isMotif()) return f = new MotifFactory();
      if (GUI.isPM()) return f = new PMFactory();
      //... etc.
   }
}
```

(a) The abstract widget factory class

```
class MotifWidgetFactory extends WidgetFactory {
   public new ScrollBar() {
      return new MotifScrollBar();
   }
   public new Window() {
      return new MotifWindow();
   }
   // ...
}

class PMWidgetFactory extends WidgetFactory {
   public new ScrollBar() {
      return new PMScrollBar();
   }
   public new Window() {
      return new PMWindow();
   }
   //...
}
```

(b) Two concrete widget factory subclasses

Figure 5.10: Using non-**`static`** factories to implement a dynamically bound abstract factory class

Figure 5.10(a) shows the abstract factory, while Figure 5.10(b) shows two concrete implementations. The factories of the widgets are all non-**`static`** and obey a dynamic binding scheme. Also worthy of note is the factory of this abstract class itself, which (while realizing the SINGLETON design pattern) determines at runtime the correct GUI library.

Figure 5.11 shows how dynamically-bound factories can be used to implement the FACTORY METHOD pattern (also known as VIRTUAL CONSTRUCTOR). The code in this figure implements the classic example by the Gang of Four [105, p.107] of an abstract `Application` class, bound to an abstract `Document` class. Each concrete subclass of `Application` can bind itself to a concrete subclass of `Document`, by overriding the dynamically-bound factories. The resulting

```
abstract class Application {
   List<Document> docs;
   protected abstract new Document();

   public void newDocument() {
      // Handles the File|New menu option
      doc = new Document(); docs.add(doc); doc.open();
   }
   // ... rest of the class omitted
}
```

(a) The abstract `Application` class

```
class MyApplication extends Application {
   protected new Document() {
      return new MyDocumentType(); // A concrete subtype
   }
   // ... rest of the class omitted
}
```

(b) One possible concrete subclass

Figure 5.11: Implementing pattern FACTORY METHOD with dynamically bound factories

code is very similar to the original Gang of Four example, except that the `newDocument` method uses ordinary construction syntax (implemented using our notion of a factory) rather than the nonstandard "factory method" dictated by the pattern.

Syntactically, the invocation of a dynamically-bound factory defined in class C for objects of class S is written as c`.new` $S(\cdots)$, where c is an instance of class C. The prefix "c`.`" can be dropped for code inside class C (so it is replaced with **`this`**).

It is not a coincidence that this looks very much like the JAVA syntax for creating an instance of a dynamic inner class: c`.new` $I(\cdots)$, where c is an instance of the containing class (possibly **`this`**) and I is the inner class's name. The constructor of a (non-**`static`**) inner class in JAVA is a method of the containing class, and not of the class it constructs—just like a client-side factory is a member of the containing class, and not of its target class. In fact, Nystrom, Chong and Myers [181] have shown that if the concept of inner classes is extended (using *nested inheritance*), most of the need for the FACTORY METHOD design pattern disappears. But while nested inheritance has many distinct advantages with regard to code modularity and the creation of extensible software systems, it only solves the need for factory methods for classes defined inside the same module as their client. Also, it does not remove the need for instance-management patterns like INSTANCE POOL or FLYWEIGHT.

5.7 Summary

Factories may be a minor perturbation to language syntax, but they are of benefit to langauge designers and programmers alike. We implemented factories as a JAVA extension using the Polyglot [182] extensible compiler framework (v. 2.0a4). This took approximately two workdays of a single programmer.

In our implementation, supplier-side factories (both explicit and AGFa) are realized as methods named `$new$` in the container class. The return type of `$new$` is the containing class itself.

Client-side factories are stored in the client, and are named `$new$`*classname*, where *classname* is the fully-qualified target class name, with every dot replaced by `$dot$`. For example, the factory for `Accounts` in class `Bank` (Figure 5.8) is realized as a method called `$new$com$dot$bank$dot$Account` (assuming `Account`'s fully qualified name is `com.bank.Account`). The return type of client-side factories is the target type (e.g., `Account`).

Any use of **`new`** is replaced by the proper method invocation, wrapped in a test that ensures a non-**`null`** value is returned (and throws an exception otherwise).

The addition of factories to interfaces is less straightforward, since interfaces in JAVA cannot contain any concrete methods. Instead, our extension synthesizes an **`static`** inner class (called `$NewHolder$`) for the interface, and places factories in this class.

The implementation generates bytecode that can be used on any JAVA virtual machine. As discussed in Section 5.4, the introduction of AGFas implies that JAVA-with-factories is fully compatible with existing JAVA source code. However, the code generated by our compiler assumes that all instantiated classes have been compiled using the same compiler, and thus have supplier-side factories (either explicit or AGFa). If factories are integrated into the JAVA language, full backwards compatibility with existing, pre-compiled classes can be achieved by having the class-loader (rather than the compiler itself) add any required AGFa to each class. This will work equally well for old and newly-compiled classes.

Clearly, the notion of factories is not limited to JAVA alone. It is not so difficult to approximate supplier-side—but not client-side—factories in SMALLTALK, by overriding the `new` class method. Adding factories to C$^{\#}$ seems rather straightforward, but it might take some cunning to add them to C++, since the language introduces two obstacles:

- First, C++'s intrinsic overloading of the **`new`** operator, is focused on the memory allocation problem rather than on instance generation. One possible solution is to introduce a new keyword, such as **`factory`**, to the language. Declarations for **`factory new`** can then exist alongside those for **`operator new`**. Such definitions can include both supplier-side factories (no explicit return type) and client-side ones (with a specific return type). Client calls to **`new`** will then be redirected to the factory, and should the factory decide to create a new instance, the **`new`** operator will be used for memory allocation (as before).

- The second obstacle is due to C++ value semantics. The compiler must know the space requirements of class instances allocated e.g., on the stack, but this is not possible with factories. A simple solution is that classes with factories are restricted to reference semantics representation only.

The EIFFEL language presents a different challenge for introducing factories. Unlike constructors in C++ or JAVA, creation procedures in EIFFEL are named. The advantage of this approach is that the distinction between the different kinds of objects that may be created is not by the kind of arguments, but rather through a meaningful name.

In terms of syntax design, the problem is that we must find a way, other than a special name, to distinguish between factory methods (which have no object to work on), and methods and creation

procedures (which start their work with a system-supplied object.)

We propose to the integration of factories into EIFFEL by introducing a new part to the EIFFEL class declaration, alongside **`feature`**s, **`create`**s, etc. The part is called **`factory`**, and it may be included only in non-**`expanded`** types. Following EIFFEL's accessibility rules, a class may provide different factories to different client classes by qualifying the **`factory`** part with a type list. Supplier-side factories have the return type "**`like Current`**"; any other return type indicates a client-side factory.

A subclass may re-classify a creation procedure as a factory (or vice-versa) when overriding it, and in particular, the default creation procedure, `default_create` (defined in the root class `ANY`) may be changed to a factory by any class that so desires. Following the principle of uniform access, clients that include a creation instruction (or a creation expression) employ the exact same syntax regardless of whether a creation procedure or a factory is being used. The syntax `!!`x`.make` is used by clients to obtain an instance, regardless of whether `make` is a creation procedure or a factory. Interestingly, the distinct name for each factory and creation method implies that this extension maintains backwards compatibility with existing code, without resorting to automatically-generated factories (AGFas).

Figure 5.12 shows an EIFFEL version of the singleton class from Figure 5.2. This class re-

```
1  class
2     S_TEMPLATE

4  factory  -- obtain an instance
5     default_create: like Current is
6        once
7           !!Result.instance
8        end

10 create {NONE}  -- private instance creation mechanism
11    instance is
12       do
13          -- initialize fields, etc.
14       end

16 end  -- class S_TEMPLATE
```

Figure 5.12: A singleton defined in EIFFEL using a factory

classifies `default_create` as a factory, so clients can use the creation instruction `!!`x (for a variable x of type `S_TEMPLATE`) to obtain the shared instance.

As we can see from the figure (line 7), no special syntax is needed to create an instance from inside the factory (the equivalent of the special **`this()`** call in the JAVA version): Since a class may include both creation procedures and factories, each with distinct names, there is no risk of undesired recursion. Whenever a new instance is required, the factory simply calls a (possibly private) creation procedure.

In summary, it is easy to see how middleware applications using shakeins can benefit from factories. Figure 5.13 is a variant of Figure 5.8, in which the client-side factory always applies the `Secure` shakein to the selected type before generating an instance. Thus, any `Account` created by a `Bank` (or a subclass thereof) will be a `Secured` one. Should the developers desire that a given shakein S must be applied to all instances of some class c, then the supplier-side factory of

```
class Bank {
   public static new Account(Customer c) {
      if (c.hasBadCreditHistory())
         return new Secure<LimitedAccount>(c); // a subclass of Account
      return Account.new Secure<Account>(c);
   }
   // ... rest of the class omitted
}
```

Figure 5.13: A client-side factory for `Accounts` in class `Bank` which applies a security shakein to all generated accounts

c can be made to return only instances of $S\,\langle c \rangle$.

Both supplier- and client-side factories can choose the particulars of which shakeins to apply based on configuration values set, e.g., in deployment descriptors. This completely alleviates the need for home interfaces for EJBs, while allowing EJB clients to obtain instances as simply as invoking **new**. The mechanism also guarantees that no undesired instances of the "raw" class, with no shakeins applied, are ever created in an uncontrolled manner. (In the standard EJB solution, this can happen if a developer mistakenly invokes **new** rather than using the home object.) Factories therefore do away with an additional level of complexity introduced by other middleware frameworks.

Chapter 6

Object Evolution

Away to Proteus! Ask that wondrous elf:
How one can come to be and change one's self.

— Johann Wolfgang von Goethe, *Faust*,
Part II, Act II, Scene VI[1]

A frog may turn into a prince, if kissed. The modeling of such a phenomenon in the object-oriented world is known as (dynamic) *object reclassification*. As indicated by the literature, starting at least as early as 1993 [218], and as we shall reiterate here, the need for reclassification arises frequently in the software world, and cases such as the frog and prince example are not rarities. To realize reclassification, one may use SMALLTALK's "`becomes:`" method [114, p. 246], a powerful and popular programming mechanism. Yet, "`becomes:`" is notoriously unsafe and difficult to use.

There are inherent difficulties in object reclassification which probably explain why only a handful of mainstream programming languages offer support for this feature. Other than SMALLTALK's "`becomes:`", we are aware of the following: early versions of PYTHON [163] which allowed assignments to the `__class__` attribute. Such assignments were not types-safe, and consequently forbidden in later versions of the language. Similarly, in PHP version 4 one could dynamically add both data- and function- members to instances, an operation which can be thought of as reclassification. Again, support for this dwindled in version 5 of the language.

State of the art research on object reclassification (see e.g., [77, 86, 87, 107–109, 201]) battles with the challenge of producing a type-safe alternative to `becomes`. In this chapter, we are interested in the tradeoff between expressive power and type-safety offered by a particular kind of reclassification, what we call *object evolution*, by which dynamic changes to an object's class are *monotonic*—an object may gain, but never lose, capabilities. Once an object evolves, it cannot retract its steps and be reclassified into its previous class.

Evolution is not as general as reclassification, and may not allow changes as drastic as a frog turning into a prince. Our main interest is not so much with the theoretical foundation of object evolution, which previously received attention in the literature, but rather in the practical issues raised by the introduction of evolution into current languages.

We argue that there are many applications of monotonic evolution in practical systems. The monotonicity property makes it easier to maintain static type safety with object evolution than in general object reclassification. Note that the monotonicity property may make evolution irreversible. This restriction is ameliorated by separating the notion of class from that of type, and with the help of shakeins we find that object evolution can support repeated state changes, and even undo changes, under certain limitations.

[1] English by George Madison Priest.

We shall also see that object evolution requires less changes to the host object-oriented programming language and collects a reduced performance toll, mostly because all descends in the inheritance hierarchy are necessarily monotonic.

The contributions of this chapter include:

1. *The Case for Object Evolution.* We argue that object evolution is in line with object-oriented thinking and accepted design paradigms. For example, the STATE design pattern [105] is naturally expressed with evolution. Object evolution also integrates well with other useful programming techniques, such as lazy data structures.

   We will explain why object evolution is thread-safe, and why this should make it easier to integrate into existing systems.

2. *Concrete Language Extension.* The integration of evolution with other language mechanisms did not receive much attention in the past. To make our proposal concrete, we studied the issues of introducing evolution into the JAVA programming language. In doing so, we noticed an interesting problem of proper initialization in the course of reclassification: The object must maintain the state of existing fields, which may have changed after their initialization, yet its newly acquired fields must also be properly initialized. The class invariants of the new class must likewise be satisfied.

   We present *evolvers* as a complementary mechanism to constructors, containing the additional initialization code that separates an object of one class from an object of another. Like constructors, evolvers can accept parameters, indicating that an object cannot be evolved into a new class without some additional required information.

   Conversely, we also show that in many cases, *default evolvers* can be automatically derived from the constructors of a class.

3. *Chart of the Language Design Space.* This chapter presents three independent flavors of the object evolution mechanism, tagged *I-Evolution*, *M-Evolution* and *S-Evolution*, relying on inheritance, mixins and shakeins, respectively. In saying that the flavors are independent we mean that a language designer can choose to implement any of the seven possible combinations, ranging from choosing a single approach to integrating all three.

4. *Analysis of Runtime Failures.* Just as an object construction operation can fail (e.g., when the constructor throws an exception), so can object evolution. We study and compare the relative merits of the three approaches by the kinds of runtime failures they may generate.

5. *Implementation Strategies.* Finally, we turn to dealing with implementation. We outline several alternatives, each appropriate for different usage scenarios. Empirical data about the frequency and type of object evolution operations that are common in programs will dictate the preferred implementation strategy.

Although in the sake of concreteness, object evolution is presented as an extension to the JAVA programming language, nothing in the discussion binds the mechanism specifically to JAVA. Most other statically-typed object-oriented languages, such as EIFFEL or C#, are just as applicable.

6.0.1 Three Approaches to Object Evolution

We will present three theoretical approaches to the concept of object evolution, each with its unique power of expression and underlying metaphor. These approaches are not mutually exclusive; all three, or any subset of thereof, can co-exist in the same programming language.

The first approach, which we call *I-Evolution*, is based on standard inheritance. Here, an object can evolve into any subclass of its own class. This change is necessarily monotonic, since a subclass may only *extend* its base class. Evolution is expressed using the syntax $v \rightarrow C(\cdots)$, meaning the object referenced by variable v is evolved (using the $\rightarrow$ operator) to an instance of class C. The parenthesis will often be empty, i.e., $v \rightarrow C()$; however, the evolution process may accept parameters.

The evolution target C must be a subclass of v's type. The set of possible reclassification targets is therefore defined by the inheritance tree. The similarity of this tree to taxonomy trees used in biology to describe evolution inspired the process's name.

The second approach, *M-Evolution*, is based on mixin inheritance [38]. With M-Evolution an object can only evolve into a subclass defined using *mixins*. Recall that given a class C and a mixin M, the application of M to C, denoted $M\langle C\rangle$, is a subclass of C. Class $M\langle C\rangle$ is an ordinary class, and can therefore serve as the target of an I-Evolution operation. With M-Evolution, however, the evolution target is selected—and possibly generated—at runtime, based on the object's actual type at the time of evolution. The M-Evolution operation $v \rightarrow M\langle v\rangle(\cdots)$ selects $M\langle V\rangle$ as its target, where V is v's runtime type. Thus, an M-Evolution can be thought of as an application of a mixin to an instance rather than to a class. Because a mixin can only extend its operand, M-Evolution is also guaranteed to be monotonic.

Finally, *S-Evolution* is limited to shakein inheritance. As explained in Chapter 2, shakeins are a programming construct that, like mixins, generates a new class from a given class parameter. Unlike mixins, a shakein does not generate a new *type*. Given a shakein S and a class C, the shakein application $S\langle C\rangle$ represents a new class but not a new type; it is an *implementation class* [72].

S-Evolution can be thought of as an application of a shakein to an instance rather than to a class. Such an application, by definition, does not change the object's type (in contrast to its class); in particular, the shaked object cannot understand any new messages. S-Evolution is therefore *trivially* monotonic, and resembles instance-specific behavior facilities in SMALLTALK [24]. However, unlike instance-specific behavior, the behavior itself is described in an organized manner (in the shakein's definition) rather than relying on ad-hoc changes to an object's message handlers.

A unique feature of S-Evolution is that it can be temporary, i.e., in certain circumstances, the object may later re-evolve into a different shakein-based class, *undoing* (or "de-evolving") the effect of the first shakein. Whereas shakeins can be used as enhanced aspects, S-Evolution introduces the possibility of using shakeins as dynamic aspects [148, 190, 192].

6.0.2 Evolution Failures

All three approaches presented above integrate with the static type system. Once an object has evolved, it assumes a new class, and it will never be the case that an object receives a message it cannot deal with.

However, in certain circumstances that cannot be statically determined, the evolution operation itself might fail. Such cases are called *evolution failures*. For example, the most trivial case of evolution failure is when the reference to the object to be evolved happens to be `null` at runtime.

Thus, with regard to type safety, object evolution can be likened to a downcast operation: The operation itself might fail, but once completed successfully, the reference or object can be safely accessed using its newly-assumed class.

Each of the three approaches entails its own set of possible causes for evolution failure.

In the I-Evolution operation $v \rightarrow C(\cdots)$, the evolution target C must be a subclass of v's type. Herein lies a risk of evolution failure, since while C can be verified to be a subclass of v's *static* type, we cannot verify in advance that it is also a subclass of v's *dynamic* type. For example,

we may try to evolve an object of static type `Pet` to type `Dog`, but if the object's runtime type is `Cat` (a different subclass of `Pet`), this evolution attempt will fail.

The target of the M-Evolution operation $v \rightarrow M\,\langle v \rangle\,(\cdots)$ is $M\,\langle V \rangle$, where V is v's runtime type. The operation's target is therefore necessarily a subclass of v's dynamic type, avoiding the risk presented by I-Evolution operations. The risk is further reduced by defining the concept of *idempotent mixins*, i.e., mixins that can be repeatedly applied to a class with no adverse effect. However, M-Evolution can still fail if mixin M cannot be applied to V for one of two reasons: If V is a **`final`** class, or if the application results in accidental overriding [8].

Finally, because it only offers trivial monotonicity, S-Evolution is the least susceptible to failure. Like M-Evolution, S-Evolution selects the target class based on the evolving object's dynamic type, thereby avoiding the risk faced by I-Evolution.

Unlike mixins, shakeins are immune from accidental overriding, because they can only override existing methods or introduce **`private`** ones. Thus, S-Evolution can only fail when a shakein is applied to a an object whose dynamic type is **`final`** (assuming shakeins are implemented using inheritance).

Chapter outline. Section 6.1 makes the case for object evolution using four motivating examples, including the STATE design pattern, a compiler implementation, and a lazy data structure. Section 6.2 presents the concept of object evolution in greater detail, and explains where a simple evolution operation might fail. Sections 6.3, 6.4 and 6.5 present the I-, M-, and S-Evolution variants, respectively. Possible implementation strategies are discussed in Section 6.6. Section 6.7 compares object evolution to other reclassification approaches, and Section 6.8 concludes.

6.1 The Case for Object Evolution

As early as 1993, Taivalsaari [218] argued that design often needs objects that change their behavior at runtime. (Taivalsaari's own proposed solution, *modes*, can be nicely implemented using S-Evolution.) This need for reclassification motivated much subsequent research, including [52, 72, 76, 77, 86, 87, 107–109, 201].

An important demonstration of this need is provided by the programming language ***e*** [140], manufactured and sold by Cadence, and used widely in the hardware verification industry. What is called *when-inheritance* [134] in ***e*** is in reality a mechanism by which an object reclassifies itself. When-inheritance is similar in fact to S-Evolution.

This section emphasizes the case for object evolution showing several cases where object evolution can be used to improve upon program design. Section 6.1.1 explains how the STATE design pattern maps naturally to evolution. In Section 6.1.2 we show how program design of lazy data structures can benefit from evolution. Two examples are used there for concreteness: The DOM representation of HTML data structures, and the evolution of the Abstract Syntax Tree in the different stages of the compilation process.

Finally, Section 6.1.4 demonstrates how evolution can be used to slightly ameliorate implementation issues in face of the recalcitrant co-variance problem [49].

6.1.1 Implementing the STATE Design Pattern

In their presentation of the STATE design pattern, the Gang of Four use a TCP connection class as an example [105, p.305]. Figure 6.1 shows the class structure realizing this example.

The connection object is required to respond differently to messages (such as `open`) based on its current state, which can be either of "established", "listen" (active) and "closed". Rather than

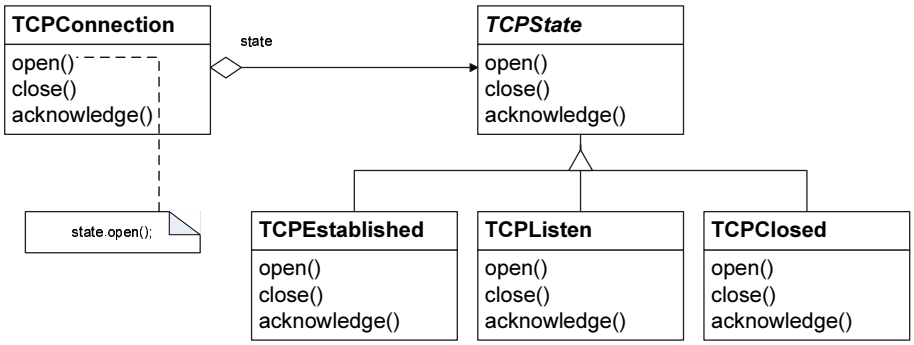


Figure 6.1: The state-changing `TCPConnection` class (from [105, p.305])

represent the state as an **`int`** data member (or an **`enum`**), the design pattern suggests representation using a data member s of a dedicated *state* type S, to which all requests are delegated.

The abstract state class (`TCPState` in this example) has a concrete subclass for each possible state. Each such subclass responds differently to messages; for example, the `close` message changes the object's state (to "closed") if it is in either the "established" or "listen" states, but throws an exception if it is already in the "closed" state.

To change its state, the object simply replaces the instance to which the state variable s refers.

The intent of the pattern is to "[allow] an object to alter its behavior when its internal state changes. *The object will appear to change its class*" [105, p.305; emphasis added]. But, as this description suggests, the same effect can be better achieved by literally allowing the object to change its class at runtime.

Figure 6.2 outlines the code for an implementation of the same `TCPConnection` class, which relies on object evolution. Here, the state-changing operations use object evolution (lines 6, 10 and 22) to change the object's state by advancing its class. Since evolution is transparent to aliasing, any reference to the connection will now use the newly-classified object, and thus any method invocation will be affected by the new state.

Several benefits of the approach should be immediately apparent:

- *Fewer classes*. Whereas the STATE design pattern solves this particular problem using five classes (a wrapper class, an abstract state class, and three concrete state classes), the evolution-based solution requires only three (one class per state).

- *No code duplication.* In the STATE pattern, we find that the state class, `TCPState`, copies the interface of the wrapper class. Such fragile code duplication is not needed with object evolution.

- *Greater efficiency*. The code in Figure 6.2 does away with the need to delegate every incoming message from the wrapper class to the state object, thereby improving performance (*cf*. the implementation of `open` in Figure 6.1). It also does away with the `state` data member, thus reducing memory requirements.

Section 6.3.2 discusses the limitations of this solution. A better solution, using shakeins and state-groups, is presented in Section 6.5.1.

6.1.2 Lazy Data Structures

Since object evolution moves objects down the inheritance tree, it can be used to evolve instances of general, top-level classes into more specific sub-classes. Such changes can be useful as more

```
1  public class TCPConnection {
2      // This class represents the initial state, ''listen''.

4      public void open() {
5          // ... establish the connection ...
6          this→TCPConnectionEstablished();
7      }

9      public void close() {
10         this→TCPConnectionClosed();
11     }

13     public void acknowledge() { ... }
14 }

16 class TCPConnectionEstablished extends TCPConnection {
17     public void open() {
18         // ... ignore
19     }

21     public void close() {
22         this→TCPConnectionClosed();
23     }

25     public void acknowledge() { ... }
26 }

28 class TCPConnectionClosed extends TCPConnectionEstablished {
29     public void open() {
30         throw new IllegalStateException();
31     }

33     public void close() {
34         throw new IllegalStateException();
35     }

37     public void acknowledge() { ... }
38 }
```

Figure 6.2: Implementing `TCPConnection` and its state-changes using object evolution

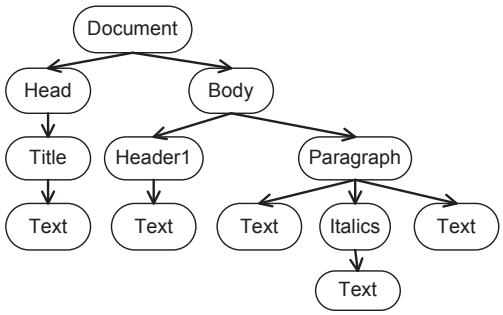
information about the object is obtained (see example in Section 6.1.3 below), or for lazy evaluation of data structures. In the latter case, nodes in the data structure are first represented as general "node" objects, to be replaced by specific nodes on a per-need basis.

Consider, for example, the hierarchical in-memory representation of HTML files (or files of other markup languages, including XML), and in particular, DOM (Document Object Module) trees [136], a common such representation.

Figure 6.3(a) shows a simple HTML file. In Figure 6.3(b) we see the DOM representation of

```
1 <html>
2  <head>
3   <title>Welcome!</title>
4  </head>
5  <body>
6   <h1>Welcome to this page.</h1>
7   <p>As you can see, it contains
8   nothing <i>meaningful</i>.</p>
9  </body>
10 </html>
```

(a) An HTML document



(b) The document's DOM tree

Figure 6.3: A sample HTML document and its Document Object Model (DOM) tree. Nodes in the tree are instances of classes shown in the hierarchy of Figure 6.4

this file. We see that every opening tag is represented as a tree node, while the HTML content that occurs from this tag to its matching closing tag is represented as the subtree rooted at this node. A sequence of plain text, with no tags, is represented as a leaf node of type `Text`.

For example, the HTML fragment

```
<i>meaningful</i>
```

in line 8 of Figure 6.3(a) is represented as a node of type `Italics` with a subnode of type `Text`.

Figure 6.4 is a UML class diagram for the classes used in Figure 6.3(b). We see that abstract class `Node` is at the root of the hierarchy, class `Text` is a final class, and that different classes offer different services.

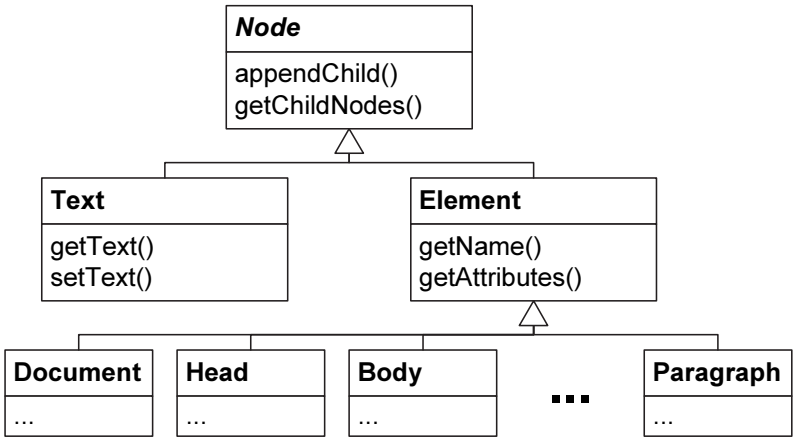


Figure 6.4: Classes used for representing DOM trees

Since programs often end up using only part of the tree, a common optimization technique is lazy evaluation, by which a given object represents an entire subtree, to be expanded on a per-need basis.

In our example, lazy evaluation means that class `Node` is not abstract. Instances of `Node` denote portions of the HTML which were not parsed yet.

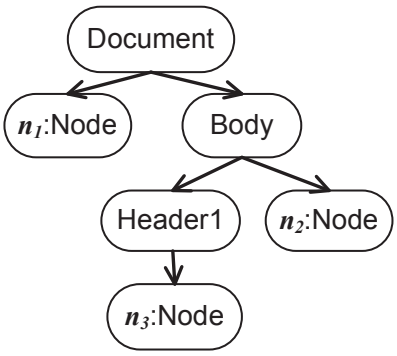


Figure 6.5: A possible stage in the lazy creation of the tree from Figure 6.3(b). Nodes labeled n_1, n_2 and n_3 were not yet expanded

Figure 6.5 shows a possible intermediate state of the tree from Figure 6.3(b). The left-hand child of the root node, marked n_1 in the figure, represents the subtree contained in the **<head>**...**</head>** tag pair. Should the program code delve into this subtree, this node must be expanded, with new nodes created to represent its children.

In a lazy implementation of a DOM parser which does not use object-evolution, the expansion step must either (a) replace the node object n_1 with a specific node (i.e., create a new object and discard the old one), or else, (b) change the state of this object, so that it now represents a specific node.

The first solution requires that the parser must not allow references to node n_1 to leak, since the existing object must be replaced with a new one, and the old one must cease to exist. This complicates the tree implementation, and in particular requires an expansion of the subtree whenever n_1 is requested by any client, even if that client will not eventually access any child of n_1. (Things are further complicated in other data structures, such as directed graphs, where there are multiple references to the object.)

The second solution implies that the `Node` class must have two operational states, pre- and post-expansion. After the expansion, it must be able to act as any of its subclasses; in this example, n_1 must be able to act as an instance of class `Head` after its expansion, whereas n_2 must be able to act as an instance of `Paragraph` and n_3 as `Text`. The STATE pattern can be used here: maintain a field of type `Node` in each un-expanded node (e.g., n_1), and, upon expansion, assign a new instance of a specific subclass (e.g., `Head`) to this field. Any message received by the node will now be delegated to the more specific `Node`-typed field. An implementation of a lazy DOM tree with the STATE design pattern is inefficient, since it requires delegation. Such an implementation is also cumbersome, complicating both the design and the implementation of classes: `Node`'s API must include the union of all methods found in all subclasses, and some of these methods might fail at runtime (e.g., the method `getText` from class `Text` must be processed in the expanded n_3, but rejected by the expanded n_1 and n_2).

Now consider the object evolution-based solution. Whenever the subtree represented by object n_1 must be expanded, we can *evolve* this object from its current class (`Node`) to any of its subclasses, and in particular `Head`. The evolution operation $n_1 \rightarrow$`Head(`$\cdots$`)` affects the object itself, so all references to it are immediately affected; there is no need to track and update each reference explicitly. The object's new class is a subclass of its old, so that the object can still accept and process any message it could previously accept; and it can now also accept and process messages added by the interface of its specific new class.

The object evolution-based solution requires no delegation, and no new object is introduced

into the system. There is also no need to artificially inflate the interface of the superclass `Node`, and type safety is maintained; e.g., if a `Node` is evolved into a `Head`, it has no `getText` method, and any attempt to use such a method will fail at compile-time.

6.1.3 Representing Knowledge Refinement

An important special case of lazy data structures are systems in which knowledge increases over time, and the increase in knowledge allows us to replace a general class with a specific one. To this end, object-oriented class hierarchies are often used, where top-level classes represent abstract notions, while classes deeper in the inheritance hierarchy represent more and more specific versions of these notions.

As a concrete example, consider the classes used to represent the abstract syntax tree (AST) data structure in a compiler implementation. A top-level class, `MethodInvocation`, can be used to represent the general notion of an invocation expression, whereas its subclasses represent specific invocation types, e.g., `StaticMethodInvocation` for static method calls, `DynamicMethodInvocation` for ordinary calls, `InterfaceMethodInvocation`, etc. Each of these subclasses is a specific, *refined* version of the superclass.

In many compiler designs, the front-end (parser) generates an abstract syntax tree from the program source code; the back-end module then processes this tree. Often, the parser does not have the knowledge required for classifying a given node in the AST at its most refined representation level. For example, given the source fragment "`x.m()`" in a JAVA program, the parser will generate a `MethodInvocation` node in the tree. The back-end will then replace this node with a more specific node, such as `InterfaceMethodInvocation`, based on data obtained from the symbol table regarding `x`'s type and the declaration of method `m` in that type. The change is a *refinement* based on gathered knowledge.

Just as with lazy data structures, a refinement entails either (a) the creation of a new node object to replace the old one, or (b) representing all possible options in the top-level class (`MethodInvocation` in this example). As in the case of DOM tree nodes, the first option implies that the AST data structure must prevent the reference to the raw type from leaking; all references must be meticulously tracked, and replaced when the object is refined. The second option implies that the top-level class must contain knowledge about all possible refinement options. This contradicts modular design and complicates future expansions.

With object evolution, refinement is represented as the object sliding down the inheritance tree to a state that represents our new, refined knowledge about it. All references are immediately updated, while the program design remains completely modular.

6.1.4 Supporting Data Covariance

Subclasses often use fields defined in superclasses in a more specialized manner than the fields' original definition. For example, consider the two linked list implementations in Figure 6.6. Class `LinkedList` in the figure is a unidirectional linked list; each node is an instance of the inner class `Node`, which contains a link to the data item as well as a `next` reference. The subclass `BidiLinkedList` is a bidirectional linked list. Here, each node is an instance of `BidiNode`, a subclass of `Node` which adds a `prev` reference. The fields `head` and `tail`, defined as `Node` fields in `LinkedList`, are *covariant fields*, since in `BidiLinkedList` they always hold an instance of `BidiNode`.

The problem lies in method `append`. We see that `LinkedList.append` creates an instance of `Node` to hold the new list item, and chains this node into the list. Ideally, `BidiLinkedList.append` should be a refinement[2] of its inherited version, only adding code for manag-

[2] "Refinement" here refers to method refinement [44, Ch.11].

ing the back-linking of nodes. However, in standard JAVA, this is not possible, because the nodes in `BidiLinkedList` must be created as `BidiNode` instances. The method must therefore be overridden and its code duplicated.

Existing solutions to this problem include the FACTORY METHOD design pattern [105], family polymorphism [94], simultaneous instantiation of templates [215], nested inheritance [181], and dynamic factories [64]. Object evolution presents another possible solution.

```
1  class LinkedList {
2    // Nested class for nodes
3    static class Node {
4      Node next = null;
5      Object data;
6
7      Node(Object obj) {
8        data = obj;
9      }
10   }
11
12   protected Node head = null;
13   protected Node tail = null;
14
15   void append(Object obj) {
16     Node n = new Node(obj);
17     if (head == null) {
18       head = tail = n;
19       return;
20     }
21     tail.next = n;
22     tail = n;
23   }
24
25   // ... rest of the class not shown
26 }
```

```
27 class BidiLinkedList
28                 extends LinkedList {
29   static class BidiNode
30         extends LinkedList.Node {
31     BidiNode prev = null;
32
33     BidiNode(Object obj) {
34       super(obj);
35     }
36
37     // ... rest of the class not shown
38   }
39
40   void append(Object obj) {
41     BidiNode oldTail =
42                   (BidiNode)tail;
43     super.append(obj);
44     // tail was constructed by super.append
45     // as a simple Node; evolve it:
46     tail→BidiNode();
47     tail.prev = oldTail;
48   }
49
50   // ... rest of the class not shown
51 }
```

Figure 6.6: Two linked list implementations. Class `LinkedList` (left) uses vanilla JAVA; the bi-directional linked-list (right) uses evolution for covariance in the `append` method (lines 40–48). This allows the inherited `append` to be refined rather than replaced

Figure 6.6 shows how, using object evolution, `BidiLinkedList`'s version of `append` can effectively reuse the inherited code without having to duplicate it. The gist of the implementation is in line 46, which uses object evolution to evolve the object created by the method's precursor (called in line 43). Finally, the node's back-link `prev` is set (line 47).

6.2 Object Evolution

An object evolution operation replaces, at runtime, the type of an object with the type of a selected subclass. As the target type is always a subclass of the current type, the set of class members is either unchanged or enlarged, i.e., the change is monotonic. Since no member is removed by the operation, we have a guarantee that *any message understood by the object prior to the evolution*

operation is understood after the operation as well, thereby ensuring type safety after the evolution occurred.

The action of object evolution is executed on a particular reference to the object, but it affects the object itself, not the specific reference used to express it. Any reference to the object, including fields, local variables (or parameters) of the currently executing code, local variables of methods up the call stack, and local variables in the call stacks of other threads now reference the evolved object. Object evolution is therefore transparent to aliasing.

Evolution is written using the syntax $v \rightarrow C(\cdots)$, meaning the object referenced by variable v is evolved (using the $\rightarrow$ operator) to an instance of class C. The $\rightarrow$ operator is written in standard ASCII as "`->`". The parenthesis will often be empty, i.e., $v \rightarrow C()$; however, the evolution process may accept parameters, as described below.

For example, consider the lazy tree evaluation scenario discussed in the introduction, and in particular the class hierarchy presented in Figure 6.4. Given the variable definition and initializations

```
Node n1 = new Node(···);
Node alias1 = n1;
```

we can now evolve `n1` into any subclass of `Node`; for example, the statement

```
n1→Head();
```

will evolve the object referenced by both `n1` and `alias1` from an instance of class `Node` to an instance of its indirect subclass `Head`.

While we have stated above that the evolution operation affects the object, rather than the specific reference used to express it, it does have an impact on that reference: Following the evolution operation $v \rightarrow C(\cdots)$, within the same innermost block of code, v's static type is C, a subclass of its current static type.

In our example, following the evolution statement above, `n1`'s static type is `Head`. Operations defined in class `Head`, or its superclass `Element`, can therefore be applied to it, as in

```
Attributes attr = n1.getAttributes();
```

However, the same method cannot be applied to `alias1`, whose static type remained unchanged. To apply a newly-acquired operation to an alias, it can be downcast into the object's new type; for example,

```
Attributes attr = (Head)alias1.getAttributes();
```

is valid.

The change of static type for the evolution reference lasts only until the end of the current block of code. The code fragment in Figure 6.7 provides an example: Following the evolution operation in line 6, the method invocations in lines 7 and 9 will compile and succeed at runtime, whereas the invocation in line 8 will fail to compile, as discussed above. The invocations in lines 5 and 12, despite seeming identical to that in line 7, will not compile; the first, because it appears prior to the evolution operation, and the second because it resides outside of the operation's containing scope.

Finally, the invocation in line 13 uses a downcast; it will compile successfully, but (as with any downcast operation) might fail at runtime, depending on the result of the condition in line 4.

```
1  Node n = new Node(···);
2  Node alias = n;

4  if (someCondition()) {
5     n.getAttributes(); // Compile-time error
6     n→Head();
7     n.getAttributes(); // Will succeed
8     alias.getAttributes(); // Compile-time error
9     (Head)alias.getAttributes(); // Will succeed
10 }

12 n.getAttributes(); // Compile-time error
13 (Head)n.getAttributes(); // Downcast might fail at runtime
```

Figure 6.7: An example of the effect of evolution on references to the object

6.2.1 Evolvers: Maintaining Class Invariants at Evolution

The object evolution operation takes an instance of one class and mutates it into an instance of another. Yet simply adding new fields and methods is not sufficient. Consider an object v of type C_0 that undergoes an evolution process, $v \rightarrow C(\cdots)$. The object state, which initially satisfies the class invariants [169, Sec. 11.8] of C_0, must now satisfy C's invariants.[3] (Note that while JAVA classes do not have invariants specified explicitly in code, they almost always have implicit *conceptual* invariants, often made explicit in the documentation.)

Standard objects of class C go through an orderly construction process, which ensures that class invariants are satisfied once the constructor execution completes. In particular, the constructor begins by invoking an inherited constructor (using the keyword **`super`** in JAVA[4]); after the inherited constructor returns, the invariants of the superclass C_0 are satisfied, and the rest of the constructor body must ensure that the additional invariants introduced in C are also satisfied. But object v had only gone through the C_0 construction process. We therefore conclude that object evolution must allow the mutating object to execute any required code in order to meet the invariants of its target class.

To this end, we define *evolvers*, which are constructor-like class members executed upon evolution. Syntactically, an evolver for class C is named $\rightarrow C$ (whereas a constructor is named C). For example, an evolver defined in class `Head` must be called `→Head`. Like constructors, evolvers have no return type. Also like constructors, evolvers can be overloaded, and a single class may contain multiple evolvers. The parameters $(\cdots)$ passed to the evolution operation $v \rightarrow C(\cdots)$ dictate which evolver will be used.

Unlike constructors, evolvers do not begin by calling an inherited version. When the evolver begins its execution, the current object (**`this`**) is an instance of the class C, which does not yet satisfy its class invariants; it only satisfies the invariants of its superclass C_0. This is similar to the state of the object in the constructor, right after the call to **`super`**$(\cdots)$ is completed. It is therefore the evolver's role to initialize the newly acquired fields, possibly based on the values of the inherited fields, so that all invariants are satisfied.

For a concrete example, consider Figure 6.8, showing an implementation of class `Head` which contains a field `title` of type `String`.

[3] The subclass invariants are always additions to those of the superclass; see the *invariant inheritance rule*, [169, p. 465].

[4] Or the keyword **`this`**, which delegates to a different constructor in the same class. Still, at the end of the

```
1 class Head extends Entity {
2    private String title;

4    public Head() { // Constructor
5       super(); // Call superclass constructor (could also be implicit)
6       initializeTitle();
7    }

9    public →Head() { // Evolver
10      initializeTitle();
11   }

13   private void initializeTitle() {
14      // Parse the node's content and set title accordingly
15      Node titleElement = findSubElementByName("title");
16      title = titleElement.getTextContent();
17   }

19   // ... rest of the class not shown
20 }
```

Figure 6.8: A class with an explicit evolver

The implicit invariant of this class is that field `title` must be **`null`** if no title was specified in the **<head>** part of the HTML file, or else it must be set to the specified title value.

To preserve this invariant, both the constructor (lines 4–7) and the evolver (lines 9–11) use the private method `initializeTitle` to handle the initialization of field `title`. Section 6.2.1 below shows how such code duplication is avoided when default evolvers are used.

The body of an evolver could be different from that of the constructor. In particular, the constructor can make certain assumptions about the state of fields inherited from the superclass; it knows for certain that the superclass was only just constructed itself. An evolver, however, can execute long after the superclass instance was created, and the state of the inherited fields can vary from what it was after construction. The only valid assumption for the evolver is that the superclass fields maintain the superclass's class invariants.

For an example of a complex evolver, recall the `LinkedList` and `BidiLinkedList` examples from Figure 6.6. The class invariants of `LinkedList` are that (a) field `head` points to the first node; (b) each node's `next` points to the next node; and finally (c) the field `tail` points to the last node. In `BidiLinkedList`, the following invariants are added: (d) each node must be an instance of `BidiNode`, and (e) each node's `prev` points to the previous node, if any.

For an empty list, the extra invariants presented by `BidiLinkedList` hold trivially. Thus, the constructor of `BidiLinkedList` has to do nothing but call the inherited constructor (and indeed, the default constructor is used). The evolver, on the other hand, might have to operate on an instance of `LinkedList` that is already populated. It must therefore evolve each existing node to a `BidiNode` and correctly update its `prev` link.

The evolver presented in Figure 6.9 satisfies invariants (d) and (e) by iterating over the list (lines 4–9), evolving each `Node` instance to a `BidiNode` and properly setting its `prev` field.

---

delegation chain there must reside a constructor that begins with a call to **`super`**`(···)`.

```
1  public →BidiLinkedList() {
2     if (head == null) return;
3     head→BidiNode();
4     BidiNode current = head;
5     while (current.next != null) {
6        current.next→BidiNode();
7        current.next.prev = current;
8        current = current.next;
9     }
10 }
```

Figure 6.9: An evolver for `BidiLinkedList` (from Figure 6.6)

Default Evolvers

Classes in JAVA that define no constructor obtain a default constructor, generated by the compiler; this constructor merely invokes **`super()`**. It is an error to define a class with no constructor if its superclass has no constructor that accepts zero parameters (i.e., if the default constructor cannot call **`super()`**).

In a similar manner, classes that define no evolver obtain one or more *default evolvers*. For every constructor that begins with a parameter-less call to **`super()`**, (directly or, by a chain of **`this`**`(···)` calls, indirectly), a default evolver is generated. Each default evolver accepts the same parameters as the constructor that triggered its synthesis, and shares the same body, except the call to **`super()`**. The visibility level (**`private`**, **`public`**, etc.) of a default evolver is identical to that of the constructor that inspired its synthesis.

Default evolvers makes it possible to remove lines 9–11 (the evolver definition) from Figure 6.8; an identical default evolver would be automatically generated. This also means that, in the same figure, the `title` initialization code could be inlined as part of the constructor itself, rather than presented as a private method. Figure 6.10 shows the resulting definition of `Head`.

```
1  class Head extends Entity {
2     private String title;

4     public Head() { // Constructor
5        super(); // Call superclass constructor (could also be implicit)
6        // The following lines also serve as the default evolver →Head():

8        // Parse the node's content and set title accordingly:
9        Node titleElement = findSubElementByName("title");
10       title = titleElement.getTextContent();
11    }

13    // ... rest of the class not shown
14 }
```

Figure 6.10: Class `Head` (Figure 6.8) rewritten using an implicit evolver

If no default evolvers can be generated (because all constructors call **`super`**`(···)` with one or more parameters), then the class must define explicit evolvers. Class `BidiNode` in Figure 6.6 provides an example: it requires an empty evolver, but that evolver is not optional.

Evolution Steps

In the DOM tree example, class `Head` extends class `Entity`, which extends `Node`, which in turn extends `Object` (Figure 6.4). Therefore, whenever a new instance of `Head` is created, the constructor first invokes the constructor of `Entity`, which first invokes that of `Node`, etc. We have that the construction process always begins at the topmost level (`Object`) and progresses down in the inheritance tree towards the actual type (e.g., `Head`), with each step initializing its own fields and ensuring that its own invariants are maintained.

When an object is evolved, some nonempty prefix of this initialization chain had already occurred (at the very least, the `Object` constructor was executed). The evolution process must now ensure that the remaining tail is executed. Therefore, given the inheritance chain $C_n \prec C_{n-1} \prec \ldots \prec C_1 \prec C_0$, when object v is evolved from class C_0 to class C_n, it is not only the evolver of C_n that executes; the evolver of every class residing between the two in the inheritance tree runs first: $\rightarrow C_1$, followed by $\rightarrow C_2$, etc. These are *implicit evolution steps*. Because v's position in the inheritance chain (its dynamic type) is known only at runtime, the required implicit evolution steps are also known only at runtime. Only the final, explicitly named evolver $\rightarrow C_n$ is guaranteed to take place when an object is evolved to type C_n.

For example, when an instance of `Node` is evolved into an instance of `Head`, the evolver $\rightarrow$`Entity` runs first (an implicit step), followed by $\rightarrow$`Head`. This completes the initialization chain for a proper instance of `Head`.

We have seen that the evolution step might accept parameters. When the statement

$$v \rightarrow C_n(p_1, \ldots, p_k)$$

is executed (assuming v's current type is C_0), the parameters $p_1, \ldots, p_k$ are passed to the evolver $\rightarrow C_n$. For other evolvers in the chain between C_0 and C_n, a parameter-less evolver is used.

If an interim step in the evolution chain, $\rightarrow C_i$ for some $i \in \{1 \ldots n-1\}$, has no evolver that accepts zero parameters, the parameter-requiring steps cannot be implicit, and v may not be directly evolved to C_n. It must first be evolved to the interim step C_i, passing parameter(s) to one of C_i's evolvers; only then can it be evolved to C_n. If there are multiple such parameter-requiring steps in the chain between C_0 and C_n, then multiple explicit steps must be used.

Consider for example the trio of classes defined in Figure 6.11.

Given the variable declaration and initialization

```
A v = new A(0);  ,
```

the evolution statement `v→C(2)` will fail (at compile time), since `v` must first be evolved into an instance of `B` before it can become an instance of `C`, and this interim stage requires its own parameter. We must therefore use two explicit stages, as in

```
v→B(1);  // v's static type is now B
v→C(2);
```

If we know `v`'s dynamic type to be `B`, we can advise the compiler by using a downcast operator, writing `((B)v)→C(2)`. This will compile successfully, but (like all downcast operations), the downcast attempt might fail at runtime.

6.2.2 When `this` Evolves

Special attention must be paid to the case in which the current object, **`this`**, is evolved. This can happen by an explicit statement, **`this`**$\rightarrow C(\cdots)$; by evolving an alias of **`this`**; or by invoking some other method that takes either of these steps.

```
1  class A {
2    int a;
3    public A(int a) {
4      this.a = a;
5    }
6  }

8  class B extends A {
9    int b;
10   public B(int a, int b) {
11     super(a);
12     this.b = b;
13   }
14   public →B(int b) {
15      this.b = b;
16   }
17 }
```

```
18 class C extends B {
19   int c;
20   public C(int a, int b, int c) {
21     super(a, b);
22     this.c = c;
23   }
24   public →C(int c) {
25     // An object of type A cannot use this evolver
26     // directly, since an argument is required for
27     // the implicit step →B. An object of type
28     // B can use this evolver without a problem.

30     this.c = c;
31   }
32 }
```

Figure 6.11: An inheritance chain where each step requires an additional construction/evolution argument

After **`this`** is evolved, it could happen that it now has a new implementation of the currently-running method. For example, consider classes A and B from Figure 6.12.

```
1 public class A {
2   public void foo() {
3     System.out.print("A.foo1;");
4     this→B();
5     System.out.print("A.foo2;");
6     bar();
7     baz();
8   }

10  public void bar() {
11    System.out.print("A.bar;");
12  }

14  private void baz() {
15    System.out.print("A.baz");
16  }
17 }
```

```
18 public class B {
19   public void foo() {
20     System.out.print("B.foo;");
21   }

23   public void bar() {
24     System.out.print("B.bar;");
25   }

27   private void baz() {
28     // No overriding −− private method
29     System.out.print("B.baz");
30   }
31 }
```

Figure 6.12: Code that evolves **`this`**

When method `foo` of class A is executed, the current object is evolved (line 4) to class B, which overrides `foo`. Following the evolution, the A version of `foo` continues, despite the overriding.

When `foo` invokes `bar` (line 6), **`this`**'s new type causes the new version of `bar`, defined in B, to execute. Yet when `foo` invokes `baz` (line 7), it is A's version of `baz` that is executed, since `baz` is a **`private`** (hence statically bound) method. The invocation of `A.foo()` therefore generates the output "`A.foo1;A.foo2;B.bar;A.baz`".

6.2.3 Evolution Failures

Certain runtime circumstances can prevent an object evolution operation from completing successfully. Each of the three approaches presented in this work is susceptible to failure for different reasons, detailed in the appropriate sections below. The current section presents two possible causes for failure that are shared by all three approaches.

First, the evolution operation $v \rightarrow C$ will fail if v is **`null`** at the time of execution. As with similar cases in JAVA, the result is a `NullPointerException` being thrown. Such failures can be avoided using simple tests prior to the evolution operation, or with language extensions for non-null types [95, 170].

Second, the evolution operation can fail if the evolver throws an exception (directly or indirectly). When a constructor in JAVA throws an exception, the object creation process is aborted, and no object is returned by the **`new`** operator. Similarly, when an evolver throws an exception, the object evolution process is aborted, and the object retains its original type. Any side-effects caused during the evolution (e.g., output, the creation of additional objects, etc.) cannot be undone, just like the side-effects of a constructor that eventually failed by throwing an exception.

If the evolution required implicit steps, then steps that were completed prior to the throwing of the exception cannot be retracted. This limitation stems from the interim evolvers' ability to pass references to the current object, in its new (interim) type, to third parties. Thus, if an attempt is made to evolve an instance of `Node` to an instance of `Head`, and the evolver →`Head()` throws an exception after the implicit interim evolver →`Entity()` was successfully completed, then the object remains an instance of `Entity`.

Like any code member in JAVA, evolvers must obey the catch-or-declare principle, i.e., their signature must include a **`throws`** part specifying which checked exceptions might be caused by their execution.

It is possible to prevent evolution into some specific class by providing an evolver that unconditionally throws an exception. However, a simpler solution is to declare an empty evolver with **`private`** visibility. This way, the refusal-to-evolve will be detected at compile-time.

6.3 I-Evolution: Evolving within the Inheritance Tree

The most straightforward of the three approaches, I-Evolution allows an object v of static type C to evolve into any subclass of C. If C is an **`interface`**, then v can be evolved into any class that implements it.

The examples presented so far were all based on I-Evolution. This includes the replacement of the STATE design pattern with a simpler solution (Section 6.1.1), the DOM tree from Section 6.1.2, and the linked lists from Section 6.2.1.

6.3.1 Evolution to Mixin-Generated Classes

The target of an object evolution operation can be any class; in particular, it can be a class generated using a mixin. As an example, consider the mixin `Blocked` (Figure 6.13).[5]

This mixin can be applied to classes that implement the JAVA's standard interface `List`. The result is a list that cannot be modified, since any attempt to add or remove objects will yield an exception.[6]

[5] We use the syntax of JAM for defining mixins in our JAVA-like language; however, in order to remain consistent with features to be introduced in the following sections, the application of mixins is expressed using a generics-like syntax, i.e., $M\ \langle C \rangle$ is the application of mixin M to class C.

[6] A true blocking of JAVA's standard `List` interface will in fact require overriding many more methods; the mixin presented in Figure 6.13 is greatly simplified.

```
mixin Blocked {
   inherited public void add(Object o);
   inherited public void remove(int index);

   public final void add(Object s) { // Override inherited version
      throw new IllegalOperationException();
   }

   public final void remove(int index) { // Override inherited version
      throw new IllegalOperationException();
   }
}
```

Figure 6.13: A mixin for creating immutable versions of classes that implement interface `List`

Using object evolution, list objects can be evolved into blocked-list objects at any stage of their life. For example, the following code can be used:

```
List myList = new Vector();
myList.add(···); // add numerous data items
myList→Blocked<Vector>();
```

Here, applying the mixin to class `Vector` generates a new class that refuses to add new items or remove old ones. Once the evolution completes, no client that holds a reference to this list object will be able to alter its content. There are many uses to this capability, including security considerations and improved performance for defensive programming [29, Item 24] (since there is no need to create a copy of the list).[7]

6.3.2 I-Evolution Limitations

I-Evolution offers great flexibility, since the target of the evolution operation can be either a standard class or a mixin-defined class. In contrast, M-Evolution and S-Evolution, the other two approaches, cannot use standard classes (such as `Head`, `InterfaceMethodInvocation`, etc.) as the evolution target. I-Evolution's main limitation, however, is that change must be *down a pre-determined path*, i.e., it can only propagate down the statically-defined inheritance tree. In the TCP connection example, once a connection object reaches the closed state, it is in what we may metaphorically term "an evolutionary dead-end" [120, Chap. 5]; it can no longer change its state. To represent a fresh connection, a new `TCPConnection` object must be created. As we shall later see, S-Evolution can be used to overcome this limitation in many cases, including this particular one.

Like all evolution approaches, I-Evolution has its own specific cases of potential evolution failure. When evolving object v of static type C_0 to some other type C, I-Evolution will fail if v's dynamic type happens to be C', which is a subclass of C_0 but not a superclass of C. Such failure results in a `ClassEvolutionException`.

For example, consider the following code, using the DOM class hierarchy presented earlier

[7] It is for these security considerations that the methods in mixin `Blocked` were defined as **`final`**—to prevent the application of a reverse mixin, "`Unblock`".

(Figure 6.4):

```
Node n = getSomeNode();
n→Head();
```

If `getSomeNode()` returns an instance of the specific class `Node`, the evolution will succeed. Likewise if it returns an instance of `Entity`, which is a subclass of `Node` and the direct superclass of `Head`. However, if the invocation returns an instance of `Text` (a subclass of `Node` which is unrelated to `Head`), the evolution attempt will fail. (Consider that some third party might have a reference to the `Text` object, and that reference's static type is `Text`; had we allowed the object to evolve into the unrelated `Head` class, messages from the `Text` class will no longer be understood.)

The special case of the evolution operation $v \rightarrow C(\cdots)$, where v's dynamic type already is C, also deserves consideration. While it might seem that this operation is "quiescent", and should be allowed to complete by doing nothing, it will in fact fail. It must fail, because the code that includes this operation expects any side-effects (e.g., output), caused by the evolver of class C, to happen at this point; and the evolver itself cannot be re-executed, for the same considerations that prohibit the re-execution of a constructor on an already-constructed object. We suppose that in languages that do allow object re-construction, quiescent object evolution is also possible.

6.4 M-Evolution: Evolving with Mixins

M-Evolution is a variant of object evolution, where the target of any evolution statement is the result of applying a mixin to the *runtime* type of an object. An M-Evolution statement for variable v uses the syntax $v \rightarrow M\ \langle v \rangle\ (\cdots)$, where M is a mixin. The operation selects $M\ \langle V \rangle$ as its target, where V is v's runtime type. If class $M\ \langle V \rangle$ did not previously exist, the evolution operation will cause it to be generated, at runtime. M-Evolution therefore avoids the "evolutionary dead-end" limitation of I-Evolution by dynamically extending the inheritance tree.

To understand the usefulness of the concept, consider mixin `Blocked` (Figure 6.13) again. While it can be used to generate a subclass of any class that implements `List`, it is hardly useful in a context where all we have is an instance whose static type is `List`, and its dynamic type unknown. This is a common case, for example, with methods that accept a `List` reference as a parameter. Should such a method wish to evolve its parameter to an immutable object using `Blocked`, it can try to evolve it into `Blocked<ArrayList>`, `Blocked<Vector>`, or any of numerous other combinations (see Figure 6.14); none however is guaranteed to succeed, since the total number of classes that implement `List` is unbounded.

The solution is to apply a mixin to the runtime type of the object. The method `blockParam` in Figure 6.15 does just that.

As can be seen in Figure 6.15, mixin `Blocked` accepts as a parameter not a type, but a variable; it generates a new class, at runtime, based on that variable's dynamic type. The resulting type of the variable after the evolution statement can be e.g., `Blocked<Stack>`, `Blocked<LinkedList>`, etc.

6.4.1 M-Evolution and Idempotent Mixins

A moment's reflection will reveal that the evolution statement in Figure 6.15 can never fail, except in certain rare scenarios described below. In most cases evolution will succeed since no matter where in the inheritance tree does the variable's runtime class reside, it can evolve downwards. There is no evolutionary dead-end to reach, since the inheritance tree is expanded at runtime by class generation. In particular, even if the type is already the result of applying the `Blocked`

```
public void blockParam(List lst) {
   if (lst instanceof Vector)
      lst→Blocked<Vector>(); // Attempt I−Evolution
   else if (lst instanceof ArrayList)
      lst→Blocked<ArrayList>(); // Another I−Evolution attempt
   else if (lst instanceof LinkedList)
      lst→Blocked<LinkedList>(); // ... etc.
   else // ... etc.
   else throw new RuntimeException("Unknown List implementation.");
}
```

Figure 6.14: A method that attempts to use I-Evolution for applying a mixin to a reference. (Figure 6.15 shows a superior alternative, using M-Evolution.)

```
public void blockParam(List lst) {
   lst→Blocked<lst>(); // M−Evolution
}
```

Figure 6.15: A method that uses M-Evolution, applying a mixin to a reference's dynamic type (*cf.* Figure 6.14)

mixin, it can further evolve; the type can change, e.g., from class `Blocked<Vector>` to class `Blocked<Blocked<Vector>>`. No complication is introduced by the repeated application of the mixin, since it is *idempotent*.

Another example is provided by the classic `Undo` mixin, reproduced in Figure 6.16.

```
@Idempotent public mixin Undo {
   inherited public String getText();
   inherited public void setText(String s);

   private String lastText;

   public void setText(String s) {
      lastText = getText();
      super.setText(s);
   }

   public void undo() {
      setText(lastText);
   }
}
```

Figure 6.16: A sample mixin, which adds an `undo` method to classes that have a property called `text` with appropriate getter/setter methods (based on [8, Fig. 1])

Mixin `Undo` can be applied to any class that features the two methods `getText` and `setText`, such as the `JButton` class from JAVA's standard library. The ability to repeatedly apply `Undo` (generating, e.g., `Undo<Undo<JButton>>`) with no adverse effect is less obvious,

since every such application introduces a new field (`lastText`), so that the mold for object creation (see Section 5.2) is changed. Also, every repeated application will add a new invocation to the chain of operations that implement `setText`. However, other than by means of performance measurement, external clients have no way to tell an instance of `Undo<Undo<JButton>>` from an instance of `Undo<JButton>`; the behavior remains identical. We therefore maintain that this mixin is also idempotent, and mark this in the source code using the `@Idempotent` annotation.

We say that a mixin is idempotent if:

1. It is annotated using `@Idempotent` (e.g., mixin `Undo` from Figure 6.16), or
2. It meets both of the following criteria (e.g., mixin `Blocked` from Figure 6.13):
   (a) It introduces no new members (fields or methods), with the possible exception of **`private`** members, and
   (b) Any method that it overrides is replaced rather than refined (i.e., the new method body does not call the inherited version using **`super`**).

Given an idempotent mixin M_I and arbitrary type T, the runtime system will always provide $M_I\,\langle T\rangle$ when asked to generate $M_I\,\langle M_I\,\langle T\rangle\rangle$.

The case of applying the idempotent mixin M_I to an object of type $M_x\,\langle M_I\,\langle T\rangle\rangle$, where M_x is some other mixin, is discussed in Section 6.5.1.

This approach prevents the creation of unnecessarily long "threads" in the inheritance tree, that might result from the repeated application of a single idempotent mixin to the same object. Replacing $M_I\,\langle M_I\,\langle T\rangle\rangle$ with $M_I\,\langle T\rangle$ maintains static correctness, since objects of both types can be assigned to variables of static types M_I, T or $M_I\,\langle T\rangle$, and will pass the same **`instanceof`** tests.

6.4.2 M-Evolution and Non-Idempotent Mixins

M-Evolution's ability to avoid failure by dynamically extending the inheritance tree is not limited to idempotent mixins. Consider, for example, the mixin `Track` from Figure 6.17.

```
public mixin Track {
   inherited public String getText();
   inherited public void setText(String s);

   public void setText(String s) {
      System.out.println("Changing the text to " + s);
      super.setText(s); // Refinement -- therefore, the mixin is not idempotent
   }
}
```

Figure 6.17: A mixin that reports any change to the `text` property. Method `setText` is implemented as a refinement, making the mixin non-idempotent

The two classes `Track<JButton>` and `Track<Track<JButton>>` will behave differently, one reporting once and the other reporting twice every change to the `text` property. The mixin is therefore not idempotent, yet nothing prevents its repeated application. In particular the M-Evolution statement $v \rightarrow$`Track<`v`>` will (almost) never fail, as long as v's static type includes the pair of methods required by `Track`. The evolution tree will be repeatedly extended as much as needed.

In JAM and similar languages, mixins (with no specific superclass provided) can be used as types in their own right. Variables of a mixin type can be defined, and the type can also appear in **`instanceof`** statements. Thus, structures such as:

```
if (!(button instanceof Track))
   button→Track<button>();
```

can be used to prevent undesired repeated application of non-idempotent mixins to objects.

6.4.3 M-Evolution Limitations

By restricting ourselves to properly crafted mixins, we find that M-Evolution will not normally fail, since it extends the inheritance tree as needed. Two esoteric cases, however, can cause evolution failure for M-Evolution.

- *Failure by final declarations.* This risk relates to leaves in the inheritance tree that are, by their own admission, dead ends that cannot be extended. Mixin application will fail whenever it attempts to override a **`final`** method, or any method in a **`final`** class. For example, method `blockParam` from Figure 6.15 will fail in its M-Evolution attempt if its parameter `lst` is an instance of a **`final`** class that implements `List`.
- *Failure by accidental overriding.* Accidental overriding [8] relates to cases where a mixin attempts to introduce a new method, only to find out that a method of this signature already exists in the superclass. From program correctness considerations, this must not come to pass, and such mixin applications will fail. We thus have that the M-Evolution statement $v \rightarrow M\,\langle v \rangle\,(\cdots)$ will fail if mixin M cannot be applied to v's runtime type due to accidental overriding.

As an example for failure by accidental overriding, consider Figure 6.18, showing a class `UndoableJButton` inheriting from `JButton`.

```
public class UndoableJButton extends JButton {
   public void undo() {
      // ... undoes any change to the button's size, color and
      // position, but not changes to its text caption.
   }
}
```

Figure 6.18: A subclass of `JButton` with its own `undo` method

The application of mixin `Undo` to class `UndoableJButton` will fail, since mixin `Undo` will accidentally override method `undo()`. Thus, in the following code fragment:

```
JButton button = getSomeButton();
button→Undo<button>();
```

the M-Evolution attempt will fail if method `getSomeButton` happens to return an instance of `UndoableJButton`.

It might seem as if mixin `Undo` cannot be applied repeatedly to the same object. If `getSomeButton` returns an instance of `Undo<JButton>` (i.e., an instance to which the mixin was already applied), then the object's runtime type already has an `undo` method (created by the first mixin application), and the second application will cause accidental overriding. However, since `Undo` was marked as idempotent, an attempt to generate `Undo<Undo<JButton>>` will simply yield `Undo<JButton>`.

6.5 S-Evolution: Evolving with Shakeins

S-Evolution is a variant of object evolution, where the target of any evolution statement is the result of applying a *shakein* to the *runtime* type of an object. An S-Evolution statement for variable v uses the syntax $v \rightarrow S\,\langle v \rangle\,(\cdots)$, where S is a shakein.

Much like M-Evolution, S-Evolution extends the inheritance tree as needed at runtime, and therefore cannot fail due to inheritance dead-ends (except when the inheritance tree cannot be extended due to **`final`** classes or attempts to override **`final`** class members). Also like M-Evolution, shakeins can be marked idempotent making their repeated application a "fail-safe" operation; shakein `ReadOnly` (Figure 6.19) is an example of an idempotent shakein. And, because shakeins cannot introduce new non-**`private`** class members, they are not susceptible to failure by accidental overriding.

```
1  @Idempotent public shakein ReadOnly {
2     pointcut setter := void set[A-Z]?*(_);

4     around: setter {
5        return; // Block silently; do not invoke original version
6     }
7  }
```

Figure 6.19: The idempotent `ReadOnly` shakein blocks all setter methods

6.5.1 Shakein State-Groups

Shakeins and S-Evolution can substitute the STATE design pattern, since in this pattern, all state classes implement the same interface. For example, state classes `TCPListen`, `TCPEstablished`, and `TCPClosed` all implement the interface defined by the abstract class `TCPState` (Figure 6.1); we have a set of classes that share the same type. Such sets can also be generated by applying different shakeins to the same base class.

We define a *state-group* of shakeins as a set of shakeins that share the `@StateGroup` annotation, with the same string parameter; different, independent state-groups can be created using different string parameters. For example, the three shakeins in Figure 6.20 form the state-group *`"Connection"`*.

The compiler enforces the limitation that all shakeins in a given state-group must define the same set of **`private`** class members.[8] In the example, this requirement is met vacuously.

Shakeins in the same state-group are *mutually exclusive* in the following sense: If C is an arbitrary class, and shakeins S_1 and S_2 are in the same state-group, the application of S_1 to $S_2\,\langle C \rangle$ yields $S_1\,\langle C \rangle$, rather than $S_1\,\langle S_2\,\langle C \rangle \rangle$. Such an application is called a *state transition*.

When applied to class `TCPConnection` from Figure 6.2, `Listen`, `Established` and `Closed`, the three shakeins from Figure 6.20, generate the state subclasses. In particular, `Established<TCPConnection>` is equivalent to class `TCPConnectionEstablished` (from Figure 6.2), and `Closed<TCPConnection>` is equivalent to `TCPConnectionClosed`. These shakeins capture the increment between `TCPConnection` and each of its subclasses, but use S-Evolution statements (lines 4, 8 and 20).

This state-group can overcome the inability of the I-Evolution-based solution to retract its steps (Section 6.3.2). Given a connection in the "closed" state, we can now change its state back

[8] Recall that shakeins can never introduce non-**`private`** class members, since they may not change the base class's type.

```
1  @StateGroup("Connection") public shakein Listen {
2     public void open() {
3        // ... establish the connection ...
4        this→Established<this>();
5     }

7     public void close() {
8        this→Closed<this>();
9     }

11    public void acknowledge() { ... }
12 }

14 @StateGroup("Connection") public shakein Established {
15    public void open() {
16       // ... ignore
17    }

19    public void close() {
20       this→Closed<this>();
21    }

23    public void acknowledge() { ... }
24 }

26 @StateGroup("Connection") public shakein Closed {
27    public void open() {
28       throw new IllegalStateException();
29    }

31    public void close() {
32       throw new IllegalStateException();
33    }

35    public void acknowledge() { ... }
36 }
```

Figure 6.20: A shakeins state-group for generating the various state classes of `TCPConnection` (*cf.* the limited I-Evolution version in Figure 6.2)

to "listen" by applying the `Listen` shakein to its dynamic type. Doing so will change the object's type from `Closed<TCPConnection>` to `Listen<TCPConnection>`. There is no limit on the number of times the state can be changed by re-applying the appropriate shakein; and these changes do not generate an inheritance tree of unbounded depth.

A note on terminology: While state transition is not, strictly speaking, a move down the inheritance tree, it is still a form of object evolution, because conceptually the new state *could* be defined as a subclass (but not a subtype) of the old one. The fact that it is not a subclass is a means for avoiding needlessly long inheritance "threads". We use the term "transition" only for this special form of S-Evolution.

Transitions within a state-group are type safe, because the type of a shakein-applied object, $S_1\langle C\rangle$, is *identical* to that of $S_2\langle C\rangle$, and to that of C itself. Shakein $S_2\langle C\rangle$ recognizes all messages that $S_1\langle C\rangle$ recognized (and vice versa). The only fine point is the type of **`this`** *in methods overridden by the shakein application.* Such methods may call **`private`** methods, or access **`private`** data members, defined in S_1. Hence the requirement that all shakeins in a given state-group include the exact same set of private class members.

The trivial case of a state-group with only a single shakein (or mixin) is in fact equivalent to an idempotent shakein; given such a shakein S_I, applying it to $S_I\langle C\rangle$ yields $S_I\langle C\rangle$. However, state-groups with more than one member can only be defined for shakeins, and not for mixins, because the mixin type itself can be used to define variables. For example, had the three shakeins from Figure 6.20 been created as a set of mixins, then some code could have defined a variable of these types, as in:

```
Established e = new Established<TCPConnection>();
```

Now, by the definition of evolution within state-groups, evolving the object to mark a closed connection would change its type to `Closed<TCPConnection>`; yet the variable `e` cannot refer to such an object. The problem is never encountered with shakeins, since no variables (or fields, etc.) of shakein types are possible.

Objects with Multiple States

Multiple shakeins can be applied to a single object. A shakein applied to a class may override methods; if the set of methods overridden by shakein S, and the set of methods overridden by shakein R, are completely disjoint, then the order of shakein application does not matter; $S\langle R\langle C\rangle\rangle$ and $R\langle S\langle C\rangle\rangle$ are indistinguishable, and we say that S and R and *commutative with respect to class* C.

Commutativity comes into play when multiple shakeins from different state-groups are applied to a single object. For example, given state-group $\mathbf{S}$ with states S_1 and S_2, and state-group $\mathbf{R}$ with states R_1 and R_2, one can create classes such as $S_1\langle R_1\langle C\rangle\rangle$, $R_2\langle S_1\langle C\rangle\rangle$, etc., depending on the shakeins used and the order of application.

What happens when we apply shakein R_2 to an instance of $S_1\langle R_1\langle C\rangle\rangle$? There are four possible semantics for handling such cases:

1. *Error semantics.* The application of R_2 results in a runtime error.

2. *Accumulation semantics.* The result is $R_2\langle S_1\langle R_1 C\rangle\rangle$, i.e., the new shakein is added to the object and does not replace the old shakein from the same state-group (no state transition takes place).

3. *Order-preserving semantics.* The result is $S_1\langle R_2\langle C\rangle\rangle$, i.e., the state-transition preserves the order in which shakeins from the different state-groups were originally applied.

4. *Re-ordering semantics.* The result is $R_2\langle S_1\langle C\rangle\rangle$, i.e., the state-transition re-orders the shakeins applied to the object.

Accumulation semantics implies that the application results in an object which is simultaneously in states R_2 and R_1; this situation is not desired since R_2 and R_1 belong to the same group, which usually implies that they are mutually exclusive. We therefore eliminate this option.

If R_1 and S_1 are commutative with respect to C, then both order-preserving and re-ordering semantics yield the same result. More generally, we say that $\mathbf{R}$ and $\mathbf{S}$ are *commutative state groups* with respect to C if for all $R \in \mathbf{R}$ and $S \in \mathbf{S}$, R and S are commutative with respect to C. In

such cases, the choice between the two semantics is of no practical importance. We can therefore choose either semantics if the two state-groups are commutative, or report an error if they are not. However, since the actual parameter passed to the shakein application is a variable (and not a class), commutativity cannot be statically determined.

Either of the two order-related semantics, re-ordering and order-preserving, however, works without resorting to runtime errors. We now present our reasoning for choosing the former over the latter.

If $\mathbf{R}$ and $\mathbf{S}$ are not commutative, there exists a method m in class C which is overridden by some shakein from $\mathbf{S}$ as well as by some shakein from $\mathbf{R}$. The order of shakein application dictates which version of m will be used. We say that the shakein (or state), from which the implementation of m is chosen, has *taken precedence* over other alternatives.[9]

Order-preserving semantics implies that the order in which the shakeins were originally applied to the object cannot be altered. Applying shakein R_2 to an object that already has a state from state-group $\mathbf{R}$ replaces the object's $\mathbf{R}$-state, but any states from other state-groups that were applied after the original $\mathbf{R}$-state still take precedence.

Conversely, re-ordering semantics implies that the most recent order of shakein application prevails. Applying shakein R_2 to an object that already has a state from state-group $\mathbf{R}$ replaces the object's $\mathbf{R}$-state, while taking precedence over any states from other state-groups that were applied after the original $\mathbf{R}$ state.

The *principle of least surprise* [193, Sec. 11.1] dictates that re-ordering semantics is preferable. Having a previously-applied shakein take precedence over the most recently-applied one (as in the case of order-preserving semantics) can lead to awkward situations. For example, if shakein R_2 is `ReadOnly` (Figure 6.19), failure to take precedence implies that applying `ReadOnly` to an object might yield a *non*-read-only object. On the other hand, it *does* make sense that applying some shakein S to a read-only object might make that object non-read-only, depending on the nature of S.

This language design decision also resolves the subtlety we raised in Section 6.4.1 in applying an idempotent mixin M_I to an object of type $M_x \langle M_I \langle C \rangle \rangle$: The result, using re-ordering semantics, is $M_I \langle M_x \langle C \rangle \rangle$.

6.5.2 Shakeins as Dynamic Aspects

Chapter 2 introduced shakeins as a more organized, parameterized alternative to aspects. As noted before, a shakein can apply advice (`before`, `after` or `around`) to methods of the inherited class, yielding a new implementation of the base class's type without changing the class itself. One example is the shakein `Log` from Figure 6.21, which can be used as a logging aspect. It accepts a string parameter, which is the filename to which the log will be written. Logging objects can be created using statements such as:

```
List verboseList = new Log["list.log"]<Vector>();
```

Now, any access to a public method of the `verboseList` object will be logged in the specified file.

With object evolution, we can dynamically apply this aspect to existing objects. For example, a method that accepts some parameter `lst` of type `List` can issue the statement:

```
lst→Log["system.log"]<lst>();
```

[9] An object with multiple states can be likened to a statechart [130] with several AND-states. However, in objects, a message m is intercepted by *one* method implementation (the most recent overriding version of m). In contrast, an event e applied to a statechart with several AND-states can be processed by any interested state simultaneously; there is no concept of a "most recent" state, and no precedence issues.

```
@StateGroup("Log") public shakein Log[String filename] {
   pointcut publicMethod = public (*);

   before: publicMethod {
      FileOutputStream fos = new FileOutputStream(filename);
      // ... log the operation to the file ...
      proceed;
   }
}

@StateGroup("Log") public shakein NoLog {
   // No changes to the base class
}
```

Figure 6.21: A dynamic logging aspect, defined as a shakein and its canceling counterpart

and evolve the list into a logging list.

The use of state-groups allows the dynamic aspect to be removed as well, given an empty shakein from the same state-group:

```
lst→NoLog<lst>();
```

(shakein `NoLog` is an empty shakein, defined in Figure 6.21).

Other implementations of dynamic aspects (e.g., JBoss's dynamic AOP) use a flag, or a list of active aspects, that should be consulted every time a method is invoked if the receiving object can be the target of dynamic aspects. The evolution-based approach offers a different tradeoff, where method invocation is faster (no flag or list to consult), but the application or removal of aspects is potentially slower. The following section discusses possible implementation strategies for object evolution, and their associated performance cost.

6.6 Implementation Strategies

To support object evolution, the runtime system (e.g., Java Virtual Machine) must be able to (a) generate classes at runtime (for M- and S-Evolution), and (b) change an existing object's type.

Since standard JVMs support runtime class loading, various technologies for runtime class generation exist, and are used in mainstream applications such as the Spring Application Framework [146], Hibernate [22], and iBATIS [12]. J2EE application servers also generate classes at runtime as part of the application deployment process [202].

More difficult is the issue of changing an object's class at runtime.

We distinguish between two cases of object evolution. In the simpler case, the evolution process does not change the class's mold, i.e., the target class does not define any new fields. Here, the type-change can be realized by changing the object's VMT pointer.[10] The operation is less straightforward, however, in the more general case, where new fields are introduced by the target class, and the object's representation in memory must therefore expand.

A simple approach would be to allocate a new object and initialize its fields with the field values of the object being evolved. Next, all references to the old object must be updated so that they now point to the new one. To this end, the virtual machine must be able to locate all

[10] Interestingly, such changes happen during the object construction process in C++.

references to a given object. Version 6 of the JAVA SE [213] includes the required "heap walking" capabilities [214] as part of the virtual machine's debug interface (JDI). However, we suspect that this approach might be inefficient.

Optimizing the performance of a solution requires empirical data about the frequency and type of object evolution operations that are common in programs. Below, we outline different approaches, each appropriate for different usage scenarios, which can be used when such data becomes available.

The discussion assumes that object v is evolved from class A to class B. The size required for an object of class X is denoted $|X|$.

6.6.1 Using Object Handles

Certain memory-management systems that support automated garbage collection store, in each reference variable, a pointer to a *forwarding pointer* [42], or *handle*, rather than a direct pointer to the object itself. The handle itself is never moved, but it points to the actual object location, which can change repeatedly during the object's lifetime.[11].

Normally, objects are re-located in memory due to the *compaction* part of the garbage-collection process [147]. However, a similar a re-location process can also realize object evolution, as follows:

1. Allocate a new block β of size $|B|$,
2. Copy v's original content to β,
3. Update v's handle so that it points to β,
4. Mark v's old location as empty space.

The main drawback of using object handles is that every field or method access involves a double de-referencing operation. Clearly, a JVM-level implementation of this double de-referencing would be faster than the double de-referencing implied by wrapper-based implementation, such as the one employed for FICKLE [86]. Optimization techniques presented in the Metronome JVM [17] indicate that double de-referencing can be optimized to a low (4%) performance overhead.

6.6.2 Compacting Evolution

The main benefit of the handles-based approach is that it ensures a constant-time, and normally very fast, evolution operation. One may assume, however, that object evolution is a relatively rare operation in the program's life cycle. We present now an alternative that focuses on a lower environmental impact, while the object evolution operation itself might be relatively costly.

In the *compacting evolution* approach, every object evolution operation forces a complete garbage collection run, including memory compaction. During this compaction, the system leaves a "padding" of $|B| - |A|$ free bytes following object v. Once the compaction completes, object v's size can be increased to $|B|$ by claiming the padding bytes.

If evolution operations are indeed scarce, the performance impact of this approach will be small, since the forced garbage collection replaces the next scheduled one.

A performance problem could arise if evolution operations are more frequent than scheduled garbage collection runs. This is the case, e.g., in Figure 6.9, where evolution is performed inside a loop.

[11] Such a mechanism is used to support reference replacement in GILGUL's virtual machine [72]

Two schemes might mitigate this. First, since loops can be detected by the JVM, compacting evolution can be modified so that the forced compaction run pads every instance of class A with $|B| - |A|$ bytes whenever an object of class A is evolved inside a loop; this ensures that a forced garbage collection will be required only for the first iteration. The unused padding will be removed by the next regularly-scheduled memory compaction.

Alternatively, a reference-counting garbage collector can be used. A known property of such collectors is that the collector's runtime is relatively short if memory usage has not radically changed since the last collection cycle [161]. It therefore addresses the more general case of frequent evolution operations, and is not limited to evolution operations inside loops.

6.7 Related Work

6.7.1 Monotonic Reclassification in the Literature

The idea of improving the type safety of object reclassification by making the changes monotonic is not new. Back in 1993, Beck presented the SCRIPTABLE OBJECTS design pattern [24], allowing SMALLTALK objects to gain *instance-specific behavior* by changing their own methods, using a method called `specialize:`. This is a trivially monotonic change, in the sense that it does not alter the object's interface (the set of acceptable messages).

Object extension [107, 109] encompasses non-trivial monotonic changes. One possible kind of changes is the dynamic extension of an object (known as *self-inflicted extension*), with no prescribed template serving as the object's new class. In statically typed systems, this implies that only the object's own methods can access any newly introduced method or data member.

Ghelli [107] suggested a calculus in which what he called "incompatible changes" cannot occur, by letting the same object assume different roles in different contexts. His work is done in the context of FIBONACCI [3], a database language. Indeed, roles are more natural to object-oriented databases than to object-oriented programming languages. The reason is that the notion of a dynamic type of an object may conflict with the "role" imposed on it. For example, failures in I-Evolution are not necessarily a result of incompatible changes.

Systems that do offer non-monotonic, type-safe object reclassification restrict the choice of objects that can be reclassified, and the range of reclassification target classes. In Serrano's *wide classes* system [201], objects can only be *widened* to instances of wide subclasses of their current class. In other words, what we call evolution is restricted to a pre-designated set of subclasses. Conversely, only wide objects (instances of wide classes) can be *shrunk*. The ability to shrink objects implies that the system cannot be used in statically-typed languages [201, Sec. 3.6].

6.7.2 The Work on FICKLE

The FICKLE programming language (and its newer versions $\text{FICKLE}_{\text{II}}$ [87] and FICKLE_3 [77]) are more general than evolution, since they allow non-monotonic reclassification. For example, a frog that turns into a prince in FICKLE may forget how to kiss. However, reclassification in FICKLE is restricted in the sense that only instances of designated *root* classes can be reclassified, and the destination class must be a pre-designated *state* subclass of the root class. When moving from one state-class to another an object sheds any fields and methods not included in the root class.

To maintain static typing, the FICKLE type system must track the *effect* of each method, namely the list of classes whose instances may be reclassified (directly or indirectly) by the method. To avoid whole system analysis, the programmer is requested to annotate each method with the list of reclassification changes it may make. The type verifier then checks that this annotation is compatible with the actual changes done by the method itself, and with the annotations of any called method. Also, the verifier makes sure that the effects list is not increased in the

course of overriding; the programmer therefore should include in the effects list of a method all anticipated actions of reclassification in overriding methods.

To understand how the effects list is used to ensure type safety, consider again the frog and prince example, which is modeled in FICKLE as an `Actor` root class, with two state classes `Prince` and `Frog`. After calling a method in which a frog may be kissed, *all* variables of type `Frog` effectively change their static type into `Actor`. To generalize, after method with an effect on root class C is invoked, all variables of state class S ($S \prec C$) must be treated as instances of the root class, and fields or methods declared in S become inaccessible. The reason is that, due to aliasing, any one of these variables may just have been reclassified, and no longer has the additional features of S. For the same reason, state classes cannot be used to define variables, although this restriction was somewhat eased in FICKLE$_{\text{II}}$ (where state classes are only prohibited from being used to define fields).

The set of classes composed of a root class and its state subclasses in FICKLE can be compared to a class and the set of re-implementations that can be derived from it using a shakein state group. In both cases, objects can be reclassified freely inside the group. Also, to maintain type safety, in both cases there are restrictions on using the non-root classes as types. In FICKLE$_{\text{II}}$, these restrictions are relaxed and pertain to fields only, whereas in our system they are absolute. Another advantage of FICKLE state classes is that, unlike shakeins, they can introduce externally-accessible, non-**`private`** members. Perhaps most importantly, FICKLE is strongly-typed, i.e., the reclassification operation cannot fail.

On the other hand, shakeins integrate into the existing type system without tracking the effect of each method, and without requiring the class serving as root to be marked as such. In FICKLE, any subclass (direct or indirect) of a root class must be a state class, whereas with shakeins, regular classes, that can have regular subclasses, are used as roots. Thus, shakeins can be used to create a state group from existing classes within a pre-defined hierarchy, such as the standard library.

Interestingly, FICKLE$_3$ removed the requirement of pre-declaration of root and state classes. In FICKLE$_3$, any object may change its class to any other class, just as with SMALLTALK's `becomes:`. Roughly speaking, FICKLE$_3$ computes the root and the state classes from the reclassification annotations, so if a method reclassifies an object of type A into type B, then after this method is called, all objects of any type A', $A' \preceq A$ are suspects of turning into a B. Therefore, after such a method is called, only features common to A and B are accessible in such objects. One may say that the method call changed the *static type* of all such variables from A' to the least common superclass of A and B.

FICKLE and FICKLE$_{\text{II}}$ were already implemented [7] by translating them into JAVA. The implementation of FICKLE$_3$, in which any object may be reclassified into any class, should deal with the need to wrap all objects as explained above in Section 6.6.

As noted before, object evolution is not as type safe as all versions of FICKLE, and may generate runtime errors. On the other hand, object evolution does not burden the programmer with writing the reclassification annotations of methods. Also, the difficulties of unrelated variables loosing their static type after a call to a reclassifying method do not occur with object evolution.

Another advantage of object evolution is its inherent thread-safety. In FICKLE, a reclassification operation can change the type of **`this`** in other threads, causing it to lose some of its associated methods (those that were introduced in its previous state class and are not available in the new one). With object evolution, since the change is purely monotonic, this problem does not arise.

FICKLE$_{\texttt{MT}}$ [76], the newest version of FICKLE, addresses the issue of thread-safety by introducing an elaborate system of locks, that prevents any object that might be reclassified in one thread from being reclassified, or even used as a receiver, in another thread. This locking system manages a dynamic dictionary of reclassification candidates of active threads, and delays any other

thread that tries to send a message or reclassify one of these candidates.

6.7.3 Object Replacement

Related to reclassification and evolution but different is *object replacement*, as offered by the GILGUL [72] language. GILGUL extends JAVA in allowing a global change of all references to a certain object, redirecting them to another object. Static type safety is maintained by the requirement that the type of the new object must be the same as that of the type of the original object, or a subtype thereof. Thus, replacement must deal with the same kind of runtime failures that might occur with I-Evolution.

GILGUL also introduces *implementation-only*, or *typeless*, classes. A typeless class extends a regular class (directly or, via other typeless classes, indirectly) without introducing a new type into the system. In particular, like shakein-generated classes in our system or state classes in FICKLE, no variables of a typeless class can be defined in GILGUL. Instances of a typeless class C can then be replaced by instances of any class having the same least non-typeless superclass.

6.8 Summary

This chapter propounded the inclusion of object evolution mechanisms into mainstream programming languages, and in particular JAVA. We showed how this mechanism integrates well with the popular mechanism of inheritance, and with the less popular, but still appealing mechanisms of mixins and shakeins.

Our language design decisions favored type-safety, but in respect of practical concerns, not with as much zeal as other approaches to object reclassification. To our knowledge, this is the first attempt to reconcile in this manner the conflicting just purposes of type-safety, reclassification flexibility and practical concerns.

In the context of the current work, i.e., employing aspect-oriented programming for the development of better middleware frameworks, object evolution provides the last piece of the puzzle, namely the ability to use shakeins as dynamic aspects.

Chapter 7

Summary

Don't fear failure so much that you refuse to try new things.

— Louis E. Boone

Middleware frameworks, the software systems used to develop enterprise applications, command a multi-billion dollar market [78, 228]. In this thesis, we have presented a manner in which the benefits of aspect-oriented programming can be ushered onto middleware frameworks, with little or no risk of disturbing existing code. **The mechanisms presented here have *the potential* of improving the modularity, and reducing the development costs, of enterprise applications.**

Four suggested extensions to object-oriented languages were presented in this work:

Shakeins allow for an orderly application of advice to classes, thereby providing the benefit of aspects without requiring any change to the underlying object model;

Factories provide classes with complete control over their instantiation process, including the ability to mandate shakein application;

JTL is a query language that enables a more expressive selection, by shakeins, of target points for advice application; and

Object Evolution allows instances to change their class at runtime without forfeiting strong typing, and enable the use of shakeins as dynamic aspects.

While JAVA was used as the base object-oriented language when presenting these extensions, nothing binds them specifically to JAVA, and each is equally applicable to other languages such as C#, EIFFEL, etc. Three out of the four suggested extensions can be added to JAVA with no change to the underlying virtual machine, and should be equally simple to apply elsewhere.

We have also presented **ASPECTJ2EE** as an outline for a possible integration of the suggested extension into existing Enterprise JAVA applications and frameworks.

It is interesting to note that if applied in union, the mechanisms suggested in this work render several classic design patterns superfluous, while improving others. Of the 23 original Gang of Four patterns [105], over a third are affected:

1. ABSTRACT FACTORY is superfluous, since any *supplier-side factory* "provide[s] an interface for creating ... objects without specifying their concrete classes."

2. FACTORY METHOD is superfluous, since any *client-side factory* "lets a class defer instantiation to subclasses."

3. SINGLETON can be implemented without disturbing clients using *supplier-side factories*.

4. FLYWEIGHT: as above.

5. DECORATOR can be implemented using shakeins and S-Evolution, which can "add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects . . . responsibilities that can be withdrawn."

6. CHAIN OF RESPONSIBILITY can sometimes be represented using a series of shakein applications, each shakein acting as a receiving object and the application order representing the order within the chain.[1]

7. OBSERVER can sometimes be represented using shakeins; alternatively, a shakein can manage the list of observers and handle the sending of notifications, thereby simplifying the subject class. Events that should generate notification can be expressed, using JTL, as a pointcut parameter in said shakein.[2]

8. STATE is superfluous given object evolution; rather than having the object "appear to change its class," an actual change of class is possible.

Other known patterns, some specific to middleware frameworks, are likewise affected, including OBJECT POOL [122], INTERCEPTION FILTER, SERVICE TO WORKER, SERVICE LOCATOR, and DOMAIN STORE [6].

7.1 Directions for Future Research

Here are a few important directions for future research, based on the work presented in this dissertation:

Applications in other domains. This work focused on the applicability of shakeins and related technologies to the domain of enterprise software development in general, and middleware frameworks in particular. However, we believe that some of these techniques could be just as usable for other software development domains. The need to master software complexity in some domains is reflected in the emergence of new kinds of middleware frameworks, such as middleware frameworks for online game development [104] and in particular massively multiplayer games [18]; middleware frameworks for wireless application development [225], and more. While the non-functional, cross-cutting concerns presented in each of these domains are significantly different than those presented in enterprise application development, they can still be represented using shakeins and related mechanisms. And although our design goal was better handling of large-scale applications, nothing in particular limits the use of the resulting mechanisms, such as factories, in small-scale application development as well.

A type system for shakeins. Another challenging topic is that of a type system for shakeins, including a type system for making and enforcing constraints on shakein parameters, and typing of shakein composition. Such a type system should deal with the case that some of the configuration parameters are specified at the time of composition.

[1] Interestingly, AOP solutions that use interceptors—such as the Spring framework—employ CHAIN OF RESPONSIBILITY to implement advice.

[2] Filman and Friedman's milestone paper about the definition of AOP [97] was motivated by the question whether event-based publish-and-subscribe mechanisms, i.e., the OBSERVER design pattern, are AOP.

Mass application of shakeins. We argued that explicit application of shakeins to classes contributes to the expressive power that programmers may need. We explained why a global, system-wide application of aspects may lead to undesired results. Still, as evident by the ASPECTJ2EE experience, it is necessary at times to apply shakeins to a large number of classes. We need a mechanism that supports this need. The answer may lie with a syntax and semantics for applying a certain shakein, or even a family of shakeins, to an entire package, or to a class hierarchy.

Reflective mixins. The two key differences between shakeins and mixins is that the latter are unaware of the class that they extend, but on the other hand they may change the type, not only the implementation, of classes to which they are applied. By adding reflection capabilities to mixins (i.e., pointcut and advice mechanisms), we can create a hybrid shakeins-mixins construct that can change the type while being aware of the modified class. This construct can be made even more powerful by accepting a new parameter type: member names. For example, consider an `Undo` mixin (based on Figure 6.16) that accepts two parameters—a pointcut defining the operation to undo (e.g., `setName`) and the name of the undo method to be introduced (e.g., `undoSetName`). Such a mixin can be applied repeatedly, each time adding a new undo method for undoing a different kind of operation.

Embedded JTL. We would like to see a type-safe version of embedded JTL, similar to the work on issuing type safe-embedded SQL calls from JAVA [71, 165] and the C# LINQ project [168]. The grand challenge is in a seamless integration, a *linguistic symbiosis* [41] of JTL with JAVA, perhaps in a manner similar to by which XML was integrated into the language by Harden, Raghavachari, and Shmueli [131].

Evolution of Generic Types. A unique combination of theoretical and practical challenge is posed by the interaction of object evolution with the parametric polymorphism of JAVA. Although `List<Dog>` is not a subclass of `List<Animal>`, we still may want to evolve the latter into the former; not just evolve the content of the list, but the list object itself. Since the two types are related, and the change is monotonic, it may be possible to develop mechanisms for this kind of reclassification which are not as general as what is found in FICKLE. A unique practical slant of the problem is that in JAVA, the two types have the same runtime representation [39].

Bibliography

[1] Alfred Vaino Aho, Brian W. Kernighan, and Peter J. Weinberger. *The AWK programming language*. Addison-Wesley series in Computer Science. Addison-Wesley Publishing Company, Reading, Massachusetts, 1988.

[2] Alfred Vaino Aho, Ravi Sethi, and Jeffry D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[3] Antonio Albano, Giorgio Ghelli, and Renzo Orsini. Fibonacci: A programming language for object databases. *The VLDB Journal*, 4(3):403–444, 1995.

[4] Jonathan Erik Aldrich and Craig Chambers. Ownership domains: Separating aliasing policy from mechanisms. In Odersky [183], pages 1–25.

[5] Eric Allen, Jonathan Bannet, and Robert Cartwright. A first-class approach to genericity. In Crocker and Jr. [74], pages 96–114.

[6] Deepak Alur, John Crupi, and Dan Malks. *Core J2EE Design Patterns: Best Practices and Design Strategies*. Core Design Series. Prentice-Hall, Englewood Cliffs, New Jersy 07632, second edition, 2003.

[7] D. Ancona, C. Anderson, F. Damiani, S. Drossopoulou, P. Giannini, and E. Zucca. A type preserving translation of Fickle into Java. In *TOSCA'01*, volume 62 of *ENTCS*, pages 69–82. Elsevier, 2002.

[8] Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam—designing a Java extension with mixins. *ACM Transactions on Programming Languages and Systems*, 25(5):641–712, 2003.

[9] Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. A framework for implementing pluggable type systems. In Tarr and Cook [219].

[10] Giuliano Antoniol, Massimiliano Di Penta, and Ettore Merlo. YAAB (Yet Another AST Browser): Using OCL to navigate ASTs. In *Proc. of the Eleventh International Workshop on Program Comprehension (IWPC'03)*, pages 13–22, Portland, Oregon, USA, May 10-11 2003.

[11] *Proc. of the Second International Conference on Aspect-Oriented Software Development (AOSD'03)*, Boston, Massachusetts, USA, March 17-21 2003. ACM Press, New York, NY, USA.

[12] Apache Software Foundation. iBATIS product homepage. http://ibatis.apache.org/.

[13] Apache Software Foundation. BCEL - the Bytecode Engineering Library. http://jakarta.-apache.org/bcel/, 2002.

[14] Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley Publishing Company, Reading, Massachusetts, 1996.

[15] Isaac Asimov. The Life and Times of Multivac. In *The Bicentennial Man and Other Stories*. Doubleday, Garden City, NY, 1976.

[16] Darren C. Atkinson and William G. Griswold. The design of whole-program analysis tools. In *Proc. of the Eighteenth International Conference on Software Engineering (ICSE'96)*, pages 16–27, Berlin, Germany, March 25-30 1996.

[17] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proc. of the Thirtieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03)*, pages 285–298, New Orleans, Louisiana, USA, 2003. ACM Press, New York, NY, USA.

[18] Rajesh Krishna Balan, Maria Ebling, Paul Castro, , and Archan Misra. Matrix: Adaptive middleware for distributed multiplayer games. In Gustavo Alonso, editor, *Middleware 2005*, number 3790 in LNCS. Springer Verlag, December 2005.

[19] Robert Balzer, Neil M. Goldman, and David S. Wile. On the transformational implementation approach to programming. In *Proc. of the Second International Conference on Software Engineering (ICSE'76)*, pages 337–344, San Francisco, California, United States, 1976. IEEE Computer Society Press.

[20] Larry A. Barowski and James H. Cross II. Extraction and use of class dependency information for Java. In *Proc. of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, pages 309–318, Richmond, Virginia, USA, October 2002. IEEE Computer Society Press.

[21] Ohad Barzilay, Yishai A. Feldman, Shmuel Tyszberowicz, and Amiram Yehudai. Call and execution semantics in AspectJ. In Curtis Clifton, Ralf Lämmel, and Gary T. Leavens, editors, *Proc. of the Workshop on Foundations of Aspect-Oriented Languages (FOAL) at AOSD'04*, pages 19–23, 2004.

[22] Christian Bauer and Gavin King. *Java Persistence with Hibernate*. Manning, New York, NY, November 2006.

[23] BEA Systems Inc. WebLogic Server product homepage. http://www.bea.com/content/-products/weblogic/.

[24] Kent Beck. Instance specific behavior: how and why. *The Smalltalk Report*, 2(6):13–15, March-April 1993.

[25] Johannes Bellert and Pavel Vlasov. *Hammurapi Use Manual: How to Hammurapi*. Hammurapi Group, 2006. http://www.hammurapi.biz/products/hammurapi/doc/User-Manual.pdf.

[26] L. Bendix, A. Dattolo, and F. Vitali. Software configuration management in software and hypermedia engineering: A survey. In *Handbook of Software Engineering and Knowledge Engineering*, volume 1, pages 523–548. World Scientific Publishing, 2001.

[27] Andrew P. Black, editor. *Proc. of the Ninetieth European Conference on Object-Oriented Programming (ECOOP'05)*, volume 3086 of *Lecture Notes in Computer Science*, Glasgow, UK, July 25–29 2005. Springer Verlag.

[28] Gordon S. Blair, Lynne Blair, Awais Rashid, Ana Moreira, João Araújo, and Ruzanna Chitchayan. Engineering aspect-oriented systems. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect-Oriented Software Development*, chapter 17, pages 379–406. Addison-Wesley Publishing Company, 2005.

[29] Joshua Bloch. *Effective Java Programming Language Guide*. Addison-Wesley Publishing Company, Reading, Massachusetts, first edition, June 2001.

[30] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. *XQuery 1.0: An XML Query Language*. W3C, 2005.

[31] Stephanie Bodoff, Dale Green, Kim Haase, Eric Jendrock, Monica Pawlan, and Beth Stearns. *The J2EE Tutorial*. Addison-Wesley Publishing Company, Reading, Massachusetts, 2002.

[32] Boris Bokowski and Jan Vitek. Confined types. In *Proc. of the Fourteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99)*, pages 82–96, Denver, Colorado, November1–5 1999. ACM Press, New York, NY, USA, ACM SIGPLAN Notices 34 (10).

[33] Joans Bonér and Alexandre Vasseur. Enabling aspect-oriented programming in WebLogic Server using the JRockit Management API. http://dev2dev.bea.com/pub/a/2004/-05/boner_vasseur.html, May 2004.

[34] Jonas Bonér. What are the key issues for commercial AOP use—how does AspectWerkz address them? In Murphy and Lieberherr [178], pages 5–6.

[35] Grady Booch. *Object Oriented Design with Applications*. The Benjamin/Cummings Publishing Company, Inc., 1991.

[36] E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem*. Perspectives of Mathematical Logic. Springer Verlag, 1997.

[37] Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, Department of Computer Science, University of Utah, 1992.

[38] Gilad Bracha and William R. Cook. Mixin-based inheritance. In Norman K. Meyrowitz, editor, *Proc. of the Fifth Object-Oriented Programming Systems, Languages, and Applications / European Conference on Object-Oriented Programming OOPSLA/ECOOP'90*, pages 303–311, Ottawa, Canada, October 21-25 1990. ACM SIGPLAN Notices 25(10).

[39] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In OOPSLA'98 [185], pages 183–200.

[40] Tim Bray, Jean Paoliand C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. *Extensible Markup Language (XML) 1.0*. W3C Recommendation. World Wide Web Consortium, fourth edition, August 2006. http://www.w3.org/TR/2006/REC-xml-20060816/.

[41] Johan Brichau, Kris Gybels, and Roel Wuyts. Towards a linguistic symbiosis of an object-oriented and a logic programming language. In J. Striegnitz, Kei Davis, and Yannis Smaragdakis, editors, *Proc. of the Workshop on Multiparadigm Programming with Object-Oriented Languages (MPOOL'02) at the European Conference on Object-Oriented Programming*, June 2002.

[42] Rodney A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 256–262, New York, NY, USA, 1984. ACM Press.

[43] John Brunner. *Stand on Zanzibar*. Doubleday, Garden City, NY, 1968.

[44] Timothy A. Budd. *An Introduction to Object-Oriented Programming*. Addison-Wesley Publishing Company, Reading, Massachusetts, first edition, 1991.

[45] Bill Burke and Adrian Brock. Aspect-oriented programming and JBoss. *OnJava.com*, May 2003. http://www.onjava.com/pub/a/onjava/2003/05/28/aop_jboss.html.

[46] Mary Campione, Kathy Walrath, and Alison Huml. *The Java Tutorial: A Short Course on the Basics*. Addison-Wesley Publishing Company, Reading, Massachusetts, 2000.

[47] Peter S. Canning, William R. Cook, Walter L. Hill, and Walter G. Olthoff. Interfaces for strongly-typed object-oriented programming. In Norman K. Meyrowitz, editor, *Proc. of the Fourth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'89)*, pages 457–467, New Orleans, Louisiana, October 1-6 1989. ACM SIGPLAN Notices 24(10).

[48] Luca Cardelli and Peter Wegner. On understanding types, data abstractions, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[49] Giuseppe Castagna. Covariance and contravariance: Conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, May 1995.

[50] Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic programming and databases*. Springer Verlag, New York, 1990.

[51] Craig Chambers. The Cecil language, specification and rationale. Technical Report TR-93-03-05, University of Washington, Seattle, 1993.

[52] Craig Chambers. Predicate classes. In Oscar M. Nierstrasz, editor, *Proc. of the Seventh European Conference on Object-Oriented Programming (ECOOP'93)*, volume 707 of *Lecture Notes in Computer Science*, pages 268–296, Kaiserslautern, Germany, July 26-30 1993. Springer Verlag.

[53] Patrick Chan, Rosanna Lee, and Douglas Kramer. *The Java Class Libraries*, volume 1. Addison-Wesley Publishing Company, Reading, Massachusetts, second edition, 1998.

[54] Yih-Farn Chen, Michael Nishimoto, and C.V. Ramamoorthy. The C information abstraction system. *IEEE Transactions on Software Engineering*, 16(3):325–334, March 1990.

[55] Shigeru Chiba and Kiyoshi Nakagawa. Josh: an open AspectJ-like language. In Murphy and Lieberherr [178], pages 102–111.

[56] Jung Pil Choi. Aspect-oriented programming with Enterprise JavaBeans. In *Proc. of the Fourth International Enterprise Distributed Object Computing Conference (EDOC 2000)*, pages 252–261. IEEE Computer, 2000.

[57] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proc. of the Tenth International Static Analysis Symposium, SAS '03*, volume 2694 of *LNCS*, pages 1–18. Springer Verlag, June 2003.

[58] Sara Cohen, Joseph (Yossi) Gil, and Evelina Zarivach. Datalog programs over infinite databases, revisited. http://www.cs.technion.ac.il/jtl/datalog-infinite.pdf, 2006. Submitted.

[59] Tal Cohen. Java Q&A: How do I correctly implement the equals() method? *Dr. Dobbs Journal*, 336:83–86, May 2002.

[60] Tal Cohen and Joseph Gil. Self-calibration of metrics of Java methods. In *Proc. of the Thirty Seventh International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'00 Pacific)*, pages 94–106, Sydney, Australia, November 20-23 2000. Prentice-Hall, Englewood Cliffs, New Jersy 07632.

[61] Tal Cohen and Joseph Gil. AspectJ2EE = AOP + J2EE: Towards an aspect based, programmable and extensible middleware framework. In Odersky [183], pages 219–243.

[62] Tal Cohen and Joseph (Yossi) Gil. Three approaches to object evolution. Submitted.

[63] Tal Cohen and Joseph (Yossi) Gil. Shakeins: Nonintrusive aspects for middleware frameworks. *Transactions on Aspect-Oriented Software Development II*, Lecture Notes in Computer Science vol. 4242, November 2006.

[64] Tal Cohen and Joseph (Yossi) Gil. Better construction with factories. *Journal of Object Technology (to appear)*, July/August 2007.

[65] Tal Cohen, Joseph (Yossi) Gil, and Itay Maman. Guarded program transformations with JTL. Submitted.

[66] Tal Cohen, Joseph (Yossi) Gil, and Itay Maman. Extending JTL to other languages. http://www.cs.technion.ac.il/jtl/jtl-over-cs-b.pdf, 2006.

[67] Tal Cohen, Joseph (Yossi) Gil, and Itay Maman. JTL—the Java tools language. In Tarr and Cook [219].

[68] Tal Cohen, Joseph (Yossi) Gil, and Itay Maman. JTL and the annoying subtleties of precise μ-pattern definitions, October 2006. 1st International Workshop on Design Patterns Detection for Reverse Engineering (DPD4RE 2006/WCRE 2006).

[69] Mariano Consens, Alberto Mendelzon, and Arthur Ryman. Visualizing and querying software structures. In *CASCON'91*, pages 17–35. IBM Press, 1991.

[70] Constantinos A. Constantinides, Tzilla Elrad, and Mohamed E. Fayad. Extending the object model to provide explicit support for crosscutting concerns. *Software - Practice and Experience*, 32:703–734, 2002.

[71] William R. Cook and Siddhartha Rai. Safe query objects: statically typed objects as remotely executable queries. In Roman et al. [196], pages 97–106.

[72] Pascal Costanza. Dynamic replacement of active objects in the Gilgul programming language. In *IFIP/ACM Working Conference*, volume 2370 of *Lecture Notes in Computer Science*, page 125. Springer Verlag, June 2002.

[73] Roger F. Crew. ASTLOG: A language for examining abstract syntax trees. In S. Kamin, editor, *Proc. of the First USENIX Conference Domain Specific Languages (DSL'97)*, pages 229–242, Santa Barbara, October 1997.

[74] Ron Crocker and Guy L. Steele Jr., editors. *Proc. of the Eighteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'03)*, Anaheim, California, USA, October 2003. ACM SIGPLAN Notices 38 (11).

[75] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Publishing Company, June 2000.

[76] Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. Re-classification and multi-threading: Fickle$_{\mathtt{MT}}$. In *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC'04)*, pages 1297–1304, New York, NY, USA, 2004. ACM Press, New York, NY, USA.

[77] Ferruccio Damiani, Sophia Drossopoulou, and Paola Giannini. Refined effects for unanticipated object r-classification: Fickle$_3$ (extended abstract). In *Theoretical Computer Science: 8th Italian Conference (ICTCS'03)*, volume 2841 of *Lecture Notes in Computer Science*, pages 97–110. Springer Verlag, October 2003.

[78] Barbara Darrow. Application integration, middleware market remains strong. *CRN Daily News*, April 2005. http://www.crn.com/sections/breakingnews/breakingnews.jhtml?-articleId=160701365.

[79] Kris De Volder. *Type-Oriented Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, 1998. Adv: Prof. Dr. Theo D'Hondt.

[80] Kris De Volder and Theo D'Hondt. Aspect-oriented logic meta programming. In *Proc. of the Ninetieth European Conference on Object-Oriented Programming (ECOOP'05)*, volume 1616 of *Lecture Notes in Computer Science*, pages 250–272. Springer Verlag, 1999.

[81] Linda DeMichiel and Michael Keith. *Enterprise Java Beans Version 3.0*. JSR 220. Sun Microsystems Inc., May 2006. http://jcp.org/en/jsr/detail?id=220.

[82] Linda G. DeMichiel, L. Umit Yalçinalp, and Sanjeev Krishnan. Enterprise JavaBeans specification, version 2.0. http://java.sun.com/j2ee/, 2001.

[83] Pierre Deransart, Laurent Cervoni, and AbdelAli Ed-Dbali. *Prolog: The Standard: reference manual*. Springer-Verlag, London, UK, 1996.

[84] Premkumar T. Devanbu. GENOA—a customizable, front-end-retargetable source code analysis framework. *ACM Trans. on Soft. Eng. and Methodology*, 8(2):177–212, 1999.

[85] Edsger W. Dijkstra. Guarded commands, non-determinancy and a calculus for the derivation of programs. In Friedrich L. Bauer and Klaus Samelson, editors, *Language Hierarchies and Interfaces*, volume 46 of *Lecture Notes in Computer Science*, pages 111–124, Marktoberdorf, Germany, July 1975. Springer Verlag.

[86] Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. Fickle: Dynamic object re-classification. In Knudsen [155], pages 130–149.

[87] Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. More dynamic object re-classification: Fickle$_{\mathrm{II}}$. *ACM Transactions on Programming Languages and Systems*, pages 153–191, March 2002.

[88] Frédéric Duclos, Jacky Estublier, and Philippe Morat. Describing and using non functional aspects in component based applications. In *Proc. of the First International Conference on Aspect-Oriented Software Development (AOSD 2002)*, pages 22–26, 2002.

[89] Eclipse Foundation. Eclipse homepage. http://www.eclipse.org/, 2006.

[90] ECMA International. *Standard ECMA-367: Eiffel Analysis, Design, and Programming Language*. ECMA International, June 2005.

[91] Michael Eichberg, Mira Mezini, and Klaus Ostermann. Pointcuts as functional queries. In Wei-Ngan Chin, editor, *APLAS*, volume 3302 of *Lecture Notes in Computer Science*, pages 366–381, Taipei, Taiwan, 2004. Springer.

[92] Michael Eichberg, Mira Mezini, Klaus Ostermann, and Thorsten Schäfer. XIRC: A kernel for cross-artifact information engineering in software development environments. In *Proc. of the Eleventh Working Conference on Reverse Engineering (WCRE'04)*, pages 182–191, Delft, Netherlands, November 8-12 2004. IEEE Computer Society Press.

[93] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley Publishing Company, Reading, Massachusetts, January 1994.

[94] Erik Ernst. Family polymorphism. In Knudsen [155], pages 303–326.

[95] Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In Crocker and Jr. [74], pages 302–312.

[96] David C. Fallside and Priscilla Walmsley. *XML Schema Part 0: Primer*. W3C Recommendation. World Wide Web Consortium, second edition, October 2004. http://www.w3.org/TR/xmlschema-0/.

[97] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *OOPSLA 2000 Workshop on Advanced Separation of Concerns*. ACM, Minneapolis, 2000.

[98] Robert E. Filman and Klaus Havelund. Realizing aspects by transforming for events. In *Automated Software Engineering (ASE'02)*, September 2002.

[99] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proc. of the Conference on Programming Language Design and Implementation (PLDI'02)*, pages 234–245, Berlin, Germany, June 17-21 2002. Compaq Systems Research Center.

[100] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Object Technology Series. Addison-Wesley Publishing Company, Reading, Massachusetts, 2000.

[101] Martin Fowler. Inversion of control containers and the dependency injection pattern. http://www.martinfowler.com/articles/injection.html, 2004.

[102] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[103] Pascal Fradet and Mario Südholt. An aspect language for robust programming. In *ECOOP Workshops*, pages 291–292, 1999.

[104] Markus Friedl. *Online Game Interactivity Theory*. Advances in Computer Graphics and Game Development Series. Charles River Media, October 2002.

[105] Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing series. Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.

[106] Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. A comparative study of language support for generic programming. In Crocker and Jr. [74], pages 115–134.

[107] Giorgio Ghelli. Foundations for extensible objects with roles. *Inf. Comput.*, 175(1):50–75, 2002.

[108] Giorgio Ghelli and Debora Palmerini. Foundations of extensible objects with roles (extended abstract). In *Sixth International Workshop on Foundations of Object-Oriented Languages (FOOL'6)*, January 1999.

[109] Pietro Di Gianantonio, Furio Honsell, and Luigi Liquori. A lambda calculus of objects with self-inflicted extension. In OOPSLA'98 [185], pages 166–178.

[110] Joseph Gil and Itay Maman. Micro patterns in Java code. In Johnson and Gabriel [145], pages 97–116.

[111] Joseph Gil and Yuri Tsoglin. JAMOOS—a domain-specific language for language processing. *J. Comp. and Inf. Tech.*, 9(4):305–321, 2001.

[112] Brian Goetz. Threading lightly, part 3: Sometimes it's best not to share; exploiting ThreadLocal to enhance scalability. *IBM developerWorks*, October 2001. http://www-128.ibm.com/developerworks/java/library/j-threads3.html.

[113] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1984.

[114] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.

[115] Simon Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. In Johnson and Gabriel [145], pages 385–402.

[116] Ian Gorton and Liming Zhu. Tool support for *Just-in-Time* architecture reconstruction and evaluation: An experience report. In Roman et al. [196], pages 514–523.

[117] James Gosling, Bill Joy, Guy L. Jr. Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley Publishing Company, Reading, Massachusetts, third edition, June 2005.

[118] O. C. Z. Gotel and A. C. W. Finkelstein. An analysis of the requirements traceability problem. In *Proc. of the First International Conference on Requirements Engineering (ICRE'94)*, pages 94–101, Colorado Springs, Colorado, April 1994. IEEE Computer Society Press.

[119] G. Gottlob, E. Grädel, and H. Veith. Linear time Datalog for branching time logic. In *Logic-Based Artificial Intelligence*. Kluwer, 2000.

[120] Stephen Jay Gould. *Full House: The Spread of Excellence from Plato to Darwin*. Harmony Books, September 1996.

[121] Joseph D. Gradecki and Nicholas Lesiecki. *Mastering AspectJ: Aspect-Oriented Programming in Java*. Wiley, 2003.

[122] Mark Grand. *Patterns in Java, Volume 1*. John Wiley & Sons, USA, 1998.

[123] Judith E. Grass and Yih-Farn Chen. The C++ information abstractor. In *Proc. of the USENIX C++ Conference*, pages 265–277, San Fransisco, CA, April 1990. AT&T Bell Laboratories, USENIX Association.

[124] Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: First-class language support for generic programming in C++. In Tarr and Cook [219].

[125] William G. Griswold, Darren C. Atkinson, and Collin McCurdy. Fast, flexible syntactic pattern matching and processing. In *Proc. of the Fourth Workshop on Program Comprehension (WPC '96)*, pages 144–153, Washington, DC, 1996. IEEE Computer Society Press.

[126] Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In AOSD'03 [11], pages 60–69.

[127] A. N. Habermann and David Notkin. Gandalf: Software development environments. *IEEE Transactions on Software Engineering*, 12(12):1117–1127, December 1986.

[128] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. CodeQuest: Scalable source code queries with Datalog. In Dave Thomas, editor, *Proc. of the Twentieth European Conference on Object-Oriented Programming (ECOOP'05)*, volume 4067 of *Lecture Notes in Computer Science*, Nantes, France, July 3–7 2006. Springer Verlag.

[129] Ruibing Hao, Ladislau Boloni, Kyungkoo Jun, and Dan C. Marinescu. An aspect-oriented approach to distributed object security. In *Proc. of the Fourth IEEE Symp. on Computers and Communications*. IEEE Computer Society Press. 10622 Los Vaqueros Circle, Los Alamitos, CA 90720-1264, 1999.

[130] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[131] Matthew Harren *et al.* XJ: integration of XML processing into Java. In *WWW Alt. '04: Proc. of the Thirteenth International World Wide Web Conference on Alternate track papers & posters*, pages 340–341, New York, NY, USA, 2004. ACM Press, New York, NY, USA.

[132] Erik Hatcher and Steve Loughran. *Java Development with Ant*. Manning Publications, August 2002.

[133] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *The C# Programming Language*. Addison-Wesley Publishing Company, Reading, Massachusetts, second edition, October 2003.

[134] Yoav Hollander, Matthew Morley, and Amos Noy. The e language: A fresh separation of concerns. In *Proc. of the International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'01 Europe)*, pages 41–51, Zurich, Switzerland, March 12–14 2001. Prentice-Hall, Englewood Cliffs, New Jersy 07632.

[135] Reid Holmes and Gail C. Murphy. Using structural context to recommend source code examples. In Roman et al. [196], pages 117–125.

[136] Arnaud Le Hors, Philippe Le Hégaret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion, and Steve Byrne. *Document Object Model (DOM) Level 3 Core Specification*. W3C Recommendation. World Wide Web Consortium, April 2004. http://www.w3.org/-TR/2004/REC-DOM-Level-3-Core-20040407/.

[137] David Hovemeyer and William Pugh. Finding bugs is easy. *ACM SIGPLAN Notices*, 39(12):92–106, 2004.

[138] Shan Shan Huang and Yannis Smaragdakis. Easy language extension with Meta-AspectJ. In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, *Proc. of the Twenty Eighth International Conference on Software Engineering (ICSE'06)*, pages 865–868, Shanghai, China, May20-28 2006. ACM Press, New York, NY, USA.

[139] IBM Corp. IBM WebSphere Application Server product family homepage. http://www-3.ibm.com/software/info1/websphere/index.jsp?tab=products/appserv.

[140] Sasan Iman and Sunita Joshi. *The e Hardware Verification Language*. Springer Verlag, 1st edition, May 2004.

[141] ISE. *ISE EIFFEL The Language Reference*. ISE, Santa Barbara, CA, 1997.

[142] Ivar Jacobson. *Object-Oriented Software Engineering - A Use Case Driven Approach*. Addison-Wesley Publishing Company, Reading, Massachusetts, first edition, June 1992.

[143] Doug Janzen and Kris De Volder. Navigating and querying code without getting lost. In *Proc. of the Second international conference on Aspect-Oriented Software Development (AOSD'03)*, pages 178–187, New York, NY, USA, 2003. ACM Press.

[144] JBoss Group. JBoss product homepage. http://www.jboss.org/, 2003.

[145] Ralph Johnson and Richard P. Gabriel, editors. *Proc. of the Twentieth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'05)*, San Diego, California, October 2005. ACM SIGPLAN Notices.

[146] Rod Johnson, Juergen Hoeller, Alef Arendsen, and Thomas Risberg. *Professional Java Development with the Spring Framework*. Programmer to Programmer series. Wrox, Indianapolis, IN, July 2005.

[147] Richard Jones and Rafael Lins. *Garbage Collection : Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, 1996.

[148] Peter Kenens, Sam Michiels, Frank Matthijs, Bert Robben, Eddy Truyen, Bart Vanhaute, Wouter Joosen, and Pierre Verbaeten. An AOP case with static and dynamic aspects. In *ECOOP'98 Workshop Reader*, pages 428–429, 1998.

[149] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Software Series. Prentice-Hall, Englewood Cliffs, New Jersy 07632, second edition, 1988.

[150] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In Knudsen [155], pages 327–355.

[151] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satushi Matsuoka, editors, *Proc. of the Eleventh European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, June 9-13 1997. Springer Verlag.

[152] Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In Roman et al. [196], pages 49–58.

[153] Gregor Kiczales and Mira Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In Black [27].

[154] Howard Kim and Siobaán Clarke. The relevance of AOP to an applications programmer in an EJB environment. Proc. of the First International Conference on Aspect-Oriented Software Development (AOSD) Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), 2002.

[155] Jørgen Lindskov Knudsen, editor. *Proc. of the Fifteenth European Conference on Object-Oriented Programming (ECOOP'01)*, volume 2072 of *Lecture Notes in Computer Science*, Budapest, Hungary, June 2001. Springer Verlag.

[156] Donald Ervin Knuth. Structured Programming with go to Statements. *ACM Computing Surveys (CSUR)*, 6(4):261–301, 1974.

[157] Kostas Kontogiannis, Johannes Martin, Kenny Wong, Richard Gregory, Hausi A. Müller, and John Mylopoulos. Code migration through transformations. In Stephen A. MacKay and J. Howard Johnson, editors, *Proc. of the Conference of the Centre for Advanced Studies on Collaborative research (CASCON'98)*, page 13, Toronto, Ontario, Canada, November 1998. IBM Press.

[158] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning, Greenwich, 2003.

[159] Ralf Lammel. Declarative aspect-oriented programming. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 131–146, 1999.

[160] Rasmus Jay Lerdorf, Kevin Tatroe, Bob Kaehms, and Ric McGredy. *Programming PHP*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, March 2002.

[161] Yossi Levanoni and Erez Petrank. An on-the-fly reference-counting garbage collector for Java. *ACM Transactions on Programming Languages and Systems*, 28(1):1–69, 2006.

[162] Roberto Lopez-Herrejon, Don Batory, and Christian Lengauer. A disciplined approach to aspect composition. In *Proc. of the ACM SIGPLAN Simposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'06)*, pages 68–77, Charleston, South Carolina, 2006. ACM Press, New York, NY, USA.

[163] Mark Lutz. *Programming* PYTHON. O'Reilly, first edition, October 1996.

[164] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: a program query language. In Johnson and Gabriel [145], pages 365–383.

[165] Russell A. McClure and Ingolf H. Krüger. SQL DOM: compile time checking of dynamic SQL statements. In Roman et al. [196], pages 88–96.

[166] Sean McDirmid, Matthew Flatt, and Wilson C. Hsieh. Jiazzi: New-age components for old-fashioned Java. In *Proc. of the Sixteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'01)*, pages 211–222, Tampa Bay, Florida, October 14–18 2001. ACM Press, New York, NY, USA, ACM SIGPLAN Notices 36(11).

[167] Sean McDirmid and Wilson C. Hsieh. Aspect-oriented programming with Jiazzi. In AOSD'03 [11], pages 70–79.

[168] Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *SIGMOD'06: Proceedings of the 2006 ACM SIGMOD international conference on management of data*, pages 706–706. ACM Press, New York, NY, USA, 2006.

[169] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Englewood Cliffs, New Jersy 07632, Englewood Cliffs, New Jersy, second edition, 1997.

[170] Bertrand Meyer. Attached types and their application to three open problems of object-oriented programming. In Black [27], pages 1–32.

[171] Mira Mezini and Klaus Ostermann. Conquering aspects with Caesar. In AOSD'03 [11], pages 90–100.

[172] Microsoft Corp. DCOM technical overview. http://msdn.microsoft.com/library/en-us/-dndcom/html/msdn_dcomtec.asp, 1996.

[173] Microsoft Corp. The official Microsoft ASP homepage. http://www.asp.net/, 2006.

[174] Russell Miles. An introduction to aspect-oriented programming with the Spring framework. *OnJava.com*, July 2004. http://www.onjava.com/pub/a/onjava/2004/07/14/springaop.html.

[175] R. Milner, M. Tofte, Robert Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[176] Yasuhiko Minamide and Akihiko Tozawa. XML validation for context free grammars. In *Proc. of the Fourth Asian Symposium on Programming Languages and Systems (APLAS'06)*, volume 4279 of *LNCS*, pages 357–373. Springer Verlag, 2006.

[177] H. A. Müller and K. Klashinsky. Rigi—A system for programming-in-the-large. In *Proc. of the Tenth International Conference on Software Engineering (ICSE'88)*, pages 80–86, Singapore, April 1988. IEEE Computer Society Press.

[178] Gail C. Murphy and Karl J. Lieberherr, editors. *Proc. of the Third International Conference on Aspect-Oriented Software Development (AOSD'04)*, Lancaster, UK, March 22-26 2004. ACM Press, New York, NY, USA.

[179] Isaac Nassi and Ben Shneiderman. Flowchart techniques for structured programming. *ACM SIGPLAN Notices*, 8(8):12–26, August 1973.

[180] Muga Nishizawa, Shigeru Chiba, and Michiaki Tatsubori. Remote pointcut: a language construct for distributed AOP. In *Proc. of the Third international conference on Aspect-Oriented Software Development (AOSD'04)*, pages 7–15, New York, NY, USA, 2004. ACM Press.

[181] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In John M. Vlissides and Douglas C. Schmidt, editors, *Proc. of the Nineteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'04)*, pages 99–115, Vancouver, BC, Canada, October 2004. ACM SIGPLAN Notices 39 (10).

[182] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *Proc. of the Twelfth International Conference on Compiler Construction (CC'03)*, pages 138–152, Warsaw, Poland, April 2003. Springer Verlag.

[183] Martin Odersky, editor. *Proc. of the Eighteenth European Conference on Object-Oriented Programming (ECOOP'04)*, volume 3086 of *Lecture Notes in Computer Science*, Oslo, Norway, June 2004. Springer Verlag.

[184] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[185] *Proc. of the Thirteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'98)*, Vancouver, British Columbia, Canada, October18-22 1998. ACM SIGPLAN Notices 33(10).

[186] Ed Ort and Bhakti Mehta. Java Architecture for XML Binding (JAXB). *Sun Developer Network*, March 2003. http://java.sun.com/developer/technicalArticles/WebServices/jaxb/.

[187] Klaus Ostermann, Mira Mezini, and Christoph Bockisch. Expressive pointcuts for increased modularity. In Black [27], pages 214–240.

[188] David L. Parnas. Information distribution aspects of design methodology. In C. V. Freiman, John E. Griffith, and J. L. Rosenfeld, editors, *Proc. of the IFIP Congress*, IFIP Transactions, pages 339–44, Amsterdam, August 1971. North-Holland Publ. Co.

[189] Santanu Paul and Atul Prakash. Querying source code using an algebraic query language. In Hausi A. Müller and Mary Georges, editors, *Proc. of the Tenth IEEE International Conference on Software Maintenance (ICSM'94)*, pages 127–136, Victoria, BC, Canada, September 1994. IEEE Computer.

[190] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, Gerard Florin, Fabrice Legond-Aubry, and Laurent Martelli. JAC: an aspect-based distributed dynamic framework. *Soft. - Pract. and Exper.*, 34(12):1119–1148, 2004.

[191] Roman Pichler, Klaus Ostermann, and Mira Mezini. On aspectualizing component models. *Software - Practice and Experience*, 33:957–974, 2003.

[192] Andrei Popovici, Thomas R. Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In *Proc. of the First International Conference on Aspect-Oriented Software Development (AOSD'02)*, pages 141–147. ACM Press, New York, NY, USA, 2002.

[193] Eric S. Raymond. *The Art of UNIX Programming*. Addison-Wesley Publishing Company, 2003.

[194] Reasoning Systems. *REFINE User's Manual*, 1988.

[195] Tobias Rho, Günter Kniesel, Malte Appeltauer, and Andreas Linder. LogicAJ home page, 2006. http://roots.iai.uni-bonn.de/research/logicaj/.

[196] Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors. *Proc. of the Twenty Seventh International Conference on Software Engineering (ICSE'05)*, New York, NY, USA, May 15-21 2005. ACM Press, New York, NY, USA.

[197] Daniel Rubio. PMD: A code analyzer for Java programmers. *NewsForge*, November 2004. http://www.newsforge.com/article.pl?sid=04/11/11/1517207.

[198] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. A comparison of bug finding tools for Java. In *Proc. of the Fifteenth International Symposium on Software Reliability Engineering (ISSRE 2004)*, pages 245–256, Saint-Malo, Bretagne, France, November 2–5 2004. IEEE Computer Society Press.

[199] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behavior. In Luca Cardelli, editor, *Proc. of the Seventeenth European Conference on Object-Oriented Programming (ECOOP'03)*, volume 2743 of *Lecture Notes in Computer Science*, pages 248–274, Darmstadt, Germany, July 21–25 2003. Springer Verlag.

[200] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Composable encapsulation policies. In Odersky [183], pages 26–50.

[201] Manuel Serrano. Wide classes. In Rachid Guerraoui, editor, *Proc. of the Thirteenth European Conference on Object-Oriented Programming (ECOOP'99)*, volume 1628 of *Lecture Notes in Computer Science*, pages 391–415, Lisbon, Portugal, June 1999. Springer Verlag.

[202] Bill Shannon. *Java 2 Platform Enterprise Edition Specification, v1.4*. JSR 151. Sun Microsystems Inc., November 2003. http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf.

[203] Bill Shannon, Mark Hapner, Vlada Matena, James Davidson, James Davidson, and Larry Cable. *Java 2 Platform, Enterprise Edition: Platform and Component Specifications*. Addison-Wesley, 2000.

[204] Michael Sipser. *Introduction to the Theory of Computation*. Thomson Publishing, second edition, 2006.

[205] Charles Smith and Sophia Drossopoulou. Chai: Traits for Java-like languages. In Black [27].

[206] Sérgio Soares, Eduardo Laureano, and Paulo Borba. Implementing distribution and persistence aspects with AspectJ. In *Proc. of the Seventeenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'02)*, pages 174–190, Seattle, Washington, November 4–8 2002. ACM SIGPLAN Notices 37(11).

[207] Olaf Spinczyk, Daniel Lohmann, and Matthias Urban. AspectC++: an AOP extension for C++. *Software Developer's Journal*, pages 68–76, May 2005.

[208] Tom Strelich. The Software Life Cycle Support Environment (SLCSE): a computer based framework for developing soft. sys. In *Proc. of the Third ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE'88)*, pages 35–44, Boston, Massachusetts, 1988. ACM Press, New York, NY, USA.

[209] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, Reading, Massachusetts, third edition, 1997.

[210] Bjarne Stroustrup and Gabriel Dos Reis. Concepts—design choices for template argument checking. ISO/IEC JTC1/SC22/WG21 no. 1522, 2003.

[211] Bjarne Stroustrup and Gabriel Dos Reis. Concepts–syntax and composition. ISO/IEC JTC1/SC22/WG21 no. 1536, 2003.

[212] Sun Microsystems, Inc. rmic - the Java RMI compiler. http://java.sun.com/j2se/1.4.2/docs/-tooldocs/solaris/rmic.html, 2003.

[213] Sun Microsystems Inc. Java 6 product homepage. http://java.sun.com/javase/6/, 2006.

[214] Sun Microsystems Inc. Java platform debugger architecture enhancements [in JavaSE 6]. http://java.sun.com/javase/6/docs/technotes/guides/jpda/enhancements.html, 2006.

[215] Vitaly Surazhsky and Joseph (Yossi) Gil. Type-safe covariance in C++. In H. M. Haddad, G. A. Papadopoulos, A. Omicini, R.L. Wainwright, L.M. Liebrock, M.J. Palakal, A. Andreou, and C. Pattichis, editors, *Proc. of the Nineteenth Annual ACM Symposium on Applied Computing (SAC'04)*, pages 1496–1502, Nicosia, Cyprus, March 2004. ACM Press, New York, NY, USA.

[216] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. JAsCo: an aspect-oriented approach tailored for component based software development. In AOSD'03 [11], pages 21–29.

[217] S. Tucker Taft and Robert A. Duff, editors. *Ada 95 Reference Manual, Language and Standard Libraries, International Standard ISO/IEC 8652: 1995(E)*, volume 1246 of *LNCS*. Springer Verlag, 1997.

[218] Antero Taivalsaari. Object-oriented programming with modes. *Journal of Object-Oriented Programming*, 6(3):25–32, June 1993.

[219] Peri L. Tarr and William R. Cook, editors. *Proc. of the Twenty First Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'06)*, Portland, Oregon, October22-26 2006. ACM SIGPLAN Notices.

[220] Michiaki Tatsubori, Shigeru Chiba, Kozo Itano, and Marc-Olivier Killijian. OpenJava: A class-based macro system for Java. In Walter Cazzola, Robert J. Stroud, and Francesco Tisato, editors, *Proc. of the First OOPSLA Workshop on Reflection and Software Engineering*, volume 1826 of *Lecture Notes in Computer Science*, pages 117–133, Denver, CO, USA, November 1999. OOPSLA'99, Springer Verlag.

[221] The Object Management Group. The Common Object Request Broker: Architecture and specification, revision 2.0, July 1995.

[222] Eddy Truyen, Bart Vanhaute, Wouter Joosen, Pierre Verbaeten, and Bo Nørregaard Jørgensen. Dynamic and selective combination of extensions in component-based applications. In *Proc. of the Twenty Third International Conference on Software Engineering (ICSE'01)*, pages 233–242, Toronto, Canada, May 12-19 2001. IEEE Computer Society Press.

[223] Allen van Gelder, Kenneth Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

[224] Jonne van Wijngaarden and Eelco Visser. Program transformation mechanics. a classification of mechanisms for program transformation with a survey of existing transformation systems. Technical Report UU-CS-2003-048, Institute of Information and Computing Sciences, Utrecht University, 2003.

[225] Roman Vichr and Vivek Malhotra. Introduction to wireless middleware. *IBM developerWorks*, September 2001. http://www-128.ibm.com/developerworks/wireless/library/wimidarch/.

[226] Eelco Visser. A survey of strategies in program transformation systems. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 57, 2001.

[227] Robert J. Walker, Elisa L. A. Baniassad, and Gail C. Murphy. An initial assessment of aspect-oriented programming. In *Proc. of the Twenty First International Conference on Software Engineering (ICSE'99)*, pages 120–130, Los Angeles, USA, May 16-22 1999. IEEE Computer Society Press.

[228] Carl Weinschenk. The application server market is dead; long live the application server market. http://www.serverwatch.com/tutorials/article.php/2234311, 2003.

[229] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proc. of the Conference on Programming Language Design and Implementation (PLDI'04)*, pages 131–144, New York, NY, USA, June 9-11 2004. ACM Press.

[230] N. Wirth and M. Reiser. *Programming in Oberon—Steps Beyond Pascal and Modula*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1992.

[231] Moshé M. Zloof. Query By Example. In *Proceedings of the National Computer Conference*, pages 431–438, Anaheim, CA, May 1975.