JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©**JOT, 2002**

Vol. 0, No. 0, Z

Better Construction with Factories

Tal Cohen and **Joseph (Yossi) Gil**
Department of Computer Science,
Technion—Israel Institute of Technology
Technion City, Haifa 32000, Israel

“*The Factory-Owning Class Controls the Means of Production*.”
K. Marx [14]

The polymorphic nature of object oriented programs means that client code expecting an instance of class C may use instead an instance of a class C' inheriting from C. But, in order to use such a different instance, one must create it, and in order to do so in current languages, must be familiar with the name of creating class. To break this coupling, we propose the novel notion of *factories*, which are class services (alongside methods and constructors) that manage the instance-creation step of object construction. In making the case for factories we propose a five-dimensional framework for understanding and analyzing the class notion in various programming languages.
We show that factories can naturally replace the “creational” design patterns, and describe the design and implementation of a JAVA language extension supporting both *supplier-side* and *client-side* factories. Possible implementations in other languages are discussed as well.

1 INTRODUCTION

Good programming languages support, at the language level, the general principle of hiding implementation details from the client [19]. 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* [15, 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.*”

This paper starts from the observation that despite the progress in language design, there is still a family of services which reveal more than they should of their implementation secrets. These services are what is known as *creation procedures* in some languages and *constructors* in others. 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.

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 dynamic aspects, 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 [20] mechanism for obtaining instances of Enterprise JavaBeans (EJBs, [6]). Clients must not directly invoke constructors for EJBs; rather, special methods of "home objects" 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.

Likewise, users of the Spring Application Framework[1] 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 [10], and OBJECT POOL [12]. 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

[1]`http://www.springframework.org`

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.

Outline Sec. 2 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, Sec. 3 expands on the motivation, by highlighting certain limitations of constructors. Factories are the subject of Sec. 4, which describes their JAVA syntax and some of the applications. This section also shows how factories support many classical design patterns. Sec. 5 describes how coupling between classes can be decreased using factories, and Sec. 6 describes the notion of client-side factories. Finally, Sec. 7 discusses the extension of the factories idea to other programming languages and concludes.

2 TERMINOLOGY

There are many ways in which people perceive the notion of class: as a "*repository for* behavior *associated with an object*" [2, p.13], a "*unit of software decomposition*" and a "*type*" [15, pp.170–171], a "*tool for creating new types*" [21, p.223], a "*group* [of objects]" [13, p.50][2], a "*set of objects that share a common structure and a common behavior*" [1, 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: *commonality*, *morphability*, *binding*, *encapsulation*, and *purpose*. The most interesting dimension is *purpose*, by which we identify, for each syntactical element of a class, a programming-language purpose. In Sec. 3 we shall argue that, judged by these dimension of evaluation, constructors make a bit of weird bird.

[2]but also a "*template for several objects* . . . [a description of] *how these objects are structured internally*"

Let us now describe in greater detail each of the five dimensions in turn.

1. Commonality. This dimension makes the distinction between *common* elements of the class notion (e.g., class variables and methods in SMALLTALK) 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.

2. Morphability. Morphability indicates the class element's ability to obtain a shape, or be re-shaped, in a subclass. We identify morphable, re-morphable, and un-morphable members. A class member obtains polymorphic behavior by being morphed in one class, and re-morphed in another.

- *Morphable* members are those that have no shape yet, and may be shaped in a subclass. They are known as `abstract` class members in some languages, `deferred` in others.
- *Re-morphable* members have a shape, but can be re-shaped in a subclass. The new shape may replace or refine the inherited one. Put otherwise, these are class members that may be overridden in a subclass. In many languages, members are re-morphable by default; in some, they must be explicitly marked as such (e.g., by using the `virtual` keyword in C++).
- Finally, *un-morphable* members have a shape that may not be altered by a subclass. This pertains to common elements in all languages, and to data members in most languages. Some languages allow the developer to explicitly mark a member as un-morphable; e.g., using the `final` keyword in JAVA.

Different languages offer different levels for morphability for similar class members. In EIFFEL, for example, a data member may be overridden by a method, making data members re-morphable. In JAVA, data members may be hidden [11, Sect. 8.3.3] but not overridden, making them un-morphable.

3. 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. Un-morphable functions 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.

4. Encapsulation. A class may encapsulate its elements. Languages exhibit great variability in encapsulation schemes. The tailor-made accessibility in EIFFEL, just as the three encapsulation levels in C++, are orthogonal to the previously presented dimensions. Conversely, in SMALLTALK encapsulation is linked with the element kind. Interestingly, **`private`** methods in JAVA and C++ are statically bound (and, being invisible to subclasses, un-morphable).

5. Purpose. Classes, being a unit of software decomposition, can be subjected to Parnas's [19] classical distinction between the *interface* and *materialization* (which Parnas calls "implementation") perspectives of a software component. We say that the interface and materialization are *purposes* that the class serves as a whole, and characterize its elements by this purpose.

But, 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 `Vector` from the standard JAVA library. The forge of `Vector`, depicted in Fig. 2.1(a), includes the signature of the four constructors provided by the class: the default constructor `Vector()`, the copy constructor `Vector(Collection)`, a variant that specifies the initial capacity (`Vector(`**`int`**`)`), and a variant that specifies both the initial capacity and the capacity growth increment (`Vector(`**`int`**`, `**`int`**`)`).

Fig. 2.1(c) shows the type of `Vector`, methods such as `addElement`, `capacity`, and others, as well as fields such as `capacityIncrement` and `elementData`. Superclasses also add to the type; in this case, the type of `Vector` includes methods and fields inherited from three superclasses. 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 implementing each method. All these may be thought of as the class *protocol*.

The *mold* for creating new objects is defined by the collection of all fields in this class

```
Vector:
   public Vector();
   public Vector(Collection);
   public Vector(int);
   public Vector(int, int);
```

(a) The forge.

`int capacityIncrement`	32 bits
`int elementCount`	32 bits
`Object[] elementData`	32 bits
Fields inherited from superclasses	
Hidden fields added by the JVM	

(b) The mold.

```
Vector<E>:
  protected int capacityIncrement;
  protected int elementCount;
  protected Object[] elementData;
  public void addElement(E);
  public int capacity();
  // ... etc.

  // From AbstractList:
  public ListIterator listIterator()
  public List subList(int, int)
  // ... etc.

  // From AbstractCollection:
  public int size();
  public void clear();
  // ... etc.
```

```
// From Object:
public Object clone();
public void wait();
public void notify();
public boolean equals(Object);
// ... etc.

// Upcast operations:
public (AbstractList)();
public (AbstractCollection)();
public (Object)();
public (Serializable)();
public (Iterable<E>)();
public (Collection<E>)();
public (List<E>)();
```

(c) The type.

Figure 2.1: The forge, type and mold of `java.lang.Vector`.

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 [8] used in C++, etc.

Fig. 2.1(b) presents the mold defined by class `Vector`. It includes fields defined in `Vector` 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.

3 MOTIVATION (CONSTRUCTOR ANOMALIES)

Factories, the JAVA language extension proposed in this paper, are methods which return new class instances. Syntactically, a factory is a method which overloads the **`new`** operator with respect to a certain class. This language extension requires no changes to the JVM.

Since factories are so related to constructors, we start the discussion with comments on constructors. These comments underline the motivation for factories, and should help in understanding the differences between the two concepts.

In mainstream object-oriented languages, clients of a class obtain instances using constructors. Analyzing constructors (in, e.g., JAVA or C++) with the terminology set in the previous section, we find that they present three anomalies. First, constructors are simultaneously common and particular: common—since they are invocable without an instance; particular—since they work on an object.

Second, constructors are both un-morphable and re-morphable: un-morphable—since they are not inherited, and re-morphable—since a subclass constructor must be a refinement of a superclass one.

Third, it is mundane to see that constructors obey 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 [10, 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*".

The confusion between static and dynamic binding penetrates into the constructor code itself, i.e., into the mill. Method invocation from the mill follows a static binding scheme in C++[3]; 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 [9]: during construction, fields of non-null types may contain null values.

It is interesting to note that in EIFFEL, constructors—known as *creation procedure*—can be viewed as purely particular, since they may only be invoked on some variable in scope. The second confusion, however, between static and dynamic binding, remains. A creation instruction such as **`create`** x`.make`[4] for some variable x of type C is statically bound, even if the creation procedure `make` is overridden in the subclass C'. To create x as an instance of the subclass, a statement such as **`create`** `{`C'`}` x`.make`[5] must be used.

In studying constructors further, we can identify three steps in an instance's birth process:

(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 exactly correspond to steps C1, C2 and C4 in the effects of a creation instruction **`create`** x`.make` in EIFFEL [15, 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 of a constructor is an *expression* rather than a statement,

[3]Even for **`virtual`** methods.
[4]Written as `!!`x`.make` in vintage EIFFEL.
[5]Written as `!`C'`!`x`.make` in vintage EIFFEL.

and can be performed without assigning the result to a variable. (EIFFEL also supports the invocation of a creation procedure as an expression [7, 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.

None of these languages, however, provides the developer with control over the *creation* step. (Note that overloading the `new` operator in C++ grants the programmer control over memory allocation, but not over the kind of object to be created, nor the decision if an 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`)[6].

Unfortunately, in contrast with most other patterns, the creational patterns cannot be implemented in OO 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 non-standard creation mechanism. 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.

Often, creational pattern also collide with inheritance. To enforce the use of a `getInstance` method and prevent accidental direct access to the constructors, one has to declare all constructors as `private`. But this habit implies that the class cannot be subclassed, since subclasses must have access to some constructor of their parent class. Defining the constructor as `protected` does not solve the problem in JAVA, since clients from the same package, which are not subclasses, will now be able to access it.

Worse still, since the `getInstance` method must be shared, it cannot be overridden in subclasses. Thus, if class C is subclassed by 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.

Unlike constructors, factories can be defined as either common or particular. 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.

[6]Such methods are sometimes called *factory methods*. While serving a similar purpose, they are different than our notion of *factories*.

4 FACTORIES

Class `STemplate` in Fig. 4.1 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

```
 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() { /* ... setup code ... */ }
11 }
```

Figure 4.1: A Singleton defined using a factory.

the expression **`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.)

In general, 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` is automatically thrown.)

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 have chosen to overload the keyword **`this`**, or more particularly, its use as a method call. Since the context prevents any ambiguity, there was no need to introduce a new keyword:

- 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(···)` 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.)

Note that the constructor can only be called from a factory in the same class; any use of **new** C`(···)`, 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 [18], or using prototype-based languages, such as CECIL [5]), the factory solution is not specific to singletons, and can be used for any creational design pattern. More examples will be presented in the sequel.

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 design 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 Sec. 3, 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.

Automatically Generated Factories

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`(···)` is always implemented by a factory whose signature matches the actual parameters. 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.

Fig. 4.2 shows an example of the AGFa rule. The class defined in Fig. 4.2(a) has a factory with no parameters. It also has a two-parameters constructor, with no matching factory. Fig. 4.2(b) shows the AGFa code that the compiler (internally) injects into the class.

```
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 no-args 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 4.2: A class (a) with a constructor and its AGFa (b).

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.*

Conversely, if an abstract class S_a *does* define factories, then you can write `new` S_a`(`$\cdots$`)` in your code. Fig. 4.3 shows an abstract class, `ScrollBar`, with a factory. This example is modeled after the famous example [10, 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 4.3: An abstract class with a factory.

As shown in Fig. 4.4, 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.

```
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 4.4: An interface with a factory.

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.*"
>
> – [3, p.526]; emphasis in the original.

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 [4]. 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. Fig. 5.1 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 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

```
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 5.1: One possible factory for the `List` interface.

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 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.

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 Fig. 6.1.

```
1 class Bank {
2   public static new Account(Customer c) {
3     if (c.hasBadHistory()) return new LimitedAccount(c);
4                                     // LimitedAccount is a subclass of Account
5     return Account.new Account(c);
6   }
7   // ... rest of the class omitted
8 }
```

Figure 6.1: 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 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 Fig. 6.1 will also be used for evaluating expressions of the form **`new`** `Account(c)` in subclasses of `Bank`.

This client-side factory 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(···)`.

Fig. 6.2 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. 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`**

```
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 6.2: Widget-factory classes defined using client-side factories.

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.

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. Fig. 6.3 shows how such factories can be used in the classical implementation of the ABSTRACT FACTORY design pattern.

Fig. 6.3(a) shows the abstract factory, while Fig. 6.3(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.

Fig. 6.4 shows how dynamically-bound factories can be used to implement the FACTORY METHOD design pattern (also known as VIRTUAL CONSTRUCTOR). The code in this figure implements the classic example (from [10, 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 code is very similar to that in the original GoF 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

```
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 6.3: Using non-**`static`** factories to implement a dynamically bound abstract factory class.

prefix “c`.`” can be dropped for code inside class C (so it is replaced with the **`this`** reference).

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 [16] 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.

```
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 6.4: Implementing pattern FACTORY METHOD with dynamically bound factories.

7 DISCUSSION

We believe factories to be a minor addition to the language syntax, which may still be of major importance in streamlining language design. We have implemented factories as a JAVA extension using the Polyglot [17] 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 `$new$`*classable*, where *classable* is the fully-qualified target class name, with every dot replaced by `$dot$`. For example, the factory for `Account`s in class `Bank` (Fig. 6.1) 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. To produce such methods, our modified JAVA compiler synthesizes an inner class for the interface, and the factories reside in this class. Thus, an interface with one or more factories will contain a **`static`** inner class called `$NewHolder$`, which in turn contains the `$new$` methods representing the factories.

Our implementation generates bytecode that can be used on any JAVA virtual machine. As discussed in Sec. 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. Note that 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. The first obstacle is that C++ already features a mechanism for overloading the **`new`** operator; yet this mechanism 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, however, is not merely syntactical: in C++, variables of any type can be defined on the stack rather than dynamically allocated. Yet given factories, the compiler may not know in advance the amount of memory to allocate on the stack for any given variable. We could require that only classes with no factory in scope (i.e., no supplier-side factory at all and no client-side factory in scope) may be defined on the stack. Factories thus introduce a dichotomy, similar to C$^{\#}$'s value vs. reference types or EIFFEL's expanded vs. reference types, into the language.

The EIFFEL language presents a different challenge for introducing factories. Unlike constructors in C++ or JAVA, creation procedures in EIFFEL can have any desired name. 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 **create** 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).

Fig. 7.1 shows an EIFFEL version of the singleton class from Fig. 4.1. This class

```
 1 class
 2    S_TEMPLATE
 3
 4 factory -- obtain an instance
 5    default_access: like Current is
 6       once
 7          create Result.instance
 8       end
 9
10 create {NONE} -- private instance creation mechanism
11    instance is
12       do
13          -- initialize fields, etc.
14       end
15
16 end -- class S_TEMPLATE
```

Figure 7.1: A singleton defined in EIFFEL using a factory.

re-classifies `default_access` as a factory, so clients can use the creation instruction **create** 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.

REFERENCES

[1] G. Booch. *Object Oriented Design with Applications*. Benjamin/Cummings, 1991.

[2] T. A. Budd. *An Introduction to Object-Oriented Programming*. Addison-Wesley, 1st ed., 1991.

[3] M. Campione, K. Walrath, and A. Huml. *The Java Tutorial: A Short Course on the Basics*. Addison-Wesley, 2000.

[4] P. S. Canning, W. R. Cook, W. L. Hill, and W. G. Olthoff. Interfaces for strongly-typed object-oriented programming. In *OOPSLA'89*.

[5] C. Chambers. The Cecil language, specification and rationale. Technical Report TR-93-03-05, University of Washington, Seattle, 1993.

[6] L. G. DeMichiel, L. U. Yalçinalp, and S. Krishnan. Enterprise JavaBeans specification, version 2.0. http://java.sun.com/j2ee/, 2001.

[7] ECMA International. *Standard ECMA-367: Eiffel Analysis, Design, and Programming Language*. ECMA International, 2005.

[8] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1994.

[9] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *OOPSLA'03*.

[10] E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[11] J. Gosling, B. Joy, G. L. J. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 3rd ed., 2005.

[12] M. Grand. *Patterns in Java, Volume 1*. John Wiley & Sons, 1998.

[13] I. Jacobson. *Object-Oriented Software Engineering - A Use Case Driven Approach*. Addison-Wesley, 1st ed., 1992.

[14] K. Marx. *Das Kapital: Kritik der politischen Oekonomie*. Otto Meissner, 1867.

[15] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2nd ed., 1997.

[16] N. Nystrom, S. Chong, and A. C. Myers. Scalable extensibility via nested inheritance. In *OOPSLA'04*.

[17] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *CC'03*.

[18] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[19] D. L. Parnas. Information distribution aspects of design methodology. In *IFIP'71*.

[20] B. Shannon. Java 2 platform enterprise edition specification, v1.4. http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf, 2003.

[21] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd ed., 1997.