Creational Design Patterns

Fritz Solms

July 27, 2015

Contents

1 Creational Patterns ........................................... 1
  1.1 Introduction .............................................. 1
  1.2 The singleton pattern ..................................... 1
     1.2.1 Intent .............................................. 1
     1.2.2 Solution ............................................ 1
     1.2.3 Example applications ................................. 2
     1.2.4 Consequences ....................................... 3
     1.2.5 Implementation guidelines ......................... 4
     1.2.6 Related patterns .................................... 6
  1.3 Factory methods ............................................ 6
     1.3.1 Intent .............................................. 6
     1.3.2 Solution ............................................ 6
     1.3.3 Example applications ................................. 7
     1.3.4 Consequences ....................................... 9
     1.3.5 Implementation guidelines ......................... 9
     1.3.6 Related patterns .................................... 10
  1.4 The abstract factory pattern ............................... 10
     1.4.1 Intent .............................................. 10
     1.4.2 Solution ............................................ 10
     1.4.3 Example applications ................................. 11
     1.4.4 Consequences ....................................... 12
     1.4.5 Implementation guidelines ......................... 12
     1.4.6 Related patterns .................................... 13
  1.5 The builder pattern ........................................ 13
     1.5.1 Intent .............................................. 13
     1.5.2 Solution ............................................ 14
     1.5.3 Example applications ................................. 14
     1.5.4 Consequences ....................................... 16
     1.5.5 Implementation guidelines ......................... 17
     1.5.6 Related patterns .................................... 17
  1.6 The prototype pattern ..................................... 18
     1.6.1 Intent .............................................. 18
     1.6.2 Solution ............................................ 18
     1.6.3 Example applications ................................. 19
     1.6.4 Consequences ....................................... 20
     1.6.5 Implementation guidelines ......................... 20
1 Creational Patterns

1.1 Introduction

The patterns discussed in this chapter provide ingenious ways of creating instances of classes. Most of the delegation of object creation out of the object’s class into a separate class whose sole responsibilities usually are that of

- creating instances of the class,
- keeping track of the instances which exist, and
- providing access to instances of the class.

Such classes are often called factory classes.

1.2 The singleton pattern

The singleton pattern is one of those patterns which has been and often still is very widely used, even though one can raise a number of concerns around a simplistic implementation of the Singleton pattern.

1.2.1 Intent

- To ensure that a class has at most a single instance.
- To avoid unnecessary object creation – particularly for stateless objects.
- To delegate the responsibility for maintaining the global access point to the single instance of the class itself.

1.2.2 Solution

The solution proposed by the singleton pattern is that the meta-class defines:

1. one or more constructors, all with access restricted to private access, and
2. a service (for example, getInstance()) which
   - provides users with a handle to the one and only instance of the class, and
   - upon receiving the first request for an instance, creates that instance.

The structure is shown in the following UML class diagram:

Note that

- the constructor has been declared private,
- the meta-class maintains a private reference variable to the instance, and that
- the meta-class provides a getInstance() service which returns a handle to the one and only instance of the class.
The dynamics of the singleton pattern is shown in the following UML activity diagram:
The meta-class service, `getInstance()`, creates the instance upon receiving the first request for a handle to the instance. It maintains a handle to the instance and on subsequent requests simply returns a handle to that instance.

### 1.2.3 Example applications

There are very many potential applications for the singleton pattern. Below we discuss some of the more common usage classes for the singleton pattern.

There are a number of classes which need to be singletons to avoid problems which would occur if multiple instances of the class were around. Examples include

- thread schedulers,
- printer queues,
- telephone answering units.

If instances of a particular class do not carry any state, then any instance is completely equivalent to any other instance. Most of these classes should be declared singletons (a possible exception is that where instances of the class act as threads for a thread pool).

For example, continuous compounding is a class which encapsulates the interest rate algorithms required for interest rates using continuous compounding, like that of converting between an interest rate and a discount factor. The instances of the class do not carry state and hence
one should declare the class a singleton – otherwise one may carry the unnecessary overhead of creating large number of instances of `ContinuousCompounding` for many of the interest rates used, while the different interest rates to which this form of compounding applies could all refer to the same instance of `ContinuousCompounding`.

Other examples are

- sorters,
- database connection pools,
- routers which route requests to different threads in a thread pool or to different machines in a cluster.

Object factories themselves should be singletons. Otherwise no object factory would take control of all the instances.

1.2.4 Consequences

- **Provides controlled access to sole instance:**
  The singleton pattern provides a single access point through which the sole instance is created and through which access to the sole instance is obtained. It avoids using a global variable as access point to the instance and thus the associated name space pollution.

- **May fail in a multi-threaded environment:**
  When multiple threads request handles to the instance of a singleton, it may happen that more than one instance is created. Combining the double locking pattern with the singleton pattern can solve this problem.

- **Too strong binding to the actual class - difficult to decouple from the class when using the singleton pattern - it may be a better solution to use a simple factory and let the factory provide the instance of the actual singleton class.**

- **Too strong binding to the fact that it is a singleton requiring code changes in the client should things change. Again, it may be a better solution to use a simple factory and let the factory behind the scenes decide whether to reuse a single instance, provide an object from an object pool or create a new instance upon request.**

- **Can create instance with information provided at run-time:**
  We can provide separate methods for instance creation and instance access. The former may take initialization information as parameters which is used to create the instance. Both services would not be provided unconditionally – the former would through an `InstanceExists` exception if the constructor service is called iteratively and the second method would throw a `NoInstanceInitialized` exception if the instance is requested prior to having created it.

- **In most cases you do not want to allow subclassing of the singleton class:**
  In many examples (e.g. that of continuous compounding), you would not want to allow the development of subclasses. Some languages like Java allow you to explicitly disallow subclassing.
• Providing specialized instances:
  A singleton could have subclasses and when requesting its instance, one of the subclass instances would be returned – a subclass instance is, after all, also an instance of the class itself. The actual class through which access to one of its specializations is obtained could even be abstract.

1.2.5 Implementation guidelines

• In languages like C++, Java, Delphi and their derivatives (e.g. C#), the concept of a meta-class is not directly supported. Meta-class methods and attributes are mapped onto static members, i.e. class functions and class attributes. The constructor, the instance handle (pointer or reference), and the access provider service (e.g. getInstance()) would in these languages all be declared static. Note: Constructors are implicitly class (static) services.

• The instance handle needs to be declared private. So does the constructor except when specializations are allowed. In such cases the constructor needs to be declared protected.

• The access provider service requires public access.

• You should consider combining the singleton pattern with the double-locking pattern to make it thread-safe.

Below is the listing of a Java implementation of ContinuousCompounding. We make the class a singleton because instances of the class do not carry any state.

```java
import java.io.Serializable;
import java.util.Date;
import java.io.Serializable;
/**
 * Implements the continuous compounding convention which allows users
 * to convert interest rates to discount factors and vice versa.
 * <P>
 * The class cannot be externally instantiated. Instead users obtain a handle
 * to the only instance of the class via
 * <PRE>
 * Compounding compounding = ContinuousCompounding.getInstance();
 * </PRE>
 * @author Fritz Solms
 * @version 1.0 final
 */
public final class ContinuousCompounding implements Compounding, Serializable
{
private ContinuousCompounding ()
/**
 * Receiving an annualized rate, a day count convention and a period
 * defined by a <code>date1</code> and an <code>date2</code> date, this
 * method returns the discount factor used to discount cashflows received
 * on <code>date2</code> to <code>date1</code>.
 * <P>
 * @param annualizedRate the annualized rate (using the compounding type
 * specified by this compounding object
 * @param date1 the date to which cash flows are to be discounted to
 * @param date2 the date from which cash flows are to be discounted from
 */
```
/**
 * @param dayCountConvention The day count convention to be used for calculating the discount factor.
 * @return the discount factor over the specified period for the specified rate for this objects compounding type and the specified day count convention.
 */
public double getDiscountFactor(double annualizedRate, Date date1, Date date2, DayCountConvention dayCountConvention)
{
    double nYears = dayCountConvention.getYearFraction(date1, date2);
    return Math.exp(-annualizedRate * nYears);
}

/**
 * Receiving a discount factor which discounts payments received on <code>date2</code> to <code>date1</code>, a day count convention and the corresponding dates, this method returns the corresponding annualized rate.
 * @param annualizedRate the annualized rate (using the compounding type specified by this compounding object.
 * @param date1 the date to which cash flows are to be discounted to.
 * @param date2 the date from which cash flows are to be discounted from
 * @param dayCountConvention The day count convention to be used for the rate.
 * @return the annualized rate corresponding to the discount factor over the specified period for the specified rate for this objects compounding type and the specified day count convention.
 * @throws IllegalArgumentException when <code>date1==date2</code>.
 */
public double getRateFromDiscountFactor(double discountFactor, Date date1, Date date2, DayCountConvention dayCountConvention)
throws IllegalArgumentException
{
    double nYears = dayCountConvention.getYearFraction(date1, date2);
    if (nYears == 0)
    throw new IllegalArgumentException("discount factor over zero time period");
    if (discountFactor == 1) return 0;
    return -Math.log(discountFactor) / nYears;
}

/**
 * @return a descriptive string.
 */
public String toString()
{return stringID;
}

/**
 * String identifier for this day compounding convention.
 */
public static final String stringID = "continuous_compounding";

/**
 * @return the unique instance of the class.
 */
public static ContinuousCompounding getInstance()
{
    if (theInstance == null)
    theInstance = new ContinuousCompounding();
    return theInstance;
}
private static final ContinuousCompounding theInstance;
1.2.6 Related patterns

The singleton pattern can be combined with most other factory classes – i.e. the factories themselves should typically be singletons. Such combinations may prove very useful for more complex construction processes.

- **Builder:**
  Consider using the *Builder* pattern if you need to create objects through a complex construction process.

- **Factory method:**
  Consider using a factory method if the decision of which class should be instantiated should be deferred to run-time.

- **Abstract factory:**
  Consider using an abstract factory to create members of a concrete realization of a framework – the decision of which family of classes is to be used should be deferred to run-time.

Furthermore, consider combining the singleton pattern with the **double locking pattern** to make it thread-safe.

1.3 Factory methods

The factory method is one of the simplest creational patterns. It is also often called the *virtual constructor*.

1.3.1 Intent

- To specify an interface for creating objects, but to let the instances of subclasses decide which concrete objects are created.

- To support polymorphic instance creation.

1.3.2 Solution

The solution is simply that classes provide their own concrete realization of a factory method, i.e. we simply use method overloading.

In the following UML use case diagram we show the responsibilities of the various components of the factory method pattern.

An abstract context decouples a client from specific realizations of that context and specifies factory methods to be realized by its concrete realizations. Similarly, an abstract product decouples clients from issues which are specific to concrete products for a specific context. The concrete contexts implement the factory method interface which returns the product appropriate for that specific context.

A context specifies the factory methods which the various concrete realizations of the context must provide. The structure of the factory method pattern is specified in the UML class diagram shown below:

**Note:** The client is completely decoupled from any specific concrete realization of a product. The client simply obtains the correct product for a specific context and uses it.
1.3.3 Example applications

Examples of the factory method pattern are scattered across many class libraries. They are often so simple that the solution is taken for granted and the use of the pattern is often not seen.

One of the classical design patterns used in collection libraries is that of an iterator (see the iterator design pattern). Iterators are used to step across the elements of a collection in a way which is independent of the type of collection used.

The responsibility of creating an appropriate iterator for a specific type of collection is typically delegated to the collection itself. When requesting an iterator, the iterator should be constructed polymorphically, i.e., the correct iterator for the collection used should be created without specifying in the code the type.

Thus, if a different realization of a collection is chosen, no changes need to be made on the code

- which requests and
- uses

the iterator.

Note: In the case of the Java 2 Collection Framework, the concrete iterator classes are defined as private inner classes and hence the implementation class of the Iterator interface is completely hidden from the users.

A special case and a very common application of the factory method is that of polymorphically cloning objects via virtual methods of the corresponding class. For example, Java, defines a single-tree class hierarchy with a common superclass, Object, which provides a default implementation of a clone() service which is a polymorphic version of a copy constructor.

Note: Traditional copy constructors do not copy objects polymorphically because constructors are class services which are not resolved polymorphically. For example, the following C++ code snippet

```cpp
class A
```
defines a class, A with a subclass, B, both of which provide a copy constructor. In main we create an instance of B and refer to that instance via a pointer of type A. We then make a copy of the instance of B using the copy constructor. The copy constructor is not resolved polymorphically and the copy of the instance of B ends up being an A. Both class also supply a virtual clone() service which is a factory method. Using this method does result in polymorphic copying of the object and the copy of the instance of B is now also a B.

1.3.4 Consequences

- Decision of which objects are created made at run-time:
  
  In the factory method the decision of which object is created is made at run-time by delegating that decision to the context we are using.
Figure 5: Iterators of a collection framework are often obtained via factory methods

- Connects parallel class hierarchies:
  Parallel class hierarchies occur when members of a framework delegate some of their responsibilities to lower-level classes. For example, the factory methods link the collections with their corresponding iterators.

- Provides subclasses the opportunity to return their own objects:
  The factory method pattern can be used when subclasses may return special service providers which realize services in a non-standard way. Every class may either simply inherit a factory method from its superclass, or may override it to return a specialized object.

1.3.5 Implementation guidelines

Factory methods are usually simply implemented by virtual instance methods which are bound at run-time, leveraging off the polymorphism provided by object-oriented programming languages during run-time linking.

1.3.6 Related patterns

- Singleton:
  The factory method may return a singleton.

- Abstract factory:
  Abstract factories are often implemented with factory methods. Here each family of classes provides its own concrete realization of a concept and the responsibility for returning an
concept is delegated to a factory method for that concept. For example, each concrete presentation layer framework (e.g. Swing, AWT or MIDP) provides its own presentation of an order obtained from a factory hosted by the factory of that framework.

- **Template method:**
  Template methods define high-level work flows whose concrete steps may be realized in different ways. The components to whom the realization of these concrete steps is delegated are often created via template methods.

### 1.4 The abstract factory pattern

The abstract factory is a very useful high-level pattern which enables one to decouple an application from the concrete implementation of an entire framework or library.

#### 1.4.1 Intent

- To be able to choose at run-time a specific family of classes and to delegate the creation of the appropriate objects from that family to a factory whose sole responsibility is to instantiate the members of a particular family of a classes.

- To be able to replace an entire family of classes with another family by changing the choice of factory – for example to replace a particular concrete realization of a framework with another concrete realization.

- To decouple clients from the specific family of object they are using.

#### 1.4.2 Solution

The solution involves encapsulating

- the requirements specification for the products which are going to be produced in a collection of abstract products, and

- the requirements specification around which objects the various concrete factories need to be able to produce and what information they will receive in order to construct them in an abstract factory.

The abstract factory itself specifies the types of objects (i.e. realizations of which abstract objects) the various concrete factories must be able to create. Each family of classes has its own factory which produces classes which realize the various abstract products.

The structure of the abstract factory pattern is specified in the UML class diagram:

- Often the various families of classes represent different realizations of the same framework.

For example, they could represent different realizations of

- a user interface framework,

- a persistence framework, or

- a pricing framework.
1.4.3 Example applications

The abstract factory has very wide applicability and is used particularly when one wants to abstract from a concrete realization of a framework in order to be able to plug in different realizations of that framework.

One can use the abstract factory pattern to abstract from any concrete realization of a presentation layer. For example between Java Swing, AWT, or J2ME-based presentation layers or between presentation layers for X-Windows, Windows and Apple-Mac platforms.

For example, the factory could contain a service for creating a presentation of an order. In the case of Swing or AWT realizations, the presentation may potentially be a single screen. In the case of a J2ME presentation running on a mobile phone the presentation may span multiple screens with navigation functionality across them. Client applications are abstracted from these, leaving the presentation layer functionality (like navigation) to the realization of the presentation layer component.

In a similar way the abstract factory pattern enables one to abstract from the concrete representation of a persistence mapper. One could switch between an XML-based persistence layer hosted, say, in a flat file to an object-relational mapping layer persisting the object structure onto a relational database to a record-store based persistence layer available on some PDAs without changing the business logic layer or any other aspects of the application.

The Java API for XML-based Processing, JAXP, uses an abstract factory to abstract from different realizations of an XML parser.

1.4.4 Consequences

- The Abstract Factory pattern provides a simple mechanism to exchange one family of classes with another.
- Clients are decoupled from the concrete classes they are using.
• Consistency among different products supporting a particular application domain is enforced.

• Changes to the way in which products are created (i.e. changes to constructor arguments) have to be rippled through the abstract factory as well as all concrete factory implementations.

• Extending the product range is cumbersome because the product must be added to the abstract factory as well as to each concrete factory.

• The factories may become very complex.

1.4.5 Implementation guidelines

The implementation of the abstract factory pattern is a straightforward mapping of the UML diagrams onto the implementation language of your choice.

• The factory itself should be made a singleton:
  Typically it is sufficient and often desirable that all instantiation of classes from a particular family of classes is done through a single instance of a concrete realization of an abstract factory.

• You can specify the factory to be used in a properties file:
  To exchange one family of classes with another (e.g. one concrete realization of a framework with another) it may be sufficient to read the name of the concrete family to be used from a properties file. The concrete factory which produces the concrete products can then be instantiated via something like

```java
java.lang.reflect.Method getInstance
  = Class.forName("concreteFactoryName")
    .getMethod("getInstance", null);

// No invoke getInstance on a null object with no arguments in order
```
The above code extract

1. instantiates a class descriptor for the concrete factory from the class name which may have been provided in a properties file,
2. requests the `getFactory` method as an object via Java’s reflection mechanism,
3. invokes that service directly without providing a handle to an instance (after all, this is a class service) and without providing any parameters,
4. and keeps an abstract factory interface handle to the concrete factory instance obtained in this way.

**Note:** *No code changes need to be made to switch between concrete realizations of a framework. It is sufficient to change the class name for the concrete factory of choice in a properties file.*

### 1.4.6 Related patterns

- **Singleton:**
  One typically implements the concrete realizations of an abstract factory as singletons.

- **Builder:**
  The concrete factories of an abstract factory are responsible for creating all the members of a framework. If the construction of any of these objects is non-trivial, the factory can assign the responsibility of creating the complex objects to separate builders (see the Builder design pattern).

### 1.5 The builder pattern

The builder pattern is one of the most useful creational patterns introducing clean decoupling of different domains and facilitating the incremental construction of complex objects.

#### 1.5.1 Intent

- To separate the high-level construction process of a complex, aggregate object from both,
  - the concrete construction of the individual components and
  - the assembling of these components into a product

  such that the same construction process can construct different representations of the same conceptual object.

- To separate the responsibilities of understanding the source domain containing the information from which the objects are constructed from logic required to construct the products.
1.5.2 Solution

The director consumes the information used to construct a product and issues high-level instructions for the construction process to some or other concrete builder which provides the construction services needed to build a product.

One of the core features of the builder pattern is that it *separates the source and realization domains*. The director only needs to understand the source domain, i.e. the information from which the objects are created while a separate builder is assigned to manage each realization domain.

The following UML use case diagram shows the responsibility allocation across the various components of the builder pattern.

![Figure 8: The responsibility allocation for the builder pattern](image)

The structure of the builder pattern is specified in the following UML class diagram:

**Note:** The director is completely decoupled from any specific concrete realization of a builder and vice versa. It directs the construction process at a higher, more abstract level.

The client creates a concrete builder and subsequently a director which uses that concrete builder. It then requests the director to construct a product which it does by requesting the builder to construct part for part.

The concrete builder keeps track of the construction process and the constructed product and ultimately the client requests the constructed product from the builder. The dynamics of the builder pattern is shown in the following UML class diagram:

1.5.3 Example applications

The builder patterns is used quite widely to abstract the concrete realization of a building process from the more abstract instructions.
One can find the use of the builder pattern in real-life processes like those in fast food restaurants which use the builder pattern to construct, for example, childrens meals with a meal, say, consisting of a main item, a side item, a drink, and a toy (e.g. a hamburger, chips, liquid fruit, and water pistol).

There are variations in the content of the children’s meal, but the construction process is the same. Whether a customer orders a hamburger, cheeseburger, or chicken drumsticks, the employee at the counter directs the crew to assemble a main item, side item, and toy. These items are then placed in a box with the drink being placed in a separate compartment within the box. Even different fast food outlets placed in a cup and remains outside of the bag. This same process is used at competing restaurants.

From a UML diagram we can generate, for example, an implementation in one of a number of programming languages. Alternatively one can generate an XML schema from a UML class diagram. For each code generator for a particular programming language and the XML schema generator could be a different realization of a builder. This example is illustrated below:

XML docbook renderers read XML docbook tags and generate a rendering in one of a number of technologies. This may include

- HTML,
- PDF,
- LaTeX,
- or even an Open-Office document.

Here the director reads the XML tags (e.g. chapter, section or table tags) and requests a concrete builder to construct a rendered document from these tags. Different concrete builders construct different realizations of the rendered document.

When using entity beans with container managed persistence, one specifies the query in EJB Query Language, a persistence technology neutral, object-oriented query language. These queries are mapped onto queries in the persistence technology chosen for that entity bean.
For example, if a relational database is used, then SQL queries are constructed from EJB-QL queries. On the other hand, if the relational database is replaced by an object database, then the SQL-builder can be replaced by an OQL builder.

Parsers typically read an input domain and construct a representation in another domain. For example, parsers of a mathematical expression decode the input domain and construct an algorithm for that expression. A builder pattern will separate the expression decoding from the algorithm construction and facilitates different algorithm constructors for different programming languages.

### 1.5.4 Consequences

- **Separation of source and destination domains:**
  The director has to only understand the source of the information from which the product is created while the builders only need to understand the construction of the product parts as well as the assembling of these parts into products.

- **Pluggable builders:**
  The pattern allows you to plug in different concrete builders which construct potentially different realizations of a product.

- **Pluggable directors:**
The pattern allows you to plug in different concrete directors which construct products from potentially different concrete source domains (e.g. construct a Java data object from either a UML class diagram or a XML schema type).

- **Modeling incremental product construction and assembly:**
  The builder models step-for-step product construction and assembly directly. As such the builder pattern provides control for the construction process.

### 1.5.5 Implementation guidelines

The implementation of the builder pattern is for object-oriented languages a straight-forward mapping of the UML diagrams onto code.

- **Should Builder be an interface or an abstract class?**
  We would recommend to always have the contract for the builder represented by an interface. You could use, in addition to this, an abstract class to encapsulate certain commonalities across some of the concrete builders.

- **The builder interface is determined by the source domain:**
  It is important that the Builder interface design is not driven by the requirements of a particular builder constructing a specific realization of a product. Instead the Builder should specify services as required by the source domain, i.e. as required by the Director.

- **Products may have little in common:**
  The different products produced by different builders may, from a user perspective, have at times only very little in common and as such the product interface may specify very few functionalities, perhaps even only the ability to obtain a streamable byte representation of the product.

### 1.5.6 Related patterns

- **Template method:**
  In many ways the builder pattern is a specialization of the template method pattern. Both patterns define a high-level algorithm where the individual steps of the algorithm may be realized in different ways.
• Singleton:
  Both, the director and the builder may be a singleton.

• Abstract factory:
  Abstract factories may delegate the responsibility of constructing complex components of
  a family of classes (e.g. instances of classes from a framework) to a builder.

1.6 The prototype pattern
A prototype produces objects by cloning (i.e. copying) a prototype of an object. The pattern
is widely used graphical design packages like electronic circuit design, UML modeling packages
and visual builders.

1.6.1 Intent
• To enable clients to select the type of object to create by selecting a prototype.
• To create instances of the selected class by copying that prototype.

1.6.2 Solution
The solution proposed by the prototype pattern is very simple. Clients select a class they want
to instantiate by selecting a prototype. Instantiation is then done polymorphically by cloning
the prototype.

In the following UML use case diagram we show the responsibilities of the various components
of the prototype pattern.

Figure 12: The responsibility allocation for the prototype pattern

The structure of the prototype pattern is specified in the UML class diagram shown below. Note that the director is completely decoupled from any specific concrete realization of a
prototype. It directs the construction process at a higher, more abstract level.
1.6.3 Example applications

Prototypes enable clients to select at run-time what objects they want to create. The prototype pattern provides a simple solution for facilitating dynamic object creation and run-time management of a registry of objects.

Prototypes are widely used in graphical editors like general drawing applications, UML editors, applications for electronic circuit design and so forth. Clients select which objects they want to create from a panel of icons. Each icon acts as a proxy for a prototype object which is clones upon selection.

Static programming languages like Java, C++, C# and Delphi are quite rigid. The behavior of an object is defined by a class and that behavior can only be changed by subclassing. Certain patterns aim to break the static nature of these languages. For example,

- the decorator pattern enables you to add extra responsibilities to an existing service,
- the state pattern enables you to specify different service realizations for different object states,
- the exposed state pattern enables you to add additional services which become available when an object is in a specific state,
- and the Visitor pattern enables you to add additional functionalities across a class hierarchy, even at run-time.

Prototype-based programming languages like Self do not introduce the concept of a class. Instead they only supply objects, but enable you to add services and attributes on the fly. New objects are created solely from cloning. Thus, the root object is cloned and that clone can evolve over time, generating further clones with different services and attributes. Note: Self was designed by David Ungar and Randall Smith in 1986. A paper describing the language appeared at OOPSLA ’87. An initial implementation was undertaken at Stanford University in 1987. After that, Self was used as a vehicle for language, environment, user interface and implementation research (as we shall see, some of the features of Self pose challenging implementation problems).

1.6.4 Consequences

- **Clients create objects generically:**
  Clients can create objects without having had the corresponding classes available at compile-time.

- **Can add and remove objects at run-time:**
  Objects can be dynamically added to or removed from the template collection. Thus new classes can be made available to the client even at run-time.

- **Can facilitate construction of composite objects:**
  The objects made available through the template collection may include components which may be added to composite objects. For example, an architectural drawing application may enable you to add doors, windows, or electrical outlets to a room. Furthermore, you may be able to add shutters to a window.

- **Makes static languages like Java, C++, C# and Delphi more flexible:**

- **Can modify default state for objects dynamically:**
  Since all objects are created by cloning a prototype, the default state for the objects generated from the prototype can be modified dynamically by modifying the state of the prototype.

1.6.5 Implementation guidelines

Although the prototype pattern is structurally virtually trivial, it does pose some complexities during implementation.

The core complexity faced when implementing the prototype pattern is the implementation of the clone() services. Often the terms deep and shallow copying are used. This is typically not very helpful since, where do you go deep and where do you remain shallow. These questions need to be solved from a design perspective and a solid UML model will specify concrete implementation requirements.

Deep copying should be done on composition relationships while aggregation and association should be implemented in the clone service via shallow copying.

The state of the prototype object should be managed to resemble at any stage the default state of the objects which are created from it.

Typically one should introduce a prototype manager who manages the prototypes available for object creation. The prototype manager should supply services for

- registering new prototypes and
- de-registering existing prototypes.

The prototype manager is typically implemented as an associative store associating user handles (e.g. dragable icons) with prototype objects. Clients may be able to browse and modify the prototype registry at run-time.
1.6.6 Related patterns

• **Factory method:**
  The clone service provides a factory method for the class itself, i.e. the context and the product of the factory method pattern are the same class.

• **Abstract factory:**
  Abstract factories enable clients to choose at run-time between different realizations from different families of classes. Prototypes enable clients to register new objects at run-time allowing clients to create instances of the corresponding classes by copying the prototype object.

• **Strategy pattern:**
  Strategies are computational algorithms which may be selected at run-time. The prototype pattern may facilitate the algorithm selection for strategies.

1.7 Exercises

1. Consider a stock-management system for a spare-parts manufacturer. Parts may be assembled from finer grained parts. Products are parts which can be sold and hence have a price. Try and identify potential uses for the creational patterns discussed in this chapter.

2. Now consider an ordering module for the system where sales staff can place orders via a traditional GUI-based client application, via a web site or via a PDA or cell-phone based application. Try and identify places where you could use the creational patterns discussed in this chapter.