CHAPTER **2**

Object-Oriented Software Engineering

Object-oriented programming —with its improved programming productivity, enhanced modelling power and better program maintenance— is among the most promising subfields of software engineering. This chapter investigates to what extent the concepts and techniques proposed by the object-oriented community provide solutions for tackling the tailorability of open hypermedia problems.

Incremental Development

What is Software Engineering ?

According to [Brooks'87] software systems are more difficult to build than any other kind of systems built in any other engineering discipline because their inherent complexity, non-conformity, changeability and invisibility. Software engineering is the discipline that investigates techniques to deal with those inherent difficulties of manufacturing software.

incremental development, is generally regarded as one way to cope with the inherent difficulties. Rather than regarding a software system as a static device that can not be changed once it left the design table, incremental development views a software system as a flexible LEGO construction. This way, running software systems can be easily adapted to the unpredictable evolution of their environment.

Object-oriented software engineering is the subfield of software engineering that has taken this incremental development approach as one of its main subjects of study.

What is Object-Oriented ?

It is quite conventional to answer the question "What is Object-Oriented ?" in a bottom-up manner, explaining basic object-oriented concepts (i.e. objects, messages, classes, inheritance and composition) and how they fit together. The problem with bottom-up answers is that they depend heavily on the subject the question is referring to. For example, the answer to the question "What is an object-oriented programming language ?" is quite different from the answer to the question "What is an object-oriented user-interface ?" (or a database, or a drawing package or a hypermedia system for that matter).

With the enormous amount of material that has been published on object-oriented software engineering, we assume the reader is familiar with the subject. So we tackle the question from a slightly different perspective and reformulate the question into "What does it mean to be object-oriented ?". Answering the reformulated question leads almost naturally the proclaimed benefits of object-oriented software engineering.

So what does it mean to be object-oriented ? According to [Goldberg,Rubin'95], p.46 "*something is object-oriented if it can be extended by composition of existing parts or by refinement of behaviours. Changes in the original parts propagate, so that compositions and refinements that reuse these parts change appropriately*".

This definition can be applied on the available software construction tools. A language is object-oriented if it allows to express composition of existing parts, refinement of behaviours and propagation of changes. Thus, encapsulation, inheritance and polymorphism make programming language object-oriented. Likewise, a database is object-oriented if it supports the storage of object models, i.e. data models that support encapsulation, inheritance and polymorphism.

But the definition can also be applied on running software systems. A drawing package is object-oriented if it supports the composition of existing parts (i.e. grouping), the refinement of behaviours (i.e. dragging or resising depends on the object one is manipulating) and propagation of changes (i.e. attributes like colour or size propagate through components of a composite). A user interface is object-oriented if allows to express the composition of existing parts (i.e. folders in desktop environments) and refinement of behaviours (i.e. highlighting behaviour changes according to the target user interface element) and propagation of changes (i.e. moving folders move their contents).

This definition is appealing because the composibility, refinement and propagation requirements map very well onto the incremental development approach. In the object-oriented community, different techniques have been forwarded to satisfy these requirements. Object-oriented frameworks and meta-object protocols are two of the more prominent techniques.

Object-Oriented Frameworks

A popular way to look at the progress in software engineering is to see every step as providing higher level abstractions for larger entities and techniques. Assembly language instructions were assembled into control structures, control structures were gathered into procedures, procedures were gathered into abstract data types, and with the advent of object-oriented programming we know how to gather abstract data types into inheritance hierarchies. Object-oriented frameworks are probably the next step in this evolution, where we learn how to gather inheritance hierarchies into reusable designs.

Object-Oriented Frameworks

a) Definition

An object-oriented framework is a state-of-the art technique in object-oriented software engineering. The field has reached consensus over the following definition: "*A framework consists of a set of co-operating classes and objects that make up a reusable design for a specific problem domain*" [Johnson,Foote'88], [Wirfs-Brock,Johnson'90], [Johnson,Russo'91], [GammaEtAl'95], [Goldberg,Rubin'95], [Cotter,Potel'95].

b) Categories of Frameworks

Many examples of object-oriented frameworks can be found in the literature. They can be classified in three categories.

Application Frameworks

These kind of frameworks make up a reusable design for the construction of applications with a graphical user-interface. Some well-known examples are Apple's MacApp [Schmucker'86], Borland's OWL, ET++ [WeinandEtAl'88] and Parcplace's VisualWorks.

Domain Specific Frameworks

These kind of frameworks make up a reusable design for a highly specialised domain. Compared to application frameworks that aim at horizontal markets, domain specific frameworks aim at vertical markets. Examples are Choices [CampbellEtAl'93] for the domain of operating systems, ET++SwapsManager [Birrer,Eggenschwiler'93] for the domain of financial engineering and DHM [Grønbaek'94] for the domain of open hypermedia systems.

Support Frameworks

Support Frameworks are similar to application frameworks but aim to support functionalities in between the operating system and the application (i.e. peripheral devices, databases, …). A prominent example is Taligent's CommonPoint [Cotter,Potel'95].

c) Object-Oriented Frameworks versus Class Libraries

According to the definition, frameworks are about reusing objects and classes, so one could confuse them with object-oriented code libraries. The crucial distinction between a framework and a class library is the idea of a reusable design. We elaborate on this distinction by adopting a list of differences from [Cotter,Potel'95] (p.57).

Behaviour versus Contracts

Class libraries are essentially collections of behaviours that can be called when needed. A framework provides collections of behaviours as well, but the essential add-ons are the framework contracts: the set of rules that governs the ways in which behaviours can be combined. Among others, the framework contracts include rules about what the framework provides, what the caller of the framework must provide, what is allowed in subclasses, etc. We elaborate on this notion of framework contracts in a later section (see framework contract, p.51).

Don't Call Us, We'll Call You (The Hollywood Principle)

With a class library, client programmers instantiate classes to create library objects and call their methods afterwards. Most frameworks do it the other way round: client programs pass their objects (in most cases these objects inherit from objects part of the framework) and it is the framework that calls the methods of the client objects. This way, the framework defines large parts of the control flow of an application and clients reuse it in their applications. To work properly, this scheme requires that clients supply legitimate objects, hence the necessity of framework contracts.

Implementation versus Design

The emphasis of class libraries lies on the reuse of implementation or code reuse, while the emphasis of frameworks lies on the reuse of design. Frameworks represent a generic design solution for a variety of problems called the domain (or application domain, problem domain). A framework is not a single program, but embodies a family of related programs and the common factor between these related programs is the framework's design expressed in the framework contracts. Each program in the family of related programs adapts this design to a specific problem, hence why frameworks are often called semi-finished applications.

Implementing Frameworks

a) Abstract Classes and Template Methods

A framework is a reusable design for a specific problem domain. The reusable design is specified with number of semi-finished inheritance hierarchies, constructed using so-called abstract classes and template methods, which are defined in terms of abstract and concrete methods.

An abstract method is a method without an implementation and an abstract class is a class defining at least one abstract method. Abstract classes are not allowed to have instances, since it is an error to call methods without an implementation. A method that overrides an abstract method by supplying an implementation is called a concrete method. A concrete class is a subclass of an abstract class that has overridden all of the abstract methods in its superclass chain. As opposed to abstract classes, concrete classes may be instantiated. The roots of inheritance hierarchies are usually abstract classes, while the leafs are usually concrete classes. Note that it is a legal operation to create an abstract subclass of a concrete class, so in the middle of inheritance hierarchies one may find both abstract and concrete classes.

A template method is a method with an implementation that calls abstract methods or other template methods. As such, a template method forms some kind of a skeleton program, where the abstract methods must be replaced by the appropriate concrete methods to make it complete. This 'skeleton program' idea is the way a framework specifies aspects of the reusable design and they are defined by means of template methods.

To make the definition of a abstract classes and template method concrete, we work out a (partial) framework for the domain of direct manipulation user interfaces. Such a framework includes a design for modelling the behaviour of sensitive regions like buttons and window close-boxes (see figure 9). When the mouse button is pushed outside the region nothing happens (1). If the mouse button is pushed while the cursor is within the region, some kind of highlighting is performed to give the user feedback (2). As long as the mouse button is down, the sensitive region adapts its highlighting depending on whether the mouse enters or exits the region (3). Once the mouse-button is released, the highlighting is undone and some action is performed depending on whether the cursor was inside or outside the sensitive region (4).

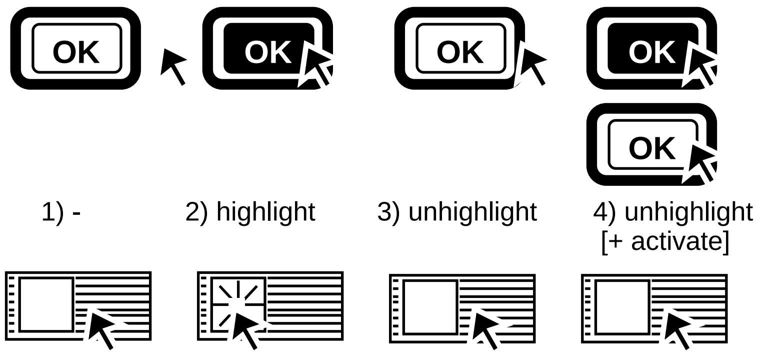
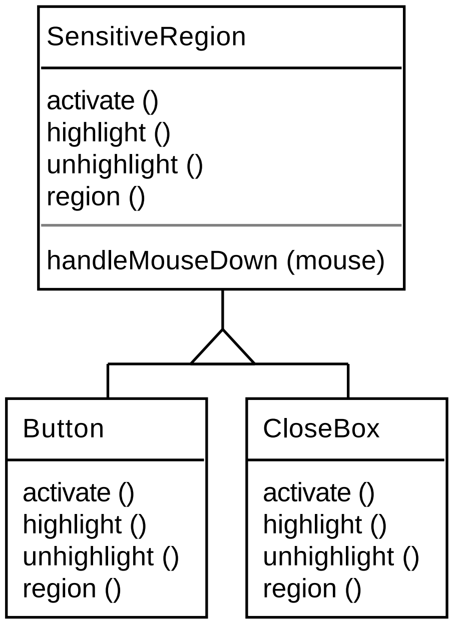


Figure 9: Sensitive Regions in a User-Interface

There are several ways to build a framework for a sensitive region and one way is presented in figure 10. There we have an abstract class `SensitiveRegion` with abstract methods `activate`, `highlight`, `unhighlight` and `region`. There is also a template method `handleMouseDown` that calls these abstract methods to model the mouse tracking behaviour. Subclasses of the sensitive region must provide different concrete methods to model their specific kind of highlighting and we included a subclass for a button `Button` and one for the close box of a window `CloseBox`.



```
method: handleMouseDown (mouse)
if contains(region(self),
            position(mouse))
  highlight(self)
  while buttonPressed(mouse)
    if contains(region(self),
                position(mouse))
      highlight (self)
    else
      unhighlight (self)
    endif
  endwhile
  unhighlight (self)
  if contains(region(self),
              position(mouse))
      activate (self)
  endif
endif
```

Figure 10: The `handleMouseDown` Template Method

b) White-box & Black-Box Reuse (Class Inheritance & Object Composition)

Frameworks are semi-finished programs: the idea is that clients reuse the framework design by filling in the open ends (i.e. supply concrete methods) of the framework. The two most common techniques to supply these methods are class inheritance and object composition but in terms of framework extensions the techniques are usually referred to as white-box reuse and black-box reuse [GammaEtAl'95].

For white-box reuse one relies on class inheritance within object-oriented programs. With white-box reuse a template method calls methods defined on the self object only; to extend the framework one supplies a concrete subclass of the class that implements the template method and overrides the appropriate abstract methods. The `handleMouseDown` method is an example of such a template method that is completed in the subclasses `Button` and `CloseBox` figure 10 by providing concrete implementations for the methods `activate`, `highlight`, `unhighlight` and `region`.

For black-box reuse one relies on the composition of objects. The template methods calls abstract methods defined in other classes and the concrete methods are provided by subclasses that are outside the inheritance hierarchy of the template method. In the example the `handleMouseDown` template method (see figure 10) calls the `contains` method, which is an abstract method defined on the class `Region` (not depicted). The `handleMouseDown` template method is reused by providing concrete implementations for the contains method in concrete subclasses like `Square` for the `CloseBox` and `RoundedCornerRectangle` for the `Button`.

The names white-box reuse and black-box reuse refer to the visibility of the implementation. White-box reuse implies subclasses that can (and must) look inside the implementation of the class defining the template method. With black-box reuse, the internal details of the classes defining the template method are completely hidden from the places where the framework must be extended.

In the remainder of this dissertation we use the term *white-box template method* to refer to a template method that implements a white-box reuse style, i.e. a template method that calls an abstract method defined on the self object. The term *black-box template method* refers to a template method that implements a black-box reuse style, i.e. a template method that calls an abstract method defined on another object.

c) White-box versus Black-Box Reuse

We illustrate the benefits of both approaches with an example. Suppose we must extend our user interface framework so that sensitive regions react to keyboard events. A white-box reuse approach involves the definition of an extra template method (`handleKeyDown`) that polls the keyboard instead of the mouse. A black-box reuse approach involves the definition of a new abstract class (`Tracker`) with one abstract method `handle`. The `Tracker` class has subclasses for all possible input-devices that may control the sensitive region, in our example `MouseTracker` and `KeyboardTracker`. The concrete method `handle` defined on the `MouseTracker` class is a minor rewrite of the older `handleMouseDown` method and the concrete `handle` method in the `KeyboardTracker` class is similar to the newly defined `handleKeyDown` method (hence the grey arrows in figure 11).

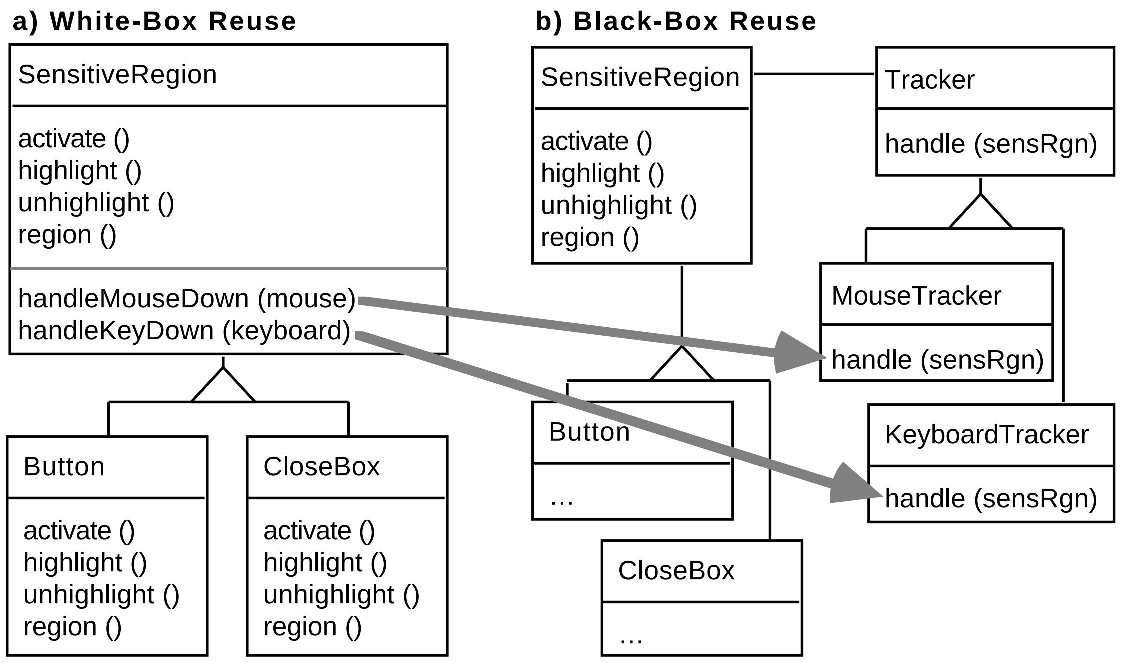


Figure 11: White-box versus Black-Box Reuse

The framework literature agrees that white-box reuse and black-box reuse both have their advantages and disadvantages [Johnson,Foote'88], [GammaEtAl'95], [Goldberg,Rubin'95].

Complexity

White-box reuse is more straightforward than black-box reuse because inheritance is better supported than composition. White-box reuse leads to a little number of large inheritance hierarchies which are usually easier to understand. Black-box reuse results in many smaller inheritance hierarchies with numerous and complex interrelationships in between them, thus difficult to understand. In the example, the white-box approach involves writing one additional method, while the black-box approach involves the creation of three additional classes and adjusting two methods.

Capability

White-box reuse offers more capabilities than black-box reuse because one can use internal implementation details when adapting the superclass. In the example, the window close-box and button may choose to override the `handleMouseDown` and exploit knowledge about the shape of the sensitive region to write a smarter polling algorithm. Indeed, the generic polling algorithm in the `handleMouseDown` method is written in such a way that it can handle disjoint regions by calling the `contains` function in each step of the polling loop. But a button and a close-box have a plain rectangular shape, and so the polling algorithm can be written faster by checking the distance the mouse has moved.

Encapsulation

White-box reuse breaks encapsulation because subclasses can look inside the implementation of the superclasses. Breaking encapsulation is sometimes necessary (see the capability argument), but is generally considered a bad thing as code tends to depend on internal implementation decisions and thus less extensible and harder to maintain. In the example, if the window close-box and button chose to override the `handleMouseDown` method then a change to the internal implementation of the basic `handleMouseDown` method in the `SensitiveRegion` class involves checking all subclasses to see whether they override that method.

Extensibility

For complex designs, black-box reuse usually involves less classes but more objects. This implies that black-box reuse is more extensible because classes are more task-specific and thus easier to subclass. Imagine that we must extend our user-interface framework to support other input devices like joy-sticks and three-button mice. The white-box approach requires us to add a new method to all of the `SensitiveRegion` subclasses for each new input-device that must be supported and there is little chance that we can reuse implementations of other input devices. In the black-box approach it is sufficient to add a new `Tracker` class, which is likely to be a subclass of an input-device with similar behaviour.

Configurability

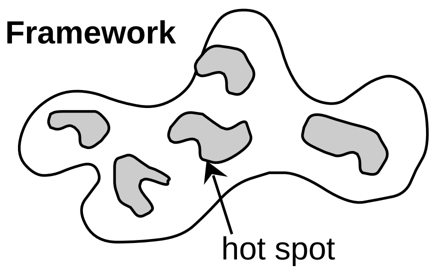
The inheritance chain of an object depends on its class and usually it cannot be changed at run-time. As white-box reuse depends on the inheritance hierarchy, one cannot change the reuse strategy at run-time. With black-box reuse, changing the association between the two objects is all that is required. Consider the example of a pop-up menu as a sensitive region where a user is asked to select an item out of a list. Some user interfaces allow the user to switch between the keyboard and the mouse while the menu is open. With black-box reuse, this can be modelled quite easily by associating two `Tracker` objects with a pop-up menu.

In general the advise is to favour black-box reuse over white-box reuse (see [Johnson,Foote'88] p.6; [GammaEtAl'95] p.20; [Goldberg,Rubin'95] p.208-209), especially since black-box reuse allows to reconfigure the system at run-time. However, black-box reuse distributes the design over a number of inheritance hierarchies and the system's behaviour depends on dynamic object relationships rather than that on static class relationships. This makes frameworks difficult to understand and difficult to build, hence the need for support.

Framework Development Techniques

a) Iterative Development

Figure 12: Framework & Hot Spots



A framework provides a skeleton application for a particular domain and uses abstract classes and template methods to anticipate those aspects of the domain that should be flexible. [Pree'94] coins the term 'hot-spot' to refer to an aspect of the domain that varies between applications and should be represented by an abstract class or method in the framework.

The inherent difficulty in developing frameworks is to identify the appropriate hot spots of the domain, because that is what makes up the reusable design. Framework researchers agree that finding these hot-spots requires domain-specific knowledge and iteration. The former is not a surprise, considering the domain-specific nature of frameworks; the latter needs a little more explanation as it is essential in the framework development process.

Developing a reusable framework requires that a framework be used and reworked several times to be confident in its quality and to make sure that the proposed hot spots correspond with reusable domain aspects. Iteration involves testing the framework in different applications and rework parts of the framework to remove mistakes and make improvements. Reworking the framework is sometimes called refactoring and shifts from white-box reuse to black-box reuse; i.e. tearing operations apart, moving methods up or down the hierarchy, rethinking interfaces, splitting classes and assembling others, …

b) The Agora, ApplFLab and Zypher Design Spaces

Domain knowledge is important to identify the hot-spots of a framework. In the PhD. dissertation [Steyaert'94], design spaces are identified as a particularly interesting way to look at problem domains with the intention to construct frameworks. A design space, captures the varying aspects of a problem domain in an n-dimensional axis system, where each axis represents a fundamental characteristic of all systems in the domain that varies depending on the exact system one is modelling. As such, a design axis corresponds with an important hot-spot of the problem domain.

Some experiments in our laboratory serve as a 'proof-of-concepts' for the design space abstraction. The Agora [Steyaert'94] design space captured the domain of object-oriented programming languages by proposing a two-dimensional design space with an object axis and a message-passing axis. The ApplFLab [SteyaertEtAl'94], [SteyaertEtAl'96] design space captured the domain of user interface builders with a two-dimensional design space based on a widget axis and a value axis. The "Zypher design space" (p.30) captures the domain of hypermedia systems with a three-dimensional —storage, presentation and navigation— design space.

c) Framework Contracts

Frameworks are reusable designs and to reuse the design, clients must respect a set of rules telling how framework components must be used to work properly. Some of these rules can be inferred from the source code directly. For instance, abstract methods proclaim that framework clients should override that particular method. Types specify the minimal interface an object should respond to [Lucas,Steyaert,Mens'95]. However, frameworks favour black-box reuse and black-box reuse results in behaviour that depends more on dynamic object relationships than on static class relationships. This implies that most of the framework design rules cannot be inferred from the framework's source code, hence the need for additional techniques.

A contract [Helm,Holland,Gangopadhyay'90] is a mechanism for the explicit specification of dynamic object relationships and interactions. A contract defines a set of participants and their contractual obligations. The contract participants are objects with a well-defined interface. The contractual obligations state how contract participants must perform a sequence of actions and make certain conditions true in response to receiving messages part of their interface. A contract also defines invariant parts that contract participants should maintain. There exists a formal notation for contracts (see [Helm,Holland,Gangopadhyay'90]) but this is beyond the scope of this dissertation.

To illustrate the idea of a contract we give an example of the contractual obligations between the `SensitiveRegion` and `Tracker` abstract classes as introduced in figure 11. Note the rule that calls to the `highlight` method must be matched by a call to the `unhighlight` method, a constraint that never can be inferred from inspecting the framework source code.

SensitiveRegion

> A sensitive region object represents an area on the screen that is sensitive to certain input devices. Sensitive regions can be highlighted to give the user feedback. Sensitive regions have an action associated with them, that is executed when the input device activates the region.
>
> All sensitive region object implements an `activate` method sent by a tracker object to request the execution of the associated action. All sensitive region objects implement a `highlight` and `unhighlight` method sent by a tracker object to highlight (or undo the highlighting) of the sensitive region. All calls to the `highlight` method must be matched by a call to the `unhighlight` method.

Tracker

A tracker object represents an input device; input devices can activate sensitive regions on the screen.

All tracker objects implement a `handle` method; the caller of the method must provide a sensitive region object as parameter. The tracker objects calls the `highlight` method and then polls the input-device, repeatedly calling the `unhighlight` /`highlight` methods depending on the state of the input device it represents. At some point in time, the input-device must reach a state where it is finished working with the sensitive region, in which case the tracker object calls the `unhighlight` method. Depending on the state of the input-device when finishing, the tracker object may call the `activate` method to execute the action associated with the sensitive region.

d) Design Patterns

A frameworks is a reusable design and to reuse the design, clients must respect a set of rules known as the framework contracts. Yet, contracts are only a partial answer to the problems of framework development.

First of all, contracts do not tell why a certain rule is important, which hinders framework refactoring. Also, contracts fail to provide alternative designs, which hinders efficient reuse. Finally, contracts are very complex and difficult to write, so one wants to reduce the effort by reusing experience from other contracts in other frameworks dealing with similar problems.

Design patterns form an answer to these problems. Design patterns attempt to record the experience of expert framework designers by systematically naming, explaining, and evaluating important and recurring object-oriented design problems and their associated solutions [GammaEtAl'95]. A design pattern is a written document containing at least four elements: a name, a problem, a solution and list of consequences. The name is important to build a common design vocabulary in a group of framework developers. The problem section is necessary to understand when a design pattern is applicable; usually this part of the document contains some kind of analysis and an example. The solution section is a generic description of an object-oriented architecture that solves the problem. The consequences section include a discussion of the advantages and disadvantages of applying the pattern and motivates why the solution answers the problem.

Design patterns are collected in design pattern catalogues. Such catalogues make it easier to reuse a successful design and architecture, and are helpful for learning how to construct reusable designs. Moreover, a software engineer acquainted with a given design pattern applied in a new framework, encounters fewer problems understanding the framework and thus reusing it. Catalogues are useful for documenting frameworks as they build a common design vocabulary.

There are several attempts to assemble design pattern catalogues and they are classified in two categories. Domain specific catalogues collect patterns about a specific problem domain; HotDraw [Johnson'92], [Beck,Johnson'94] is a well-known example for the domain of graphical editors. Generic catalogues collect domain independent patterns: the most prominent example is [GammaEtAl'95]; [Pree'94] collects meta-patterns as all possible combinations for template methods.

The most appealing characteristic of a design pattern (i.e. a description of an object-oriented solution in an abstract, language independent way) is precisely its largest deficiency: design patterns are recorded using informal prose which makes them vulnerable for misinterpretation. Design patterns must be used in conjunction with other framework techniques such as framework contracts.

e) Object Factories

Experience has learned that it is very important to have flexible ways to create and compose objects within frameworks. This should not come as a surprise: frameworks favour black-box reuse over white-box reuse, implying that the heart of the framework is formed by loosely associated objects. Moreover, most frameworks adopt the Hollywood principle, i.e. the framework provides the general control flow of the application and client customise parts of it by supplying domain specific objects. Usually, the Hollywood principle assumes that clients supply their classes and that the framework creates the necessary objects. However, the framework itself cannot rely on client specific details and needs abstraction layers to hide direct references to client classes.

That is why most frameworks provide some kind of object factories (note that the generic design pattern catalogue [GammaEtAl'95] devotes an entire chapter to object factories). Object factories provide an abstraction layer that shields the framework from client specific details. Object factories are objects that have the special responsibility to create (and destroy) objects and object structures by means of some kind of abstract description. Object factories often involve a large body of the framework code and require special programming tricks, hence the need for supporting tools [VanLimberghen'96].

In the example of a user-interface framework, the design should be independent of particular subclasses clients provide for the `SensitiveRegion` class and the `Tracker` class. The user-interface framework may solve this by defining a special object that associates mnemonic names with specific classes. On start-up, the framework client installs the necessary classes in the association table and passes the mnemonic names as arguments to the framework operations that requests them. The user-interface framework calls the services of the object factory to create the objects, supplying the mnemonic name as a parameter. The user-interface framework may choose to incorporate a similar association table, to know which tracker objects can be associated with what sensitive regions.

Conclusion

Reuse is a way to reduce development effort. The object-oriented community proposes object-oriented frameworks as a technique to reuse a design for a particular problem domain. A framework is then a set of co-operating classes and objects, together with a set of rules on how they can be combined.

The set of rules is the crucial ingredient of a framework, and two important techniques exist that support the management of these rules. Contracts explicitly specify dynamic object relationships and interactions within a framework. Design patterns describe sets of rules that solve recurring design problems in an abstract, implementation independent way.

Frameworks are often called semi-finished programs in the sense that a framework consists of template methods that call methods without an implementation (the so-called abstract methods defined in abstract classes). Those template methods define the reusable aspects of the design. Clients must complete the framework by providing concrete classes with concrete methods overriding the abstract methods. As frameworks must be independent of the concrete classes provided by clients, most frameworks include some kind of object factories.

There are essentially two techniques for defining template methods, one is by class inheritance and the other is by object composition. Within frameworks, the techniques are usually referred to as white-box reuse and black-box reuse. The general advise is to favour black-box reuse over white-box reuse, because this makes frameworks more extensible and easier maintainable. However, black-box reuse is more difficult to understand and more difficult to build, hence the need for support.

The best support comes from an iterative development process, where a framework is tested in different applications and reworked to improve its design. A technique we found useful during experiments, was the specification of a design space. A design space, captures the varying aspects of a problem domain in an n-dimensional axis system, where each axis represents a fundamental characteristic of all systems in the framework domain that varies depending on the exact system one is modelling.

Meta-object Protocols

If progress in software engineering is about providing higher level abstractions, and if incremental development is about adaptability, then an obvious step is to provide abstraction levels that deal with the adaptation of a software system. That is the intent of an open implementation (often referred to as meta-level abstraction) — an abstraction technique that advances the state of the art in software engineering by providing abstraction levels that deal with the change of the system. Open implementations are a subject of study in the design of object-oriented implementations and there they are called meta-object protocols.

Open Implementation (Meta-Level Abstraction)

a) Black-box Abstraction versus Open Implementations

Software engineering is an engineering discipline and like all engineering disciplines, it is about coping with the complexity of the systems to build. The principal techniques for all engineering disciplines are abstraction and decomposition [Abelson,Sussman'84]. Software engineers have summarised these techniques into the principle of black-box abstraction, i.e. decompose a software system in modules where each module exposes its functionality through its interface but hides the way this functionality is implemented from the other modules.

Open implementation researchers claim that black-box abstraction is not always a good idea, because in many cases, clients want to influence the way the functionality is implemented. That is, if a module exposes only functionality, it hides away implementation issues that are not always details but often crucial strategy issues that bias the performance of the resulting application [Kiczales'94a], [Kiczales'94b].

b) Mapping Dilemmas, Mapping Decisions and Mapping Conflicts

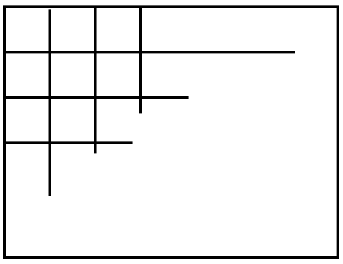
The claim follows from the insight that each black-box maps a higher level abstraction onto a lower level one. Often, such a mapping involves a mapping dilemma, i.e. a choice between several ways of mapping. The implementors of the black-box have to solve this dilemma taking a mapping decision, so the resulting implementation works better for some cases than others. However, particular clients may endure severe performance penalties by adopting the offered implementation and such a situation is called a mapping conflict.

The proposed solution is to open the box in a controlled way and allow the client to control the mapping decision hidden inside the implementation. Hence the name, open implementations.

c) The Spreadsheet Example

A recurring example in publications discussing open implementations (see [Rao'91], [Kiczales'94a], [Kiczales'94b]) is about implementing a spreadsheet on top of a windowing system.

When a windowing system is implemented as a black-box, it exposes functionality like sharing screen space and mouse-tracking and hides implementation details like window data structures and mouse-tracking algorithms. Having access to a functional interface for a windowing system, it is fairly easy to implement a spreadsheet (a rectangular array of non overlapping cells) by modelling each cell with a separate window, like shown in figure 13.



```
FOR i = 1 TO 100
   FOR j = 1 TO 100
      makeWindow (100, 100,
         i * 100, j * 100)
   END
END
```

Figure 13: The Spreadsheet Example of a Mapping Conflict

According to the principles of black-box abstraction this is a perfect solution. The engineer writing the spreadsheet application is not bothered with the underlying implementation details and can build a spreadsheet by a mere call to some `makeWindow` function for each cell in the spreadsheet. Doing this, the spreadsheet implementor would not need to write its own mouse-tracking algorithms and cell data structures.

However, the solution does not work in practice because the engineer writing the spreadsheet program is confronted with a mapping conflict. The mapping conflict is caused by the mapping dilemma involved in implementing a windowing system. Implementors have to choose data structures for the windows and algorithms for mouse-tracking and normally, they favour a solution that works well for a relatively small number of overlapping windows. Such a solution involves large window data structures and dumb mouse-polling algorithms. However, in the case of the spreadsheet the program would consume a huge amount of memory because the program reserves ten thousand memory segments each holding a complete window structure. Also, it would end up to be a very slow spreadsheet, because the mouse-tracking does not exploit the property that all cells are lined up one beside another but perform some sequential search through the list of all opened windows.

[Kiczales'94a] gives other examples (virtual memory, compilers) to sustain the viewpoint that mapping conflicts to some degree appear in all software modules that are reused by clients with different needs. This implies that mapping conflicts are more than just nice theoretical problems and thus that black-box abstraction should be extended to cope with mapping conflicts.

d) Base Level Interface and Meta-Level Interface

To deal with mapping dilemmas, open implementations propose a software engineer to control the mapping decision instead of hacking an entry into the black-box. However, open implementations do not want to loose the advantages of black-box abstraction and recognise the need to separate interface from implementation.

In open implementations, the separation of concerns is achieved by offering two separate interfaces, a base level interface and a meta-level interface. The base level interface is the functional interface that results from a traditional black-box design and exposes functionality only. The meta-level interface exposes implementation issues and is used to control the mapping decisions.

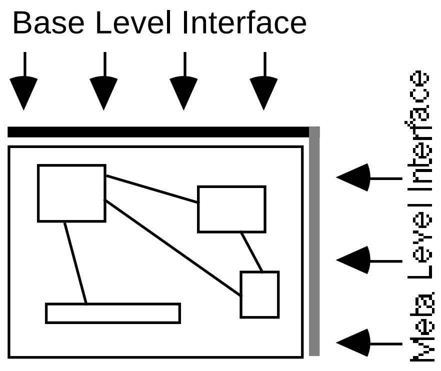


Figure 14: Base Level Interface & Meta-Level Interface

e) Designing a Meta-Level Interface

A well-designed meta-level interface is supposed to expose critical implementation issues and hide insignificant implementation details. Consider the example of a database system, where several techniques (i.e. hashing, B-trees, …) exist to tune the performance of the system. A software engineer tuning a database system is not forced to implement its own hashing table or a B-tree, but instead can fill-in some parameters to optimise the behaviour of the database to the needs of the application.

However, the question how to design meta-level interfaces for software systems is hard and remains open [Kiczales'94b]. The current thinking is that designing a meta-level interface is comparable in effort to the design of a base level interface and that similar techniques can be applied there [McAffer'95]. So abstraction and decomposition are necessary in the design of meta-level interfaces as well, hence iteration is crucial to filter out the significant issues that should be accessible through the meta-level interface. *However —and this is an essential problem of open implementations— iteration over the meta-level interface is more difficult and demands more resources than iteration over base level interfaces.*

To iterate over the meta-level interface, one must find a new application that requires a similar abstraction (i.e. base level interface) yet a different mapping decision (i.e. meta-level interface). If an application requires a different mapping decision, it means that the application tackles a different problem, thus that the application resides in a different application domain. To build applications for new application domains one must acquire new domain knowledge, a costly process as it involves new people, extra time and more money. Hence the conclusion that the design of meta-object protocols must be supported by theory and guidelines.

The contribution of this dissertation is a partial answer to the question how to design a meta-level interface (see the Zypher contribution - p.63). Before we can present the answer we need a little bit more background on the intuitive notion of a meta-level interface. We need to answer the question "what is meta ?", a question that is studied within the reflection community.

What is Meta ?

To summarise the previous section, an open implementation is an abstraction technique that proposes a scheme which allows client modules to negotiate with their service modules to find an optimum internal representation. Observing this scheme with a little more detail, reveals that a program with an open implementation somehow analyses its working and adapts its internal representation accordingly. This ability of self-analysis and self-adaptability is called reflection and has been studied within computer science for quite a few years now. With the advent of open implementations, reflection has found its way into software design.

An open implementation proclaims that a system should have two interfaces, a base level interface and a meta-level interface. To support the design of meta-level interfaces, we need an answer to the question "what is meta ?". The only problem is that there is no clear-cut answer to the question: reflection is an active field of research and there are exist many possible answers in the community.

Below we give an overview of two of the views that exist on the notion of reflection. And since there is no answer to the question "what is meta ?", we look for a criterion that helps us to decide "when is something meta ?".

a) Computational Reflection or Meta-Linguistic Abstraction

The term computational reflection stems from [Maes'87], while meta-linguistic abstraction comes from [Abelson,Sussman'84].

Computational reflection is a particular way to look at a software system. The view starts from the assumption that there exists a programming language used to express problems on a certain abstraction level. Expressions in that language are called programs and programs are turned into working systems (called computational systems) by means of an evaluator. An evaluator (called the meta-system) is a computational system that executes the program, i.e. it performs the actions required to realise the meaning of that program. The program, the computational system and the meta-system are depicted in the left hand side of figure 14.

Computational reflection is then the ability of a computational system to "*inspect and/or manipulate the representations of the computational process specified by a system's program*" [Rao'91] and is depicted by the grey arrow in the right hand side of figure 14.

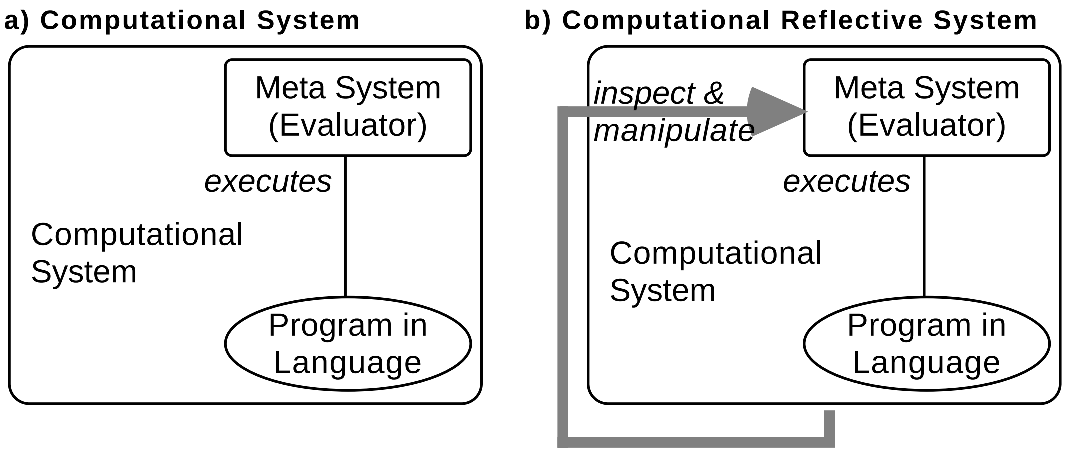


Figure 15: Computational Reflection

A well-known experiment applying computational reflection on program language design was the CLOS meta-object protocol [Kiczales,Rivières,Bobrow'91]. The experiment showed that reflective programming languages provide an extra abstraction layer, which eases extensibility, compatibility and efficiency. Reflective programming languages ease extensibility because it is possible to define a small and fixed kernel language and use that kernel to extend the language expressiveness; they ease compatibility because it is possible to ensure that earlier versions of the language lie within the scope of meta-extensions; they ease efficiency because one can differ the implementation strategy depending on the problem to program is solving. Moreover, programming environments for reflective programming languages may use the reflection abilities to monitor the evaluation of programs and construct powerful tools like debuggers and code optimisers more comfortably.

Computational reflection is a valuable view to look at reflection in programming languages. In [Rao'91], an attempt is made to adapt the computational reflection view to fit open implementations. This view is called implementational reflection.

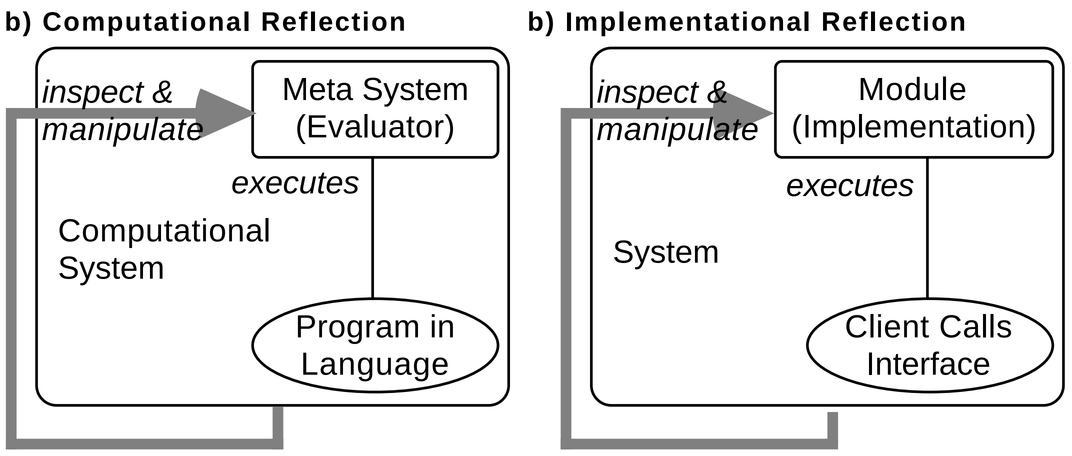


Figure 16: Implementational Reflection

The implementational reflection view is based on a one-to one mapping from programming language concepts to system concepts depicted in figure 16: a computational system is mapped on a generic system; the programming language is mapped on an interface of a module in the system; the evaluator is mapped on the internal implementation of that module and the program itself corresponds with system calls to the interface of that module. Implementational reflection is then the ability of a system to "*inspect and/or manipulate the implementational structures of the modules used by that system*" [Rao'91]. Seen this way, computational and implementational reflection are just different characterisations of the same essential capability, namely some form of self-analysis.

However, this mapping is not evident in systems that are not about programming languages. This follows from the fact that two co-operating subsystems use each others interface to communicate, so each of the two can be regarded as a meta-system for the other. Consider the example of a relational database system. Client systems send SQL queries to the database system, which evaluates these statements to produce desired data records. So in this case, the database system plays the role of a meta-system for the client system. On the other hand, client systems receive a stream of data records and must interpret that stream to identify the data records. So client systems play the role of a meta-system for the database.

We conclude that the implementational reflection is not able to tell which subsystem is the base level and which is the meta-level. However, this separation is essential for designing a meta-level interface (see p.57), and so the meta-linguistic view is not appropriate when discussing open implementations.

b) Definition of Reflection

Since the meta-linguistic view for reflection is not appropriate for open implementations, we have to search for another one. [Maes'87] proposes a definition for reflection that is applicable to any model of computation: "*A reflective system is a system which is about itself in a causally connected way*".

The three main ideas in this definition (i.e. system, about-ness and causal connection) need a little explanation to make things more precise. A 'system' is software running on a computer with the intention to answer questions about and/or support actions in some domain. A system incorporates internal structures representing its domain, that is why a system is said to be 'about' its domain. A system is said to be 'causally connected' to its domain if the internal structures and the domain they represent are linked in such a way that if one of them changes, this leads to a corresponding effect on the other.

[Maes'87] includes an example of a system steering a robot-arm to illustrate the definition. Such a system incorporates internal structures holding data values, relations and algorithms to represent the position of a robot-arm, so the system is 'about' the robot-arm position. These system structures are 'causally connected' to a particular robot-arm if (a) movements of the robot-arm originated by some external force changes the internal structures accordingly and (b) changes to the internal structure (by some computation) make the robot-arm move to the corresponding position.

When facing a software system with several subsystems, a rigorous application of this definition allows us to see what subsystems belong to the meta-level. A subsystem is part of the meta-level if and only if, that subsystem is (a) about the software system in (b) a causally connected way. If we reconsider the relational database example, we see that neither the database system nor the client system are meta-systems because they are not about the system but about the domain of the database applications. However, the parts of the database and client systems that exchange commands to create database structures (the DDL statements) form together the meta-system for the parts that exchange commands to query and update database structures (the so-called DML statements).

c) The Meta-Object Criterion

The last example makes clear that this definition of reflection is applicable as a criterion to judge whether a particular subsystem belongs to the meta-level, so is more appropriate in the study of meta-level architectures that are not about programming languages. So it is the latter view on reflection that is adopted for the remainder of this thesis.

When meta-level interfaces are studied within the context of an object-oriented implementation they are referred to as a meta-object protocol. Most open implementationsare based on an object-oriented implementation, hence the reason why the terms meta-object protocol and open implementation are often used interchangeably. We rephrase our meta-level criterion in terms of objects.

Meta-Object Criterion

An object is part of the meta-level if and only if, that object is (a) about the software system in (b) a causally connected way.

In this criterion, 'being about X' means incorporating internal structures representing X and 'causally connected' means that if one of the sides of the connection changes the other side changes correspondingly.

c) Systems Meta-Object Protocols

The CLOS meta-object protocol [Kiczales,Rivières,Bobrow'91] is the most prominent example of an open implementation for an object-oriented language. Among others, this meta-object protocol has been used to make CLOS persistent [Paepcke'90].

Agora [SteyaertEtAl'93], [Steyaert'94], [CodenieEtAl'94], [Steyaert,DeMeuter'95] is a reflective, prototype-based language that features a general mixin-based approach to (multiple) inheritance. Agora uses its meta-object protocol to explore different object-oriented programming paradigms within a design-space of object-oriented programming paradigms.

CodA [McAffer'95] is especially important as it shows that a meta-level is "just another application" and that one can apply traditional software design techniques like abstraction and decomposition on the design of meta-level interfaces as well. CodA has been used to open up the implementation of Smalltalk message passing and is able to add meta-level infrastructure to Smalltalk objects so that additional behaviour like concurrency or distribution can be added when necessary.

Abstract Communication Types [AksitEtAl'93] is similar in aim but different in approach than CodA. Abstract communication types are objects that abstract interactions among objects and can be used to model distribution and concurrency. Unlike CodA, the abstract communication types are intended to be part of the design of a new programming language (i.e. Sina).

Apertos [Yokote,Teraoka,Tokoro'89], [Yokote'92] is an example of a meta-object protocol used in an object-oriented operating system for mobile computing. Because portable and mobile computers change locations frequently, the operating system must evolve and adapt itself to its execution environment. Apertos supports distribution and object migration.

SOM, the System Object Model part of IBM's OS/2 operating system [Forman,Danforth,Madduri'95] includes the notion of meta-classes, which can be used to wrap additional behaviour around base level message sends. Among others this can be useful in tackling persistency and concurrency.

Silica [Rao'91] is an open implementation for a windowing system. By providing special meta-objects, clients of the windowing system can fine-tune the Silica windowing system to their needs. In [Rao'91] it is described how Silica would handle the spreadsheet example described earlier.

ApplFLab [SteyaertEtAl'94], [SteyaertEtAl'96] is an experiment that investigates what is needed to make a user interface builder incrementally refinable. ApplFLab starts from the assumption that user interface builders are essential tools for the development of modern applications, and argues that the state of the art of the field lacks a way of incorporating new user interface paradigms (e.g. direct manipulation, menu driven, navigational). ApplFLab is an object-oriented framework for a user interface builder and uses a meta-object protocol to install new interface paradigms.

Conclusion

To cope with complexity, software engineers apply a divide and conquer technique based on abstraction and decomposition. A system is decomposed in different modules, where each module exposes some functionality through an abstract interface, this way hiding its internal implementation. Each module maps high level functionality onto lower level functionality and this mapping restricts the possibilities of the higher level functionality.

An open implementation is a particular abstraction technique where the interface of a module is split in two separate parts: the base level interface and the meta-level interface. The base level interface is used to access the basic functionality of the module, while the meta-level interface allows to control the mapping imposed by the module. Experiments have shown that the open implementation abstraction technique makes systems more adaptable.

When meta-level interfaces are studied within the context of an object-oriented implementation they are referred to as a meta-object protocol. The separation of concerns principle dictates that it is better to have a clear separation between the base level interface and the meta-level interface. This implies that one needs a criterion to recognise meta-level objects in existing software systems. We adopted a criterion stating that an object is part of the meta-level if and only if, that object is about the software system in a causally connected way. In this criterion, 'being about X' means incorporating internal structures representing X and 'causally connected' means that if one of the sides of the connection changes the other side changes correspondingly.

The Zypher Contribution

Within today's software and hardware industry, there is a growing tendency towards openness. This tendency can be observed in areas like operating systems (e.g. UNIX and OS/2), databases (e.g. Exodus and CORBA), inter application communication (e.g. OLE and OpenDoc), tailorable software (e.g. Emacs and AutoCAD) and programming languages (Smalltalk and CLOS). More recently, the Taligent company [Cotter,Potel'95] —the joint venture between Apple, IBM and Hewlett-Packard— emerged as one of the most prominent advocates for openness, developing an open software platform that is able to keep pace with the growing demands and emerging technological innovations of the software market. Open hypermedia (p.22), is another example of a field where openness is considered a valuable, and —as argued earlier— the field is an excellent representative for the larger collection of open systems.

The software engineering research community has contributed to this open systems tendency by means of object-oriented frameworks and meta-object protocols discussed in the previous sections. Those techniques have found their way into the design of commercially available open systems: Taligent [Cotter,Potel'95] already identified frameworks as technological cornerstones for their CommonPoint open platform and IBM incorporated meta-objects in the System Object Model (SOM) underlying OS/2 [Forman,Danforth,Madduri'95]. Nevertheless, the today application of both techniques is more an art than a science and certainly the total effect of combining both techniques remains an open question.

We have performed the Zypher experiment to investigate the precise effect of these techniques by applying them on the design and implementation of an open hypermedia system. That is, we want to investigate why it is a good idea to use an object-oriented framework and a meta-object protocol in the design of an open hypermedia system and to what extent the two techniques overlap or differ. Moreover, it has been our explicit intention to be able to generalise our results to the design of open systems in other domains as well.

Three levels of tailorability: a Recapitulation

In the domain of software engineering, it is common practice to validate techniques experimentally [NationalAcademy'94] — researchers build prototype systems for a particular application domain to investigate benefits and drawbacks of a technique. Such a prototype system is called a software artefact and the Zypher open hypermedia framework is the software artefact we have been using for the experimental validation.

The design of the Zypher open hypermedia framework is based on the Zypher design space and the notion of tailorability as defined in the section entitled "The Zypher Perspective" (p.30). To develop our argumentation, we recapitulate the main ideas in what follows.

Open Hypermedia Working Definition

The Zypher design space is a three-dimensional (storage, presentation and navigation) axis system. Each dimension represents a fundamental characteristic of all hypermedia systems that varies depending on the exact system one is modelling. Each point on a design space axis is a particular variant of the fundamental characteristic.

A hypermedia system is a relation (in the mathematical sense of the word) between points on the three axes of the Zypher design space. An open hypermedia system is a "tailorable" hypermedia system.

Figure 17 represents a hypermedia system by a 3-dimensional volume in the Zypher design space; for the sake of simplicity we represent such a volume as a cube. An open hypermedia system is represented by a tailorable volume.

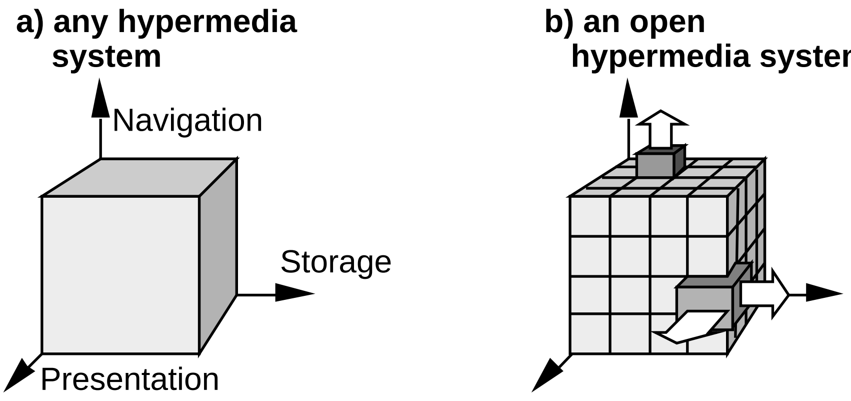


Figure 17: Zypher Design Space with an Open Hypermedia System

Our working definition proposes tailorability as the characteristic property of an open hypermedia system. We define three levels of tailorability based on three possibilities to tailor a volume in the open hypermedia design space.

Domain Level Tailorability

Domain level tailorability is the kind of tailorability needed to deliver a hypermedia system for a specific application domain. Typical usage of domain level tailorability in hypermedia systems would involve the incorporation of new data formats, extra information presentation methods or supplementary types of navigation techniques. *Domain level tailorability corresponds with the addition of extra points (or the modification of existing points) on the axes of the Zypher design space.*

System Level Tailorability

System level tailorability aims to deliver services that affect the global behaviour of the hypermedia system. Examples are concurrency control (i.e. manage concurrent access to shared data), logging (i.e. maintaining a log of navigation activities to provide backtracking features), caching (predict future behaviour on the basis of registered activities), authority control (check whether the user of the system has the privileges to see or modify information) and integrity control (control operations to preserve the consistency of the system's data structures). All these examples have in common that they change the behaviour of several modules in the system in a uniform way. *System level tailorability corresponds with wrapping uniform behaviour around different points residing on an axis of the Zypher design space.*

Configuration Level Tailorability

Configuration level tailorability aims to provide a 'plug and play' hypermedia system, where the coordination between the system modules is adapted without changing their internal implementation. Typical usage of configuration level tailorability is to make the system run-time extensible. For instance, the table of helper applications in most world-wide web browsers, where users can associate a viewer application with a given document type is about the configuration between the storage and presentation axis. The Java approach [Java'95] is another example, where world-wide web browsers load system modules for handling unknown document formats. The final example is protocol negotiation, where system modules communicate to choose the optimal protocol for exchanging data; protocol negotiation is valuable in cross-platform communication. *Configuration level tailorability corresponds with changing the relationship (in the mathematical sense of the word) between the points on the axes in the Zypher design space.*

Note the important property that *the definition of the three levels of tailorability is independent of the type and numbers of axes in the design space*.

Tailorability and Object-oriented Software Engineering

Now that we have proposed three levels of tailorability as a distinctive characteristic of an open hypermedia system, it is time to reconsider the question how object-oriented frameworks and meta-object protocols can be applied in the design of an open hypermedia system. To answer this question, we look for a mapping between the three levels of tailorability and the two techniques provided by the object-oriented software engineering community. If we could find such a mapping, then we may conclude that the two techniques overlap since they both share the common goal of tailorability, yet differ in the exact level of tailorability that can be achieved.

We start with an intuitive mapping and elaborate on our ideas later.

Framework Level Domain Level Tailorability

A framework provides a skeleton application for a particular domain and clients must supply concrete methods for the "hot spots" of the framework (i.e. iterative development - p.51). If we would design our framework so that each axis in the Zypher design space corresponds with a black-box template method (i.e. white-box & black-box reuse - p.48), then adding points to the design space axis corresponds with supplying concrete implementations for the abstract method in the template. So, a framework seems like a dedicated means to provide domain level configurability.

Meta-Level System Level Tailorability

The crucial idea of system level tailorability is to adapt the global behaviour of the system in order to offer additional services. Locking is an example of such an additional service and to implement a successful locking strategy we must change all data-access operations in a uniform way: we cannot risk to miss a single read or write operation since this may violate data integrity.

We can accomplish locking behaviour by changing the implementation of all the modules in the system in a uniform way. This is not a very appealing solution however. First of all, we have to make sure that all modules that do call or implement a method that reads or writes a data-value are changed, but there are no tools to detect them. Secondly, changing the internal implementation of the modules implies that we must understand all internal details of each of these modules. This is hard, since read and write operations may be hidden deep inside the code. Finally, it is not always possible to change the internal implementation of a module, as it may be supplied by an external vendor.

Locking is a particular example, but the same is true for the other kind of services that depend on changing all operations of a certain kind. For instance, maintaining a trail of all visited locations implies that the system must log all navigation operations in a uniform way; to implement a cache, the system must trap all storage operations in a uniform way; authority control requires the system to trap all presentation operations in a uniform way; integrity control requires the system to trap all storage, navigation and presentation operations in a uniform way.

So to accomplish system level tailorability, we need a way to trap all operations of a certain kind, independent of the module that calls or implements that operation and wrap uniform behaviour around the operation. In other words, our hypermedia system must have a special interface that allows us to wrap extra code around these kind of operations. A meta-level interface seems like a dedicated means to provide this kind of functionality.

Meta-Level Configuration Level Tailorability

Finally, configuration tailorability is about changing the configuration of the system's modules, without changing their internal implementation.

If we review the examples that come with configuration level tailorability, we see that the problem is in fact very similar to the system level tailorability problem: we need a way to change all decisions about the system's configuration in a uniform way. The minor difference is that for system level tailorability we must wrap additional behaviour, while configuration level tailorability requires us to change the behaviour. For instance, the table of helper applications is a decision table that tells which presentation module should handle the output of what storage module. The Java approach can be handled by checking the above decision table, and requesting the server to supply a presentation module if we do not find an entry. Protocol negotiation is an example on deciding which two modules should handle a certain service.

So to accomplish configuration level tailorability, we need a way to change all operations in the system that determine the system configuration in a uniform way. In other words, our hypermedia system must have a special interface that allows us to plug in special configuration code. A meta-level interface seems like a dedicated means to provide this functionality. Note that such a meta-level interface corresponds to the idea of an object factory (p.54).

Explicit Framework Contracts are Meta-Objects

So, it seems that a meta-object protocol is applicable to achieve both system level tailorability and configuration level tailorability. Our intuition has been guided by the insight that both levels of functionality could be incorporated by having two dedicated interfaces: one that allows us to wrap additional behaviour around important operations and another one that allows us to plug configuration code. Nevertheless, we have not motivated why such a dedicated interface is part of the meta-level.

In the section on meta-object protocols, we have identified the meta-object criterion (p.61) that enables us to decide whether an object is part of the meta-level. So if we know what objects are included in the dedicated interfaces, we should be able to tell whether they belong to the meta-level interface or not.

The encompassing property of the dedicated interfaces for system and configuration level tailorability, is that they capture the important design aspects of the framework. Indeed, the operations we want to log, cache, lock, control with system level tailorability are the operations that perform critical system tasks, so are important aspects of the framework design. Also, the operations that determine the system configuration are crucial for the behaviour of the framework, thus an important aspect of the system design. Important design aspects of a framework are normally specified using framework contracts, so we may conclude that objects part of the dedicated interfaces are explicit representations of framework contracts.

Using the meta-object criterion, we can show that *an object which is an explicit representation of a framework contract is part of the meta-level.* The meta-object criterion for an object to be part of the meta-level; that object should be (a) about the software system in (b) a causally connected way. Point (a) follows from the fact that an object which is an explicit representation of a framework contract is by definition about the system and not about the system's domain. To satisfy the causal connection requirement (b), we must show that if one of the sides of the connection changes the other side changes correspondingly. One direction follows immediately — changing the implementation of the contract object changes the behaviour of the system. Moreover, since the object represents a framework contract, the only legal way to change that part of the system behaviour is to change the corresponding contract. For if one changes the framework implementation by violating the framework contract, one forsakes the safety net captured in the framework design principles. So the other direction is satisfied as well, thus the causal connection requirement is satisfied.

The insight that an explicit representation of a framework contract is part of the meta-level, is vitally important as it leads to the conclusion that *the design of a framework provides an initial ground for the design of a meta-object protocol.* As a consequence we can be confident that the design of the meta-level interface can reuse parts of the design of the framework's base level interface. Knowing that the design of the meta-level is more difficult since iteration is more expensive, and knowing that there are lots of promising techniques for designing base level interfaces are under way, this is an extremely valuable perception in its own right.

Framework Design Guidelines

Knowing that explicit representations of framework contracts are meta-objects, we must say something about what contracts should be made explicit in the design. Obviously, we need some support to tell what contracts are pertinent, as a typical framework comes with numerous contracts. As argued below, the Zypher design space provides an ideal representation of the contracts that should be made explicit.

First, we propose a framework design methodology summarised in four design guidelines.

Framework Design Guidelines

1) Devise a design space for the problem domain.
2) Each design space axis should correspond to a black-box template method.
3) Each design space axis should correspond to a framework contract.
4) The configuration of the design space axes should correspond to a framework contract.

Second, we show that a system designer who respects these guidelines can build a system incorporating the three levels of tailorability.

1) The first guideline constrains our methodology to problem domains where it is possible to devise a design space. The Agora [Steyaert'94] and ApplFLab [SteyaertEtAl'94], [SteyaertEtAl'96] experiments indicate that this is at least feasible.
2) The second design guideline ensures that a framework provides domain level tailorability. A software engineer who supplies a concrete implementation for the abstract methods called by the template method adds a point to the corresponding design space axis.
3) A framework that respects the third guideline, can be made system level tailorable by extending the design with meta-objects that are explicit representations of the framework contracts that follow from the guideline. By supplying a different contract object, one can provide extra behaviour for the execution of the contract, this way wrapping uniform behaviour around different points residing on the corresponding design space axis.
4) A framework that respects the fourth guideline, can be made configuration level tailorable by extending the design with a meta-object that is an explicit representation of the framework configuration contract. By supplying a different configuration object, one can change the configuration behaviour of the framework, this way changing the relationship between the axes of the design space.

Note that points 2, 3 and 4 imply the following implications

Object-Oriented Framework	Domain Level Tailorability
Meta-Object Protocol	System Level Tailorability
Meta-Object Protocol	Configuration Level Tailorability

These implications provide the answer to the question how both object-oriented frameworks and meta-object protocols can be combined properly. In some sense, object-oriented frameworks and meta-object protocols are overlapping techniques since they both share the common goal of tailorability. Yet the techniques differ in the exact level of tailorability that can be achieved.

The final remark is about generalising the results for other kinds of open systems. Note that the domain dependent features are captured in the type and number of design space axes but that the framework design guidelines and the three levels of tailorability are completely independent of the type and number of design space axes. This leads to the conclusion that *the proposed framework design methodology can be applied on the construction of open systems in other domains than the hypermedia domain.* The constraining factor is the ability to construct a design space for the problem domain at hand.

Conclusion

We have provided an answer to the question how object-oriented software engineering can contribute to the development of open hypermedia systems. More precisely, we have argued how object-oriented frameworks and meta-object protocols can help to develop a hypermedia system with three levels of tailorability — domain level, system level and configuration level. By adopting those three levels of tailorability as distinctive characteristics of open hypermedia systems, we have concluded that object-oriented frameworks and meta-object protocols overlap, since they both share the common goal of tailorability, yet differ, since they achieve another level of tailorability.

The main point in our argumentation was the insight that *explicit representations of framework contracts are meta-objects*. Based on this insight, we have formulated four framework design guidelines and we have proposed a framework design methodology to incorporate the three levels of tailorability.

Moreover, we have argued that the same approach can be applied to open systems outside the hypermedia domain, because all our arguments are independent of the hypermedia domain. So if someone is able to devise a design space for a particular domain (that is, an n-dimensional axis system) and if that design space is turned into a framework that respects the framework design guidelines, then it is possible to incorporate the three levels of tailorability.