Automatic Presentation of Model Data in MVC++ Applications

Markku Vuorenmaa

University of Tampere
Department of Computer and
Information Sciences
Master's thesis
April 2000

University of Tampere
Department of Computer and Information Sciences
Markku Vuorenmaa: Automatic Presentation of Model Data in MVC++ Applications
Master's thesis, 63 pages

April 2000

Although Model-View-Controller architecture and MVC++ support building object-oriented and modular applications, designing and implementing the model data handling in view and controller layers requires a lot of work. If an abstraction about the generic behaviour of the model layer can be constructed, an object-oriented framework could be designed to provide automatic handling and presentation of model layer data.

This thesis presents a design of such a framework as well as some early usage experiences. The presented framework succeeds in supporting presentation and modifications of already existing data, but can not offer good support for object creation and removal. Experimenting with test programs and prototypes has been encouraging, but has also revealed some typical problems of framework design, implementation and usage.

As a conclusion, one can say that a framework reaching beyond MVC++ functionality can provide partly automatic data handling for MVC++ applications. The presented Model Presentation Framework undoubtedly needs more work to reach the maturity level found in more advanced frameworks.

Keywords: frameworks, object-oriented design, design patterns, Model-View-Controller architecture, MVC++

Acknowledgements

The colleagues in my department at Nokia have greatly encouraged and supported me during this work. They deserve my full gratitude. Especially I would like to mention former department manager Juha Tuominen, who approved the faint idea and believed in it in the beginning. My group manager and thesis instructor, Heli Viitanen, has given me good advice and new viewpoints throughout the process. Peter Burgess has kept my spirits up, and supported my views. He was also kind enough to correct the language.

At the University of Tampere, my supervisor Roope Raisamo has been very helpful, patient and understanding. He has been able to bring balance to the content and to guide me towards new directions. My second supervisor, Tarja Systä, has been very flexible and thanks to her I was able to keep my schedule in the end.

Finally, I wish to thank Merja for love and support she has given me during my studies.

Contents

1. Introduction

The modern world relies more and more on software [Jacobson *et al.*, 1997]. Many daily activities have moved or are moving into computers or computer networks and software is embedded into various devices that people use [Jacobson *et al.*, 1997]. Customer requirements for new software products get more demanding all the time. On the other hand software is used in critical areas, where the quality has the highest priority [Pree, 1995]. The competitive market causes pressure to produce software faster, cheaper and better [Jacobson *et al.*, 1997]. That can be done by improving the efficiency of software production and increasing the amount of reuse [Jacobson *et al.*, 1997]. Object-oriented design and programming solve some problems of efficiency and reuse [Lewis *et al.*, 1995], but achieving best results requires planned and purposeful adoption of reuse in software development processes [Jacobson *et al.*, 1997].

Frameworks are abstract subsystems that can be specialised to produce new software [Jacobson *et al.*, 1997]. Model-View-Controller (MVC) architecture can be seen as a low-level application framework [Jacobson *et al.*, 1997]. MVC++ is an implementation of the MVC architecture for C++ and Java [Bonnet, 1999]. MVC++ applications have many common characteristics. Model, view and controller parts are clearly separated in design and implementation phases. Although the common conventions help in doing the work better and faster, there still are many repeating tasks in building the user interfaces for real-life applications. A gap between a small MVC++ example and an actual complex user interface application leads to huge increase of work effort in practise. A lot of this work is put to making a user interface look and behave as designed. One common and heavy work phase is visualising the data stored in the model layer of the application. The data must be exchanged between view and model layers in some form, and the task is not always trivial. Displaying the data in complex user interface components often requires implementation of a new controller and even model functionality into view layer of the application. This means that we actually implement nested MVC structures inside the view layer.

This thesis examines the MVC architecture, object-oriented frameworks and the issues involved in building frameworks. As an example I use the design and implementation of the Model Presentation Framework, which is an extension to the MVC++ architecture and addresses the problems found in bringing the application data to the user interface layer. The framework presented here provides automatic mapping of data from the model part of an MVC++ application to view components in the user interface and thus offers one solution to the problem. I will discuss the usual strengths and weaknesses of application framework development and compare those with the findings from the development process of the Model Presentation Framework. ET++ and HotDraw frameworks are introduced and small comparison between them and the Model Presentation Framework is presented.

Chapter 2 introduces the Model-View-Controller architecture and also some shortcomings found in it. Chapter 3 covers the nature of frameworks in the object-oriented programming and chapter presents the building process of object oriented framework. Chapter 5 specifies the design of the Model Presentation Framework and chapter 6 gives some examples to demonstrate how the Model Presentation Framework would be used in the development of the MVC++ application. The experiences of using the Model Presentation Framework and more advanced frameworks, ET++ and Hotdraw are discussed in chapter 7. Chapter 8 presents a conclusion on the contents and the results of the thesis.

2. Model-View-Controller Architecture

Methods and guidelines in ever more complex software development have become extremely important. Object-oriented design and programming encourage reuse by inheritance and polymorphism, but still the reusability can only be reached by a careful design [Johnson and Foote, 1988]. Nowadays, as the sizes of software products grow larger and larger, the architecture, design and implementation have to be well organised, clear and maintainable. On the organisational level this means that all developers should use similar working processes in different projects. When people switch from one project to another they should already know the basic principles and be able to get productive faster. This is an important factor as the goal is to reduce the time-to-market. These needs are the motivation for creating predefined software architectures, design methods and conventions. Model-View-Controller is one example of such application architecture.

This chapter introduces first the original Model-View-Controller (MVC) architecture and moves on then to the MVC++. The roles of all parts, model, view and controller are explained. In relation to MVC++, the chapter introduces the concepts of abstract partners and observers that increase modularity and reusability. In the end, advantages and some drawbacks of the MVC++ architecture are discussed.

2.1. Principles of Model-View-Controller Architecture

This section covers the Model-View-Controller architecture and its C++ implementation called MVC++. The purpose of the layers in the Model-View-Controller architecture is presented and loose coupling between the MVC++ layers is dealt with.

2.1.1. Parts of Model-View-Controller Architecture

The idea of Model-View-Controller architecture is to divide application into manageable parts, which have their own responsibilities [Jaaksi, 1994]. Model-View-Controller architecture was originally designed for Smalltalk-80 applications in Xerox Palo Alto Research Center [Krasner and Pope, 1988]. Figure 2-1 illustrates the original Model-View-Controller design.

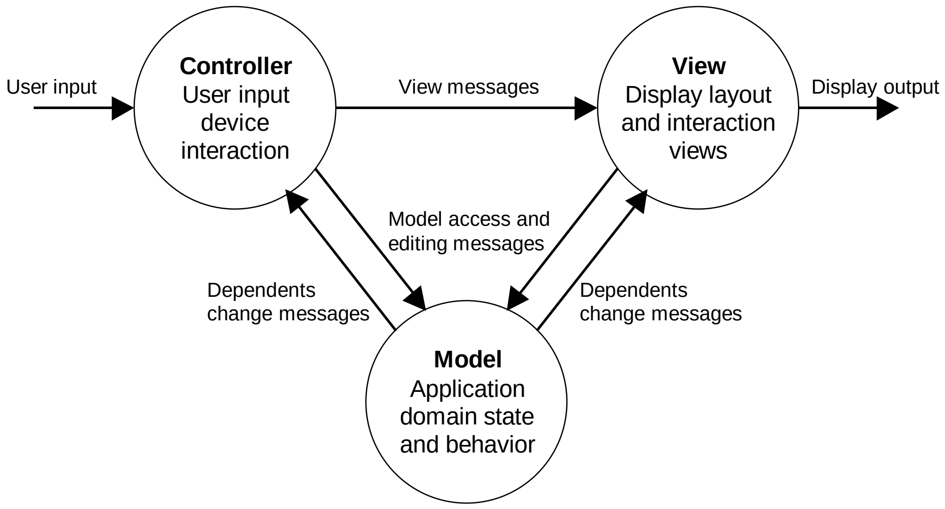

User input
Controller
User input device interaction
View messages
View
Display layout and interaction views
Display output
Model access and editing messages
Dependents change messages
Dependents change messages
Model
Application domain state and behavior

Figure 2-1. Model-View-Controller Architecture [Krasner and Pope, 1988].

Model-View-Controller architecture defines the structure and abstract base of classes for Smalltalk-80 applications. As similar guidelines for designing and implementing X/Motif software using C++ did not exist at Nokia Telecommunications, the Model-View-Controller architecture was adapted to suit development in C++ and OMT++ design process [Jaaksi, 1995]. This modified MVC has been named MVC++ and has successfully been applied in building a family of network management system products in Nokia [Jaaksi, 1995]. The MVC++ has the same three functional layers as the original Model-View-Controller architecture, but their interaction is a bit different.

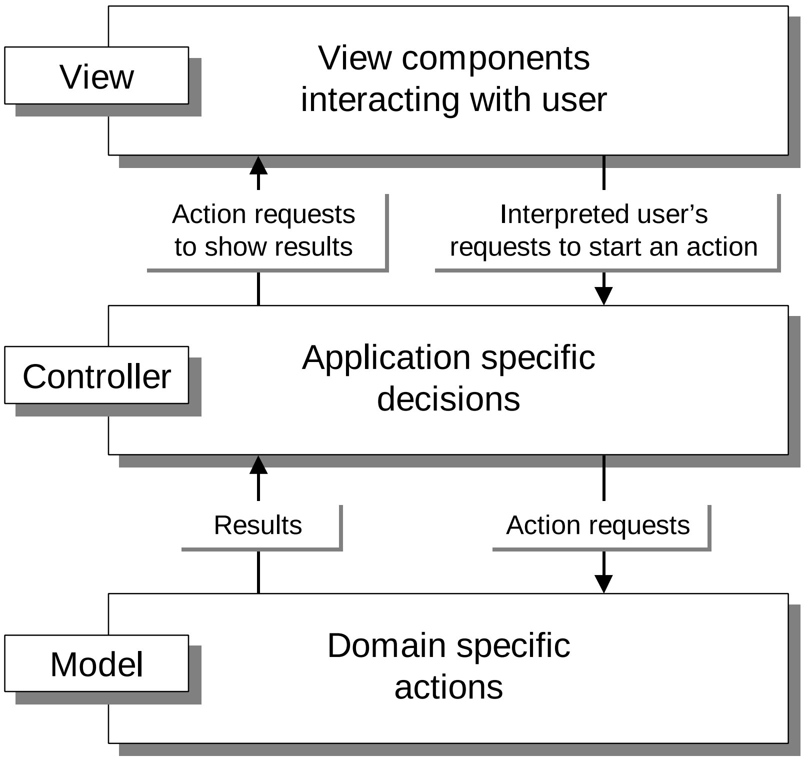

View
View components interacting with user
Action requests to show results
Interpreted user's requests to start an action
Controller
Application specific decisions
Results
Action requests
Model
Domain specific actions

Figure 2-2. Parts of an MVC++ application [Jaaksi *et al.*, 1999].

Shortly, the tasks of the different layers in Figure 2-2 are the following [Jaaksi *et al.*, 1999]:

- The *model* layer defines the classes that represent the concepts of the problem domain.
- The *view* layer forms the user interface.
- The *controller* layer is glue between model and view layers. It handles the interaction between model and view and therefore contains a lot of the logic of the application.

In MVC++ there is no direct connection between the view and the model. [Jaaksi, 1995]. The user interacts with views and they pass the information to their controllers. At this point the controller makes the application specific decisions and operates according to them. In the original Model-View-Controller architecture the view does not receive user input as the controller makes the decisions and delegates the needed actions to view and model layers [Krasner and Pope, 1988].

In more detail, the model part contains all application domain specific data and knows how to manipulate it [Jaaksi, 1994]. The model does not have any kind of user interface and it does not define any application specific logic. The model should be able to perform the data related tasks independently, without knowing anything about the controller and the view layers [Jaaksi, 1994]. On many occasions the model can be completely separate of the rest of the application as it may have only a static role of doing the operations when requested [Jaaksi, 1995]. As an exception, some models may be acting independently. For example, they may perform real-time monitoring or receive events from an external source. In such cases the model layer is able to invoke operations by calling the controller layer without any user interaction [Jaaksi, 1995].

According to Jaaksi [1994], the view shows the state of the model to the user, displays the user interface, and manages all user interactions. The view can also contain reusable view components that may have a simple view-only implementation or they can also define their own internal MVC++ structure [Jaaksi, 1995]. The view component can have methods for manipulation, feedback and querying the state of the view [Jaaksi, 1995].

The controller manages the interaction between the model and the view parts. Whereas model defines the logic of the real world, it is a job of a controller to define the actual application logic [Jaaksi, 1995]. The controller knows the tasks that the model and view can perform, and delegates the application tasks to them. The controller does not need to have detailed knowledge of how the model and the view actually handle the delegated operations.

2.1.2. Loose Coupling with Dynamic Binding

Jaaksi [1994] introduces abstract partners that are not a part of the original MVC architecture. They are a way to reach loose coupling between objects and increase reusability. The view component defines an interface towards the controller by declaring an abstract partner class that controller must inherit and so it is possible to build reusable views as components. Those views can be used in any application and the only requirement is that the new controllers (the partners) implement the abstract methods defined by abstract view partner class. The similar approach between different controllers makes it possible to replace old controllers with new implementations that conform to the old interface [Jaaksi, 1994]. Abstract controller partner between controller and model may also be needed. If the model for instance is monitoring some dynamic system, it may need to pass the notification about the changes to the controller part [Jaaksi, 1995].

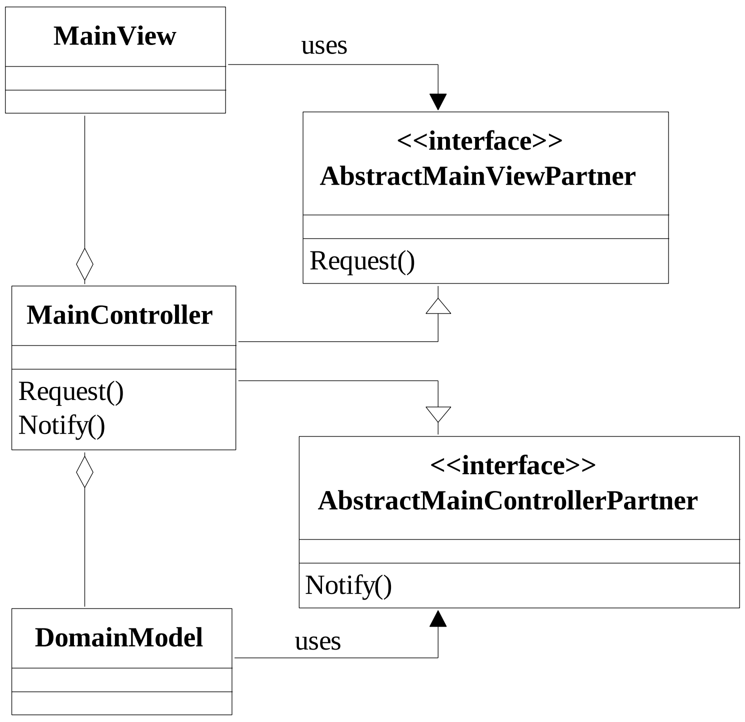


Figure 2-3. Abstract partner relationships between MVC++ layers.

Figure 2-3 above shows a simple object model illustrating typical abstract partner usage. *MainView* class is able to ask information by calling *Request()* method defined in *AbstractMainViewPartner* interface and *DomainModel* class is able to notify *MainController* about changes in the model by calling *Notify()* method defined in *AbstractMainControllerPartner* interface. The valid

MainController (in this case) must implement the methods defined in abstract partners as it inherits both of them.

When an MVC++ application is started, the main controller has a special role when compared to other controllers. The main controller is responsible for creating a model and the view for the application [Jaaksi, 1995]. It must also create controllers for other views, which can be dialogs or other secondary windows [Jaaksi, 1995]. The controllers created by the main controller are called subcontrollers. Their methods can be called directly by the main controller, but the subcontrollers should call the main controller through abstract partner interfaces [Jaaksi, 1995]. Therefore the main controller often inherits multiple abstract partner classes as it can be accessed by the main view, model or other controllers [Jaaksi, 1995].

Recent implementation of MVC++ for Java abandoned the idea of abstract partners for two reasons. Generally, complex views are usually application specific and therefore not reusable anyway [Bonnet, 1999]. So being, the view and controller are allowed to call each other directly in the MVC++ Java implementation [Bonnet, 1999]. In addition, the observer pattern [Gamma *et al.*, 1995] can be used to replace the abstract partners in controller-controller and controller-model relations [Bonnet, 1999]. Unlike abstract partners, the observers offer one-to-many relationships between the interacting parties [Bonnet, 1999]. In general, the solution is very similar to the event model in Java AWT and Swing libraries and thus familiar to Java developers [Bonnet, 1999]. To sum it up, the Java implementation of the MVC++ allows the views and controllers to be bound statically together and it applies the observer pattern in controller-controller and controller-model interactions.

2.2. Applying MVC++ in Practise

It has been shown that by using abstract partners it is possible to replace all parts of an MVC++ application with new implementations. However, in practise many domain specific parts of the application are often built from scratch and replacing the parts of the implementation later is not considered. Reusability in that case often means using generic services and fairly simple reusable UI components. So being, all complicated domain specific functionality and the user interface are practically rebuilt every time.

My own experience has shown that it requires a lot of discipline and patience to keep the application implementation purely MVC++ structured. In addition, MVC++ is sometimes applied in development environments that are not based on C++ and especially in those cases the shortcuts exist and there can be strong reasons for taking them. The adjustments made for the Java implementation of the MVC++ also indicate the need for environment specific modifications [Bonnet, 1999]. The original desktop environment for applying MVC++ was based on X-Motif and also the architecture of user interface applications in other environments can have significant differences. Moreover, using abstract partners is not possible with all programming languages. Under these circumstances there can be various reasons to bypass the MVC++ architecture in some small details of the implementation.

The MVC++ architecture can as well be interpreted in multiple ways and this leaves software developers a bit too much freedom. This is potentially harmful. For instance a set of objects from the model layer can be passed directly to the view component by the controller instead of making an abstraction of presentation. When the view component has the actual model objects it has the total freedom to access and modify them. If that is done, the MVC++ layers are still formally in place, but the view has made an application specific decision that breaks the MVC++ on a logical level. A better solution in this case would be to pass the model objects back to the controller with the information about the user's changes and let the controller make the decisions. But this also causes more work effort for the designer and the MVC++ guidelines do not define this detail very clearly.

The downside is that once MVC++ model is broken, there is no turning back. In the worst case the model becomes dependent of the view and is thus no more reusable. On the other hand the view may contain some logic that actually belongs to the model part. Even if the view makes a simple decision on behalf of the controller, it is not anymore completely reusable. When these rough corners start piling up, your application slowly turns into a non-reusable entity that has no clear structure, and is therefore hard to maintain.

2.3. Laborious MVC++

Although MVC++ has proven to improve software architecture and quality, it has some downsides as well. The number of classes grows and more work is needed to handle the domain specific data. The following sections cover these issues in more detail.

2.3.1 MVC Architecture requires many Classes

The MVC++ model defines three modular layers and interfaces between them. In practise layers and interfaces are realised as classes that divide the implementation into manageable parts in several files. Jaaksi [1994] uses a simple bank application with two views (windows) as an example. The views are also split into visual and functional classes as Nokia Telecommunications design methods required for X-Motif applications at the time. Due to MVC++ rules and working methods a fairly simple functionality is split into eleven classes in the object model. Implementations of those classes are fairly simple, but it is hard to get the idea if you do not know the MVC++ architecture beforehand.

Of course the MVC++ is not at its best in these small examples. In larger applications MVC++ has more benefits and power. It divides the functionality nicely into separate logical parts, especially if the application consists of a few fairly complex windows that allow many user actions. In that kind of application the number classes does not grow so much, but nowadays the user interface guidelines usually demand a different approach. A typical user interface has a main window, which is the starting point of almost all action sequences in the user interface. Many actions in turn open their own dedicated dialogs that should be fairly simple and usable. This is good for the end user, but it also requires more user interface design and increases the number classes in the MVC++ application. All views have own dedicated classes with abstract partners and some of them have own controllers and maybe even models. A typical user interface application may have about twenty windows or dialogs. Even if five of them are completely reusable this can still easily mean about fifty classes in the implementation. Complex views usually have complex model parts, and then the model layer can easily define ten to twenty more classes and the total number of classes can be about 80 or 90.

A user interface application of this size is usually so big that it will be designed to have more than one module or library with well-defined interfaces. When this kind work is done in many separate projects, some questions should be asked. The goals of MVC++ architecture were modularity, clear structure and reusability. If the user interface applications still tend to grow so big, is there something more that could be done to increase reusability? Are there tasks in a typical user interface applications that are performed in almost every application, or over and over again inside the same application?

2.3.2 Domain Object Parameters in MVC++ Model Part

When a new application is going to the implementation phase there are many choices to speed up the development. Possibly there are libraries, templates or even frameworks for doing the skeleton of typical MVC++ application. Graphical User Interface may be designed in some GUI builder, which in PC platform often is actually a complete development environment. After a fast kick-start with the user interface there is still a lot of work left.

Let us assume that we use C++ to implement an MVC++ application in which the architecture is designed as presented by Jaaksi [1994]. Our assumed client-server product should display and modify many different objects and the parameters contained in them. Now is the time when we start adding support for different parameters.

We shall skip modifications in the server and client-server interface part as it is out of our scope of interest in this context. First we need to add the new parameters to the model part implementation that describes the real world we are dealing with. Then we need to support the new parameter in the user interface. This means new fields or mapping of the parameter to some existing component in a view part of the application. As the view should not call the model directly [Jaaksi, 1994], we also need to add support for new parameters in a controller part. Furthermore, handling the new parameter may need some additional logic that must be implemented either in controller or model, depending on which kind of decisions should be made.

Adding each parameter means changes at least to two source files (C++ header and implementation) in model, view, and controller layers. Possibly also the abstract view partners have to be changed, and therefore the new parameter usually causes changes from six to eight files. With Java the number is most likely four, meaning changes in all MVC layer plus at least one observer class. If this seems laborious at the first time, it is even more so in maintenance phase, when implementation details are have been partly forgotten or a new developer is maintaining the product.

Again, a question arises. When model objects need to support new parameter or parameters, could there be one single or a few well-defined places where the support for new parameters could be added? Could such places be centralised to fit inside the model classes that need to have the new parameters? Furthermore, could there be a unified way to bring the information about the model objects and their parameters automatically to the view part of the application? If these issues could be solved by a generic approach, it would reduce the effort of creating user interfaces.

2.4. Reducing the MVC++ Overhead

It is time to make a couple of statements. In my opinion coding the functionality of user interfaces is not very productive; a lot of the work repeats itself. When a new window or a user interface component is created, it usually needs some external data. This data often comes from the model part and when the view retrieves it through the controller, the data is in some format that is not compatible with the user interface components. Thus, the controller or view does some interpretation work to create a set of data that fits inside the user interface component or components. This is the ordinary way of putting to data into the user interface components and it is repeated for different applications and even inside the same application.

Holub [1999a] criticises the MVC architecture strongly by stating that it fails as an application-level architecture. He introduces a visual-proxy architecture, where objects have very strong control of themselves [Holub, 1999b]. In short, the objects have control over their internal data, behaviour and even the user interface for editing the object data [Holub, 1999b]. This solution implements all layers of the MVC architecture inside each object class, and thus seems to break the modularity of the MVC. A rationale for doing this is the fact that it saves the developer from passing parameters in and out of all the MVC layers, as the objects of course can access their data internally [Holub, 1999b]. Although Holub's solution is far from traditional, it has many valid points that are worth consideration. Effectively, when the model is changed, work focuses on a single class: to the implementation of the model class itself. That is a great advantage when compared to the typical MVC++ architecture implementation. However, I do not see that the visual-proxy architecture can be applied to large data sets very easily. If each object defines its user interface, handling thousands of objects can cause some performance and memory problems. Visual-proxy architecture may be good for editing small amount of objects in form based user interfaces, like Holub [1999b] presents it. However, the same solution does not necessarily scale well enough to components that display a large set of objects, for instance list, table, or tree components.

To support large data sets, I will concentrate more on the abstraction of data than providing object specific user interfaces. I think that it is possible to define the objects in the model part of the application so that they always have a common interface when they are handled in the view layer. Furthermore, it is also possible to convert the data so that it is automatically suitable for different user interface components. This claim applies to model objects and individual parameters inside them. What it means is that once the application developer has defined a structure and content of the model data and implemented the model classes of the application, the data can be shown in the user interface components with minimal effort. It also means that when new parameters are added the changes in the code are more local than in a typical MVC++ application. This was one of the goals that Holub [1999b] reached with visual-proxy architecture.

Having made those statements, I should next find a way of putting some credibility behind them. Therefore, it is time to go further in the world of object-oriented design and study the principles of object-oriented frameworks.

3. Object-Oriented Frameworks

This chapter describes object-oriented frameworks. The concept of object-oriented framework is studied and different characteristics of frameworks are introduced. On the surface frameworks seem very similar to class libraries, but a closer look reveals some differences. This chapter defines the concept of frameworks in the scope of this thesis. It also studies the frameworks in relation to other close concepts in the field of object-oriented design.

3.1. Framework Definition

Framework seems to be an overused term in a software industry, as it suits to many occasions. For instance, it can be used in description of high-level system architecture, in connection with the way the software documentation is produced or in relation to project management and administration guidelines. Frameworks can also be wide and colourful sets of separate services that support software development. From now on I set the concept of the framework to mean object-oriented characteristics of an object hierarchy, which make it an object-oriented framework. Thus framework should be interpreted as object-oriented framework, unless otherwise mentioned.

As Lewis [1995] states, frameworks are more than static services as they also define dynamic portions of the program. He sees object-oriented frameworks as class hierarchies that know how the objects in the hierarchy interact with one another. Whereas Lewis [1995] usually considers frameworks as minimal, but already working applications, which can be extended, Koskimies [1997] states that a framework is a collection of interrelated classes that forms an application or a significant part of it, when those classes are properly complemented. The Taligent white paper states that in addition to the benefits of object-oriented programming frameworks provide a working infrastructure for developers [Taligent, 1993]. Johnson's definition [1997] contains the idea of frameworks in simple but sophisticated form:

> *"A framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact."* [Johnson, 1997]

Pree [1995] takes a little more concrete approach by stating:

> *"A framework is a collection of abstract and concrete classes and the interface between them, and is the design for a subsystem."* [Pree, 1995].

These definitions do not set frameworks into a specific domain in the real world or application domain. This is easy to see as well by studying all the three framework types described by the Taligent white paper [Taligent, 1994]:

Application frameworks are suitable for different types of applications. They provide functionality that fits into many application domains. Many such frameworks assist for instance in building user interface applications.

Domain frameworks are built for a certain domain. They offer specialised logic for example control systems, multimedia service or data access.

Support frameworks implement system-level services. These kind of frameworks come handy in situations where existing software must be able to support new hardware or other technological advancement. In such cases support framework isolates the writing a new device driver behind its own interface and there is a minimum effect to the rest of the system.

As this thesis will cover the design of a concrete framework, both of the presented definitions are valid. Another definition by Johnson [1997] is very close to Pree's definition, by referring to frameworks as customisable application skeletons. In the scope this work, I prefer Pree's definition as we are later going to handle the design and implementation of a framework, that is a generic subsystem for presenting and modifying domain specific data in user interface applications. So, from this point on the term framework refers to object-oriented frameworks as defined by Pree [1995].

3.2. Frameworks and Class Libraries

The concept of class library sounds more coherent than a framework. Drawing a line between class libraries and frameworks is not a trivial task, as both can have individual features belonging to the other. In practise class libraries and frameworks often complement each other [Fayad and Schmidt, 1997].

Class libraries and frameworks can be seen to form a continuum. In Figure 3-1 class libraries are in the other end and frameworks in the other. [Taligent, 1994]

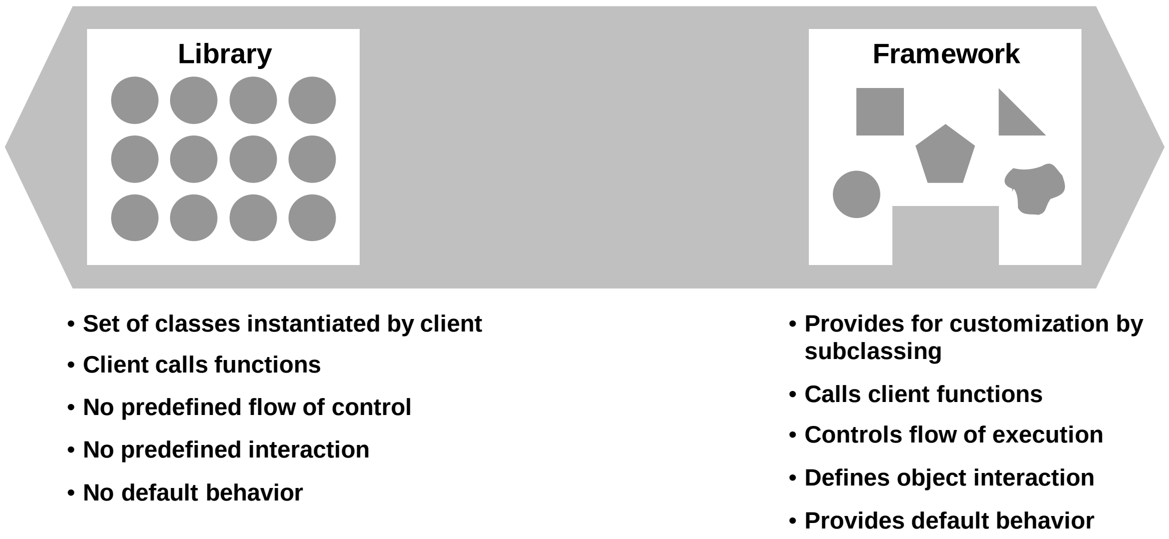

Library
Framework
• Set of classes instantiated by client
• Client calls functions
• No predefined flow of control
• No predefined interaction
• No default behavior
• Provides for customization by subclassing
• Calls client functions
• Controls flow of execution
• Defines object interaction
• Provides default behavior

Figure 3-1. Taligent's view on class library – framework continuum [Taligent, 1994].

Instead of routines in a class library, framework user reuses design solution shared by framework classes [Koskimies, 1997]. Moreover, Koskimies [1997] specifies three forms of reuse: composition, specialisation of classes and the reuse of the flow of control.

According to Fayad and Schmidt [1997] the power of the frameworks comes from increased modularity, reusability, extensibility and inversion of control. For a software developer the significant difference between using class library and framework is in the working method. When class libraries are used the developer exploits bits and pieces from the class libraries and defines the skeleton of the application by himself. With frameworks the developer defines the bits and pieces and the framework defines a skeleton for the whole application or a part of it.

3.3. Frameworks and Design Patterns

Design patterns are very essential part of object-oriented software development. They are generic solutions to reoccurring problems that are constantly faced in designing software as well as any other products [Gamma *et al.*, 1995]. Both design patterns and frameworks reuse the design.

Like design patterns, frameworks also solve repeating problems, but there are a few clear differences between the two. Frameworks contain implementation, but design patterns are abstract and must be implemented every time they are utilised. Frameworks usually use several design patterns, but design patterns never use frameworks, because the design patterns are smaller entities. Moreover, frameworks are more specialised than design patterns. Frameworks solve application domain specific problems, whereas a design patterns are generic and apply to many problem areas that have similar characteristics. [Gamma *et al.*, 1995]

3.4. White-box and Black-box Frameworks

Frameworks can be divided into white-box and black-box frameworks according to the way they are used in building the applications. White-box frameworks are utilised by inheritance and dynamic binding, whereas black-box frameworks rely on object composition done by the user [Fayad and Schmidt, 1997].

Usually frameworks are invented from concrete problems that are faced and solved repeatedly in software projects. Designer finds some abstraction that suits to more than one occurrence of the same problem. Parts of the earlier solutions are found to be generic and they are collected into an abstract class defining the common logic that handles the problem. Once the abstract class has been designed, subclasses can then inherit it and define the application specific functionality. This is the way white-box frameworks work. The user tailors the application behaviour by defining and overriding framework methods in subclasses [Johnson and Foote, 1988]. Actually a partly abstract class can be a minimal white-box framework, although practically all frameworks consist of multiple classes.

In order to use white-box frameworks, the developer must understand how they work [Johnson and Foote, 1998]. The developer must know when and why the framework is calling his methods. Johnson and Foote [1988] also state the fact that an application developer must be familiar with the structure of the framework itself. A fairly large amount of subclasses are created when an application is developed using a white-box framework [Johnson and Foote, 1988]. This may cause confusion, as the complete logic is not visible in the application. The scattered logic is especially hard for a new software engineer that tries to study the application in order to maintain it [Johnson and Foote, 1988].

Black-box framework behaviour is customised by components, which can be used from a separate component library. In this case the framework user must only know the interface of components, and thus such frameworks are called black-box frameworks [Johnson and Foote, 1988].

The learning curve of a black-box framework is easier, but also the potential for customisation depends on the available set of components [Johnson and Foote, 1988]. If the desired component is not available the framework user could create a suitable one by implementing the abstract interface of the needed component, but this again is the way things are done with a white-box framework. Over time the frameworks tend to get more and more black-box features, and in many cases frameworks are somewhere between the two extension methods [Fayad and Schmidt, 1997].

3.5. Deploying multiple Frameworks

Taligent's programming model is building strongly on top of existing frameworks in their environment [Taligent, 1993]. In fact, Taligent [1994] even encourages designing new frameworks along with new applications. This process surely requires highly controlled and mature development process.

Well-designed frameworks, all acting in separate domains, are potentially able to act or co-exist together inside the same application. As an example Johnson and Foote [1988] mention FOIBLE, which is a framework built on top of Smalltalk MVC framework. Similarly, the Java MVC++ framework built in Nokia Networks is targeted to serve all user interface application developers [Bonnet, 1999]. Such frameworks do not restrict the application above the very basic structure, so it is fairly easy to deploy another framework on top of it. With more detailed and restricting frameworks the situation may become more complex. According to Koskimies [1997] problems may arise when two frameworks assume that they have the control of the application or when two frameworks leave a gap between them in the application functionality. The control problem is clearly caused by frameworks that work at least partially in the same domain area. The gap problem may emerge for instance when the two or more frameworks were not originally designed to work together.

Many frameworks may be deployed in a single application, without even giving it much consideration. In Java, for example, there are event delegation model (AWT), data models for several advanced user interface components (Swing), collections framework and object serialisation [Java2]. All those provide framework features and they may be used for instance in an application that is built on top of Java MVC++ framework designed in Nokia Networks. The application utilising the mentioned frameworks would still be perfectly able to use another framework that would have a different purpose, for example defining the data specific operations in the application domain. Fayad and Schmidt [1997] see that application development is turning more and more into utilising and integrating multiple frameworks. It is reasonable to agree with that point of view, as it possible to develop ever more complex software only by extensive reuse.

3.6. Framework Benefits and Pitfalls

When the decision to build a new framework is made, it should be remembered that frameworks are a long-term investment. In general, building a framework is slower than creating a traditional library and the learning time for designers that start to use the framework is longer. [Taligent, 1994]

Benefits are gained only by multiple uses of the framework [Taligent, 1993]. It is possible to reduce long-term development costs by applying frameworks, as well as by collecting the domain experts' experience [Taligent, 1994]. Frameworks also improve the consistency of applications and reduce the maintenance work as errors fixed in the framework automatically affect all the applications using it [Taligent, 1994]. Furthermore, frameworks let the programmers focus more on the value-added functionality of the application as the generic solution already exists [Taligent, 1994].

Framework development disadvantages are as well unavoidable. Building a framework and learning to use it takes time [Taligent, 1994]. As the control of the application is partly in the hands of the framework itself, a framework user must be able trust that the framework functions correctly. If errors occur finding them can be difficult, as frameworks are harder to debug [Taligent, 1994]. In addition to normal technical documents, frameworks require more extensive documentation with examples for framework users to support the learning of the framework [Taligent, 1994]. Framework users also require support when or if the documentation does not answer their questions. One must not also forget that the framework itself needs maintenance [Taligent, 1994].

From the project management viewpoint, frameworks seem risky. Developing a framework is expensive and the framework alone does not have any value. Using the framework should produce some results and realise cost savings very quickly, but in practise such savings may not be visible in the first projects that apply the framework. Projects that try out a new framework should not be in the most critical path, as schedules can be affected by framework development [Johnson and Russo, 1991]. Iterative development rounds and requirements found in other projects may also create pressure for changing the framework [Johnson and Russo, 1991]. Those changes affect all the projects deploying the framework and therefore it is reasonable to wait until the framework interfaces are relatively stable before taking the framework into extensive use [Johnson and Russo, 1991].

3.7. Framework Life Span

Usually frameworks are born as white-box frameworks. Over time they will absorb more black-box framework features. This process takes time and it is often not possible to tell what kind of black-box framework evolves from the original white-box design. The evolving process requires profound understanding of the framework and its domain area. Some frameworks will never reach the goal of being a pure black-box framework, but this does not mean that they are not usable for building applications. White-box frameworks can greatly support the product development. [Johnson and Foote, 1988]

White-box frameworks are usually harder to learn and use, but at the same time they can be more customisable. It is a good idea to offer both black- and white-box specialisation mechanisms for the framework. Designing classes for black-box specialisation also gives an early first impression of the extensibility of the new framework. Evolving towards a black-box framework may actually require that framework is deployed in a few projects. After this the common features that could be provided as black-box extensions can be identified [Brugali *et al.*, 1997].

Frameworks need routine maintenance caused by new requirements or changes in the environment. This causes pressure for the application using the framework to be changed as well [Fayad and Schmidt, 1997]. If the interfaces are changing this is obligatory. Reasons causing an optional change can be for example increasing performance or clarity of application design. A well-designed framework of course tries to minimise these changes, but the iterative nature of development often creates needs for the changes.

Eventually the framework lifecycle comes to an end. This could happen for many reasons. Possibly a new hardware or software technology makes the framework obsolete or unusable [Viitanen, 1999]. Alternatively someone may find a better solution for the domain the framework is handling and this solution will replace the framework [Viitanen, 1999]. In the software industry changes happen all the time and therefore it can be very hard to anticipate how long the framework will be in use.

4. Developing Object-Oriented Frameworks

Designing frameworks is quite different from designing a specific software product. In this chapter the design process and rationales for framework development are discussed. The roles and needs of framework designers and framework users are taken into account.

4.1. Identifying Abstractions

Object-orientation has helped the software industry by changing the way systems are designed so that the object-oriented design encourages reuse of design and code [Johnson and Russo, 1991]. One task in object-oriented design is finding the abstractions. As it is very hard to create new abstractions, they should be identified from existing solutions and examples [Johnson and Foote, 1988]. This often leads to a repeating process where abstractions are revised on the basis of new experiences [Johnson and Russo, 1991]. Implemented operations in abstract classes call the unimplemented ones and this way they exercise both reuse of design and code [Johnson and Russo, 1991]. On a practical level you should find software solutions that are built repeatedly and in addition recognise the common and unique parts in them [Taligent, 1994].

Johnson and Foote [1988] point out that reuse does not happen accidentally and thus the experts are responsible for reusing old and looking for new reusable components. Abstractions can be found by defining standard interfaces when designing new classes and by designing minimum size methods that increase modularity of classes [Johnson and Foote, 1988]. By doing careful class design, finding abstract classes becomes easier. The abstract classes should be placed at the top of class hierarchies and the hierarchies themselves should be deep and narrow [Johnson and Foote, 1988]. Access to variables in abstract classes should be minimised and the subclasses of an abstract class should be specialisations [Johnson and Foote, 1988].

Good abstraction emerges when there is a clear understanding of hot spots, places where maximum adaptation is needed [Pree, 1995]. According to Pree [1995], hot spots are the domain specific parts in the abstraction or a framework that must be kept as flexible as possible. The framework user hooks his application into these hot spots and provides specialised implementations for them. If hot spots have been correctly identified and flexible customisation is supported in them, the framework is meaningful to use and it becomes more powerful. Finding the hot spots is a task where domain area experts are needed [Pree, 1995]. A higher level flow of control can be defined around the hot spots, thus combining their functionality into a larger entity.

4.2. Designing a Framework

Johnson and Foote [1988] point out that designing good frameworks is harder than designing abstract classes. An idea of a possible framework is usually born from experience [Johnson and Foote, 1988] and therefore a framework should targeted to the area of designer's expertise.

A good framework is complete, flexible, extensible and understandable [Taligent, 1994]. This means that the framework user understands how the framework works and is able to concentrate on the value-added functionality in his application [Taligent, 1994]. Framework can also solve problems of different applications and when something is missing or needs to be redefined, the framework does not prevent extensions or modifications [Taligent, 1994]. Obviously all these features are hard to reach in single framework, but it is good to measure against them, as they are the requirements that the framework users expect you to meet.

Good design and programming practises can also help in finding potential frameworks in existing projects. When developing software you should always consider splitting large classes that have grown over time. Similarly you should divide differences in the same operation into separate subclass implementations and define the abstraction of the operation in a common superclass that can be declared abstract. You should also implement generic operations outside your classes and utilise them from many classes. Classes should not contain parameters that indicate some internal state, as they are practically same as global variables. Such information should be delivered as parameters in method calls. [Johnson and Foote, 1988]

Johnson and Foote [1988] also state that designing a white-box framework first is a good starting point. As the actual application and test program design and usage tests the fitness of the framework, the iteration rounds are essential in guiding the development to the right direction [Johnson and Russo, 1991]. Testing the framework in pilot projects can reduce the risks [Johnson and Russo, 1991]. This can also be done during the earlier iterative rounds, as possible corrections do not have any critical impact. Extensive use of the framework in early stage can be a serious mistake as the framework interfaces keep changing and cause a lot of maintenance work in projects that already are developing on top the framework [Johnson and Russo, 1991].

Abstractions for frameworks can be found on the solutions that are built repeatedly [Taligent, 1994]. Unique and common features should be identified and separated. They are the cornerstones of new framework or frameworks. The abstractions can be discovered from existing solutions or an experienced designer can invent them based on his past knowledge and expertise [Taligent, 1994].

If you can utilise design patterns in your frameworks, you built on the common knowledge that helps framework users to better understand the structure of your framework. Design patterns can also help you to save time in your framework development [Taligent, 1994].

You should also consider whether you could provide specialisation of abstract classes on top of your framework [Taligent, 1994]. Black-box type default implementations can greatly help the framework user, as there is no need to learn the internal functionality in detail. The number of classes as well as the number of member functions should be minimised [Taligent, 1994]. A balance in flexibility should be found; if the framework is not flexible, it is neither very usable. On the other hand too much flexibility creates a lot more work in the framework implementation and usually makes the framework even harder to use or learn.

On some domain areas a very deep knowledge is needed to become an expert. If a framework is to be implemented for such a domain, an external expert must be allocated.

Koskimies and Mössenböck [1995] describe a framework design process through step-by-step generalisation. In the first phase the generic concepts of problem domain, application or framework, are iteratively examined and mapped to design patterns and other solutions [Koskimies and Mössenböck, 1995]. After multiple generalisations the problem should be at its most general form [Koskimies and Mössenböck, 1995]. In the second phase framework or frameworks for generalisations are designed in a reverse order going from the most generic solution towards the most complete [Koskimies and Mössenböck, 1995]. The last framework design that is produced by this process should be the most complete, the one that solves the original problem [Koskimies and Mössenböck, 1995].

Koskimies and Mössenböck [1995] admit that solving real and complex design problems with this method may be hard and the actual design process may not proceed so cleanly. However, they point out that knowing the stepwise generalisation process can help the framework designer, even if the process is not followed in detail [Koskimies and Mössenböck, 1995]. After all, building a framework typically includes finding abstractions, iterative design and implementation [Koskimies and Mössenböck, 1995]. Basically Koskimies and Mössenböck describe more formal and structured way to perform those steps.

Frameworks may use services of another frameworks. These kinds of dependencies can be hard to manage and therefore it is wise to consider minimising them. Taligent [1994] states that frameworks should be connected with an interface or server object, which makes dependencies minimal. If designing such an intermediate class is possible, it can provide architectural benefits and clarify the design.

4.3. Framework Implementation

It is possible to implement a framework without object-oriented programming language [Johnson and Russo, 1991]. However, as the object-oriented design and programming have a strong position in the current software development, I will concentrate on framework development process only with object-oriented languages.

Pree [1995] introduces two types of methods for framework design: template and hook methods. Template methods are based on hook methods, which in turn can be abstract methods, regular methods or other template methods [Pree, 1995]. Template methods implement non-changing, frozen spots in the framework, thus defining implement abstract behaviour and flow of control [Pree, 1995]. Hook methods define the hot spots and they are more elementary as they are called by template methods [Pree, 1995]. Whether a certain method in the framework is template or hook method depends on the point of view. As an example, a smaller scale template method calling some hook methods can act as a hook method for a template method of larger scale [Pree, 1995].

Framework designers should keep an open mind for new solutions even in the implementation phase. As Pree [1995] states, the evolving process of framework development may create new requirements and needs revising design. That is why framework developers should always be ready to design new features and redesign the old ones during the implementation [Pree, 1995].

Taligent [1994] also encourages a prototyping approach for building frameworks, for example by implementing a subset of problem domain. Concrete experiences in prototyping approach can be collected in very early phase, but also the developers trying to deploy the early prototype are lacking a good documentation and they often have to update their application because of changing framework interfaces. In general prototyping can be seen as first iteration round, and therefore the first actual release may be more ripe and complete. This can be extremely important benefit when multiple applications are being designed on top of the framework.

4.4. Testing and Supporting the Framework

Testing the framework has two sides, the technical and practical. Technically a framework developer may be able to fully test the framework. Technical testing tests that the framework actually works as expected and therefore test drivers or otherwise automated testing can be very useful. With the frameworks that define user interface functionality by themselves, an interactive testing with small test user interfaces may be necessary. This is because there might be no reliable way to pass user interface events automatically to the framework components. Technical testing may also utilise other testing tools, such as test coverage analysers, code optimisation or analysis tools.

I consider the practical side of framework testing as testing against known and unknown requirements. A framework developer is possibly able to perform tests against the known requirements as well. For the unknown requirements the framework needs testing that is done by application developers using the framework [Taligent, 1994]. This is the first opportunity to gather new requirements and improve the old ones. This process also requires a lot of discipline, because the flood of new requirements can be overwhelming. You should select a reasonable set of requirements to be implemented, but most likely some of them have to be dropped or implemented later.

You should regard your frameworks as products [Taligent, 1994]. Even though they may not directly make any profit, the developers using your framework have the role of clients. The quality of framework code and comments in it are therefore essential, but still not enough [Taligent, 1994]. Framework users expect to see sample programs, technical documents with architecture diagram, verbal descriptions of the framework and its usage [Taligent, 1994]. In other words, you have to reveal the nuts and bolts of your design by example and guidance, as it will be too hard to find and learn them otherwise.

Johnson [1992] gives framework documentation three goals. It should describe the purpose of the framework, how to use it and the detailed design of the framework. Johnson [1992] uses patterns to document frameworks. In this context they mean informal, but structured essays about different features of the framework [Johnson 1992]. The first pattern should give an overview of the framework domain, and the others should concentrate to specific topics, gradually deepening the reader's knowledge of how to use the framework [Johnson 1992]. Technical design is easier to learn, after the reader has had some experience of using the framework [Johnson 1992]. That is why Johnson

[1992] thinks that the framework design should the last issue in the documentation. This indicates that documenting technical design is targeted to users who really need in-depth knowledge of the framework or are just otherwise curious of what is actually happening under the hood. Not having this knowledge should not prevent the developer from using the framework itself [Johnson 1992]. This is true especially for black-box frameworks, but with white-box frameworks some design details may be necessary to know.

Emphasising further the importance of documents, it can be possible to extend your framework with extensive documentation. Just outside the reach of framework domain, there can be tasks or operations that are frequently needed in the applications using the framework. As framework developers know the hearts of the framework, such issues should be brought to their attention. Knowing those problems, they can also find a sophisticated ways to implement such features in the applications by effectively using the framework to partially perform such tasks. When these solutions are written down to cookbooks or corresponding documents, they actually extend the scope and usability of the framework further and by doing so, they provide additional value to the framework as well as to the applications.

4.5. Framework Maintenance

After the framework release and some projects that have applied the framework, the knowledge of the problem domain and true framework requirements have probably improved [Taligent, 1994]. When the framework maintenance is started, you should carefully collect as many experiences as possible and design and prioritise the maintenance work accordingly. A few aspects of the framework maintenance phase are discussed here.

All known errors in the framework should be fixed as fast as possible [Taligent, 1994]. Correcting errors affects all the software using the framework and also prevents software developers from making additional code just to avoid existing errors. Later, such solutions may cause new problems to arrive. Naturally the severity of errors must be considered, and work must be prioritised.

Initially during the early development of the framework, it is easy to refine and change existing interfaces and you should not hesitate to do so [Taligent, 1994]. Eventually, the time of freezing the framework interfaces comes. After this the interfaces should be kept as stable as possible, as the existing software is already using them [Taligent, 1994]. This is the point where new functionality should be added by extending the old interface instead of making changes in the existing parts [Taligent, 1994]. In some cases however, there might be very strong reasons to change the existing interface. If this must be done you should notify the users of the framework early enough and also find a good timing for making those changes [Taligent, 1994].

Once both framework and application developers have gained experience, one important task in the maintenance phase is to identify places where usability of the framework could be improved by offering new black-box functionality. Those ideas can emerge from the questions and feedback received from developers using the framework. It can also be worthwhile to study the software that has been implemented by using the framework. One or more developers may have found a good way to complement the functionality of the framework in their own code. If that kind of solutions can be adapted to be a part of the framework itself, it will create additional value to the framework.

4.6. Framework Development Tools

In theory, frameworks can be built for any domain. They can work in different areas of the application and multiple frameworks can be utilised to build a new application. Conceptually, frameworks are hard to understand and finding a common denominator from them is not so simple. Framework design and implementation often leads to long and unpredictable roads [Hakala *et al.*, 1999]. Therefore, building a development tool for building frameworks is a challenging task.

Universities of Helsinki and Tampere have built FRED (Framework Editor), which is a tool supporting framework development. FRED's model of framework development is based on specialisation templates and design contracts. Design contract is described as a design pattern without the semantics of the pattern, thus being structural sets of classes that fulfil certain conditions and constraints. Like classes in object oriented design, design contracts can extend other design contracts. In FRED, a language Cola is used to describe them. [Hakala *et al.*, 1999]

A specialisation template is based on a certain design contract and fills a flexibility requirement of a framework [Hakala *et al.*, 1999]. It is used to bind a generic design contract into the framework [Hakala *et al.*, 1999]. Unbound entities are then left to application developer, but specialisation templates may also extend more general specialisation templates [Hakala *et al.*, 1999]. This leads to process of stepwise implementation [Hakala *et al.*, 1999], which has also been described by Koskimies and Mössenböck [1995].

The purpose of FRED is to help and guide a developer to specialise a framework into an application [Hakala *et al.*, 1999]. FRED maintains a list of tasks that need to be done and can create also default implementations for some tasks [Hakala *et al.*, 1999]. In addition, FRED can notice when user implementation does not conform to the requirements of the framework [Hakala *et al.*, 1999]. New tasks for correcting such errors are created immediately, when the error is detected in the integrated code editor [Hakala *et al.*, 1999]. FRED can be used to specialise a framework further, thus creating a new specialisation template, or to develop an application on top of existing framework [Hakala *et al.*, 1999]. The project team sees that FRED clears the framework building process, as well as it especially helps the designers to utilise white-box frameworks with lesser knowledge of the framework itself [Hakala *et al.*, 1999]. Personal evaluation of FRED indicated that some learning is needed in order to use FRED, but it also convinced me, that FRED and also framework development tools in general have potential. Undoubtedly, more research is needed to better understand the process of building frameworks to aid framework designers in their work. The FRED team has taken promising steps in that direction.

4.7. Summary

We now have an idea of what frameworks are and what it takes to design and implement them. Identifying the right abstraction from existing solutions is essential and the framework development process has to be iterative and flexible to be able to respond to real needs. Helpful documentation alongside the framework itself has a very important role and one should also reserve time for maintaining the framework. We have also reason to assume that tools and applications for developing frameworks will become more common in the future.

At this point we can again face the problems of MVC++ architecture that were brought up in chapter 2. All MVC++ applications handle the domain specific data in model, controller and view layers. In principle this has to be re-implemented every time, because the data is always different. The starting point here is that if a generic interface can be defined to represent the application data, a framework can then implement generic handling of the model data in other layers of the MVC++ application. This can also mean working support for different user interface components, which require different kind of model implementations in order to visualise the data. In the next chapter, we will address these issues further.

5. Model Presentation Framework

The purpose of this chapter is to describe the Model Presentation Framework, which was constructed in order to reduce the work that is repeated in building many MVC++ user interface applications. In short, the purpose of the Model Presentation Framework is to assist the work of a software developer, when the domain model data must be presented and in the user interface. The developer must implement his model layer so that it is Model Presentation Framework compliant and use the adaptation services of the framework when he passes the data to the user interface components.

5.1. Motivation behind the Model Presentation Framework

The idea of the Model Presentation Framework grew up after experiencing the design and implementation work of user interface applications, mostly network management applications built in Nokia Networks. According to my experience, some common characteristics of user interface applications can be collected. Typically a user interface application displays or presents the data to the user. When the user interacts with the application, he can execute two types of tasks. The user can perform tasks of the user interface itself, such as managing windows, using menus or customising the user interface. Another type of interaction done in the user interface is actually interacting with the data. This type of interaction can mean for instance navigating inside the data hierarchy, creating or deleting objects or modifying their parameters. Even with the MVC++ architecture, the interaction with the data is usually re-implemented in each user interface application. This is because the data is different almost every time. Implementing data presentation and editing functionality for the user interface can be a hard and time consuming task, especially if the structure of the data is complex and consists of many different types of objects.

If a generic way of describing the data can be defined as an interface, it is possible to define generic operations that handle the data through the defined interface. And as there are common operations for the data, also they can be generalised. The generalised parts can then be combined into a framework implementation, so that they can be utilised in multiple applications. This approach can reduce the time of designing and implementing a user interface application. The use of such framework automatically gives more responsibility to the model classes. This is because they must describe the object hierarchy for the user interface. Therefore the applications based on this framework will have a more predefined way of designing the model classes than the MVC++ architecture, which does not tell much about how the model classes should actually be designed. Using the framework also leads to the situation where the view layer has a minimal amount of information about the application model part. As the model layer is often accessed through the framework, the developer has not such a great temptation to implement the logic of the controller or the model in the view layer. This should harmonise and simplify the application architecture in view and controller layers.

As described earlier, typical data related functions in user interface applications are:

• Displaying the data for the user	(output)
• Navigation in hierarchically structured data	(input/output)
• Modifying the data	(input)

These operations define the domain for the Model Presentation Framework. Java was selected as a development environment, because more and more user interface development in Nokia Networks is moving towards that direction.

5.2. Model Presentation Framework Overview

The Model Presentation Framework defines a generic interface for describing model classes of an MVC++ application. On top of this abstraction, framework defines a functionality, which adapts the Model Presentation Framework model objects for presentation in different user interface components.

The Model Presentation Framework sets some requirements for the application, but gives a lot of services in return. Figure 5-1 illustrates an application that is built around the Model Presentation Framework. White area is the application and the grey boxes are different parts in the framework. The correct way to read the figure is to consider the model part of MVC++ application being at the very bottom, where application model is placed at the picture. The application view is at the top and the controller defining the logic of the application on the left side. Boundaries of MVC++ layers in the framework side are instructional. They show an example of which the Model Presentation Framework layers should be used in view, controller and model layers of the application.

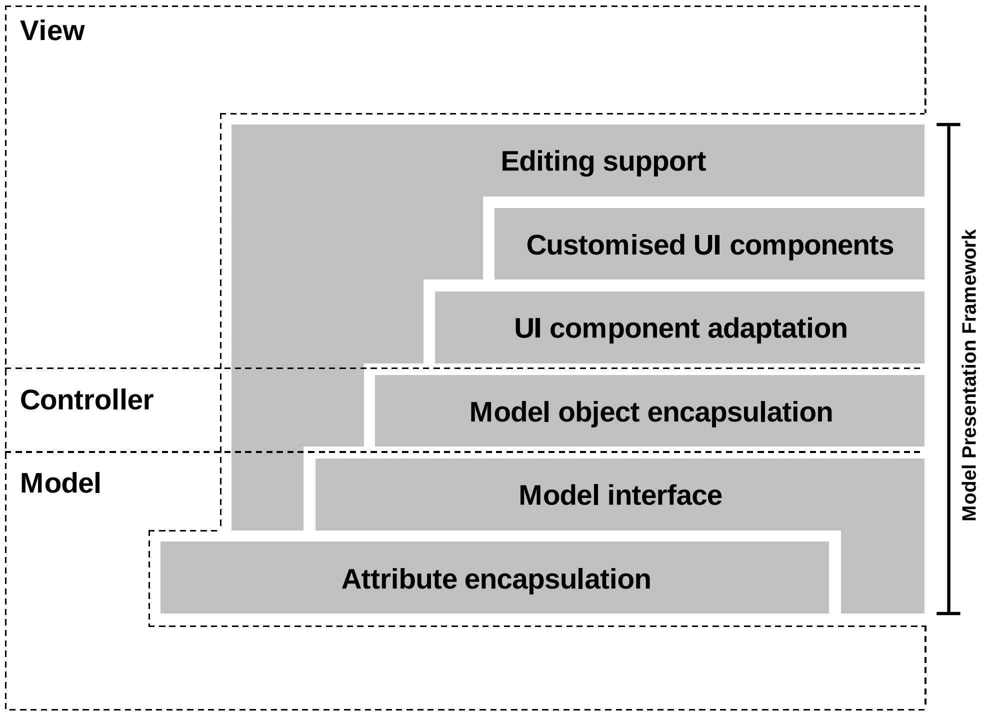


Figure 5-1. An application using the Model Presentation Framework.

The Model Presentation Framework itself has roles in all parts of the MVC++ application. For the model it defines the base classes, some default functionality as well as an interface for describing hierarchical data structures in the application. On many occasions the logic that would be implemented in the controller part for navigation or making modifications to data is already implemented inside the framework. This way the framework takes over some general logic from the application controller. In the view part the Model Presentation Framework defines adaptation for several user interface components, as well as customised user interface components that already have default functionality. Editing support can be seen as a complementing piece that uses the framework presentation functionality on several layers and provides additional pieces on many layers to implement the editing functionality.

To further clarify Figure 5-1 the meaning of different parts of the Model Presentation Framework need to be explained. They are covered in more detail later in this chapter, but following summaries should help to form the overall picture:

Attribute encapsulation: Primitive types are encapsulated into class presentations. Besides the attribute value, an attribute class may contain additional information. This can mean for example a set or range of valid values. Attributes can be grouped into attribute sets, from where attributes are accessible by name. Attribute sets are dynamic and allow runtime insertions, deletions and modifications.

Model interface: Model objects can form hierarchies that describe the application domain. Their attributes and hierarchy are handled through a predefined interface. Model classes have two sets of data. Basic data allows model object presentation without displaying attributes or children. Complete data set includes attribute values and children of the object. Object with complete data can be in expanded state in the user interface.

Model object encapsulation: This layer defines a data node that can store any Model Presentation Framework model object. Model objects are encapsulated inside data nodes before they are presented in the user interface. This means that user interface is able to display many nodes, even if the nodes actually refer to the same model object instance. This prevents an unnecessary multiplication of the data.

User interface component adaptation: As the data nodes are presented in different user interface components that have unique requirements for the format of the data, they must be interpreted separately for each type of user interface component. User interface component specific adapter classes provide this functionality.

Customised user interface components: As the Model Presentation Framework defines functionality that supports automatic expansion of object hierarchies, it is also reasonable to define extended user interface components. They have built-in support for navigation in the model object hierarchy and may define some look-and-feel issues as well. This is a more convenient way of applying the Model Presentation Framework services, as it saves the work of a software developer who builds the end-user application.

Editing support in user interface components: The Model Presentation Framework has a support a limited editing functionality in certain user interface components. This support means providing separate data aware components for editing operations and an automatic transfer of modifications internally through the Model Presentation Framework. The editing components can be integrated into the actual presentation components that display the model object data, which in turn simplifies the work of a framework user.

The Swing user interface components in Java Foundation Classes form a very good basis for these abstractions. When using Swing components, the developer does not have to insert the data into components by hand. The components expect the user to give them objects that implement a certain interface, a component specific model that describes the contents of the data [JavaTutorial]. The problem is that these interfaces are component specific [JavaTutorial]. On one level the Model Presentation Framework provides a new abstraction layer that implements the object presentations for different Swing components. In the architectural level Swing components give an ideal starting point for creating such a higher level abstraction.

5.3. Model Presentation Framework Architecture

Next we will take a more detailed look into the architecture of the Model Presentation Framework. Object models and interaction between different layers are presented. The focus is on the design of class functionality and interfaces rather than specific design and implementation details. Therefore many class attributes and insignificant methods have been left out of the presented object models.

5.3.1. Attribute Encapsulation

Presenting attributes in a generic manner throughout the Model Presentation Framework requires powerful abstractions starting from the bottom layer. Attribute encapsulation layer defines class presentations for all primitive types of Java language. Java already defines class presentations for primitive types [Java2], but to provide additional functionality customised class presentations were designed. This allows assigning more functionality into the common base class, which is great asset when designing upper layers of the framework. In addition these customised classes can be extended to contain attribute specific information if a need arises. Such information can be for example valid value ranges for numeric attributes, or maximum lengths for strings. The overview of class hierarchy is illustrated in figure 5-2.

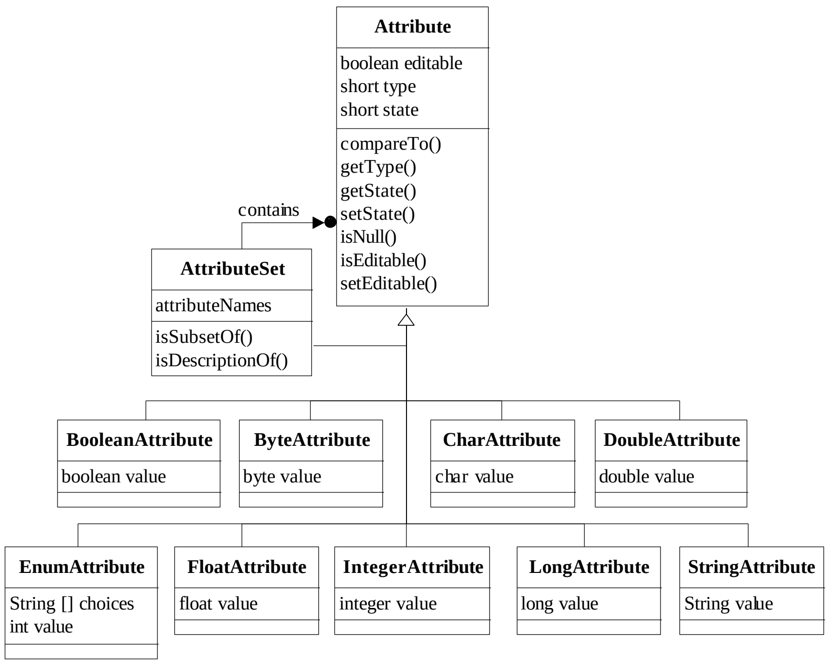


Figure 5-2. Attribute encapsulation layer.

All classes for different types of attributes and attribute sets are derived from the *Attribute* class based on which specific implementations are designed. In addition to primitive types, defining attributes for strings, enumerated types and grouping of attributes into sets are supported. *Attribute* is a base class for all attributes. It has a field for the type of the attribute that can be used for dynamic checking of attribute type. The state field does not have any meaning inside the framework and the application user can choose to use it in some meaningful manner, if there is one. The state field was created as a response to future project requirements for storing detailed information about the different modifications to the domain specific data. Introducing the state in a base class provides the feasible solution to the problem already at the attribute level.

It is possible to create an instance of the *Attribute* class. Such an instance always represents a null value. *AttributeSet* class makes grouping attributes into sets possible. Construction of attribute sets from instances of *Attribute* class itself is a typical example of applying Composite design pattern [Gamma *et al.*, 1995]. There is also a method that can check whether one *AttributeSet* contains a subset of another. *AttributeSet* has proven to be a powerful abstraction that can be used for performing many operations in the framework. Subsets can be used to implement for instance filtering or selection functionality, which is partly generic.

5.3.2. Model Interface

The model interface is the part that handles most of the interaction between the Model Presentation Framework itself and the application using the framework.

The model interface defines how the application designer must implement his model classes for them to be compatible with the Model Presentation Framework.

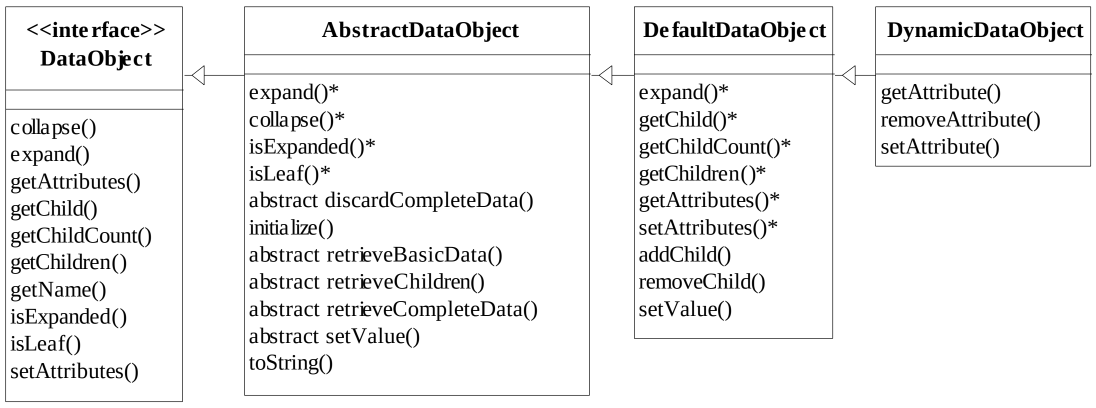


Figure 5-3. Model interface and implementations of *DataObject* interface.

Figure 5-3 presents the interfaces and classes in the model interface layer. The inheritance relationships in the figure are drawn from left to right for presentation reasons. The *DataObject* defines the interface that is used internally by higher layers of the framework. However, this interface is very primitive and quite large and therefore not the most convenient. More specialised classes with default functionality are offered on top of this interface.

Methods implementing the *DataObject* interface are marked with '*' in the figure. The *AbstractDataObject* class defines such methods of *DataObject* interface that are completely generic. They are *collapse, expand, isExpanded* and *isLeaf*. This class also declares some new abstract methods, which are new hook methods called by the methods that are already implemented. With default functionality and new hook methods the *AbstractDataObject* defines the concepts of basic and complete data. An object that is not yet expanded is assumed to have only basic data. In practise the basic data should be enough to visualise the object in the user interface without showing its parameters. This functionality was designed for optimising performance and memory consumption when large amount of objects are presented for example in tree, list or drop down list user interface components. An expanded object in turn is assumed to have a complete data for presenting attributes and the children of the object. The children in this case do not need to be in expanded state. All objects in user interface components that represent object attributes must be in expanded state and therefore have complete data. In practise the Model Presentation Framework ensures that such objects are expanded before presentation. As the user actually implements the application model classes by inheriting objects of the model interface layer, it is up to him to define whether there is any practical difference between basic and complete data set.

The *AbstractDataObject* is the first class that has a reasonable functionality that can be inherited into application model classes. As the name indicates the *AbstractDataObject* class still leaves many methods unimplemented and especially if the framework user wants to define hierarchical model structures a more complete implementation is needed.

The *DefaultDataObject* defines the most complete functionality for the model classes. Only five methods are left abstract, although to build a more complete functionality some methods with default implementations need to be overridden. The abstract methods left are *getName, discardCompleteData, retrieveBasicData, retrieveChildren,* and *retrieveCompleteData.* Methods *getAttributes* and *setValue* have default implementations that do absolutely nothing. They only save the trouble of defining them if they are not needed. However, implementing them is often essential, but also dependent of domain specific data, and therefore the implementation must be left to application developer's responsibility. The *DefaultDataObject* implements default child handling that supports building object hierarchies. This means mostly implementations for child-related methods of *DataObject* interface, but also methods for adding and removing children. Children can be added and removed freely, and the default functionality also serves the upper layers of the Model Presentation Framework reliably. In spite of the most complete implementation, utilising the *DefaultDataObject* is not so easy and this is why very good documentation and concrete examples are needed to assist framework users. In the future the *DataObject* interface could be extended to support sorting of child objects and the *DefaultDataObject* could have the default implementation of that functionality [Virtanen, 1999].

The *DynamicDataObject* class extends the *DefaultDataObject* further by making handling of dynamic parameters possible. As parameters can be added or removed during the runtime of the application, this class is able to provide an extended functionality for special purposes. It could be used for instance in cases where the application should allow the user to specify additional information at runtime and attach this information to any object. It could be also used in cases when the structure of same type of objects can vary a lot, meaning for example that the number of attributes varies greatly between individual instances of the same class.

The model interface layer is the basis for designing application model classes. To pass information to upper layers of the Model Presentation Framework in generic manner, application model classes must use the services provided by attribute encapsulation layer that was presented in the previous section. Model interface is clearly a hot spot in the Model Presentation Framework. Although default implementations from the model classes exist, none of them is complete or usable in the application without inheriting and extending it.

5.3.3. Model Object Encapsulation

An application developer using the Model Presentation Framework defines the application model classes by inheriting them from the classes in the model interface. However, on top of the application model there is a need for yet another model layer. This need can be hard to realise or understand unless you have enough experience of user interface design. Actually, same layers exist in almost any modern user interface, but in those the model object encapsulation is usually designed to be as a part of the user interface component in question, or at least component dependent.

To formalise this issue further, we must define concepts of the data model and presentation model. Data model is closer to real world and it should have as accurate interpretation of the real world state as possible. Data model often has many-to-one relationships between object instances. Understandable examples of such relationships can be found from the daily life. For example multiple persons may live at the same address or many employees can have the same superior. In sports two teams have played the same game against each other. In the world of telecommunications a very good example is the routing of calls. Many dialled digits can map to the same destination that routes the calls to a certain area in the network.

Many-to-one relationships can be drawn as graphs, but many user interfaces have less visual methods for presenting information, such as lists, tables or trees. Those presentations can be seen as sets or hierarchies and typically they visualise the target object of many-to-one relationships multiple times. In a fictional address database application, a tree component in the user interface could show the same address under two or more persons. An existing application called Traffic Flow Management does the same in digit to destination mapping presentation for a switch in mobile telephone network [TFM, 1999].

As we have demonstrated a possible difference between the real world and an interpretation of the real world in user interface, we can find reasons why presentation model should be defined. When the user interface displays the same object multiple times on the screen, it actually needs many instances of the same object. With direct mapping to the real world model, this would mean that copies of the object must be created in order to present the information.

This is a waste of memory and just makes managing the model data more complicated. If the attributes of the object are modified, we must make sure that modifications are made for each and every copy of that object. A hierarchical model requires that a parent can be found for each object. If we have many separate object instances that actually should be the very same object, which one of their parents is the correct one? Such problems can be avoided by defining the presentation model.

The presentation model is a new abstraction on top of the data model. In the Model Presentation Framework it defines a data node, which always has a reference to a single object in the data model. As many instances of data nodes can refer to a single object in the data model, we have solved the problem of many-to-one relationship. In the presentation model, a data node can always have only one parent object, but in the data model there is no need for such relationship anymore. When a single object needs to be displayed multiple times in the user interface, we make copies of data nodes and set them to refer to a single object instance in the data model side. Very likely, the copies of data node instances use less memory than copies of objects that model the real world data. The data node is able to give handle to the real world object behind it, so that a single object is accessible through many data nodes. Data nodes also have children like the real world objects in data model. As a summary the data model can have cross-relationships and can be seen as a graph. The presentation model is an interpretation of that graph as a hierarchy where some cross-relationships have been systematically forgotten and as an implication, object instances are multiplied if need be.

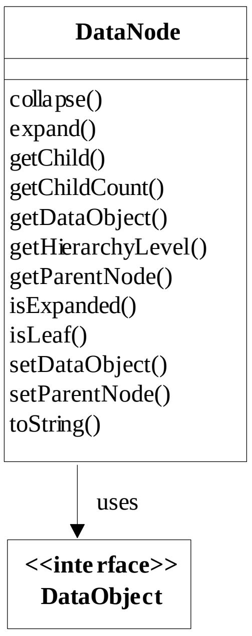


Figure 5-4. Model object encapsulation using *DataObject* interface.

In the Model Presentation Framework design a single class called *DataNode* (in figure 5-4) is used to implement model object encapsulation, in other words the presentation model. It defines the same operations as the *DataObject*, and in many ways is only a call-through layer towards it. The *DataNode* always encapsulates one *DataObject* instance and in addition it contains a reference to the parent node, because bi-directional one-to-one relationships exist on this layer.

5.3.4. User Interface Component Adaptation

As the lower level layers of the framework together form an entity which describes the application model objects and their relationships to each other in a generic way, we now need to define how these objects can be displayed by user interface components in the view part of the application. Figure 5-5 shows the user interface component adapters offered by the Model Presentation Framework. Many Swing user interface components in Java API require a dedicated data model [Java2]. These models have component specific interfaces and therefore a single generic model cannot be used anymore above this layer. The solution is to provide user interface component specific adapters that are able to use the common interfaces of *DataNode* and *DataObject* classes. The adapters in turn implement the interfaces required by Swing user interface components and can serve as a model class instance towards the components.

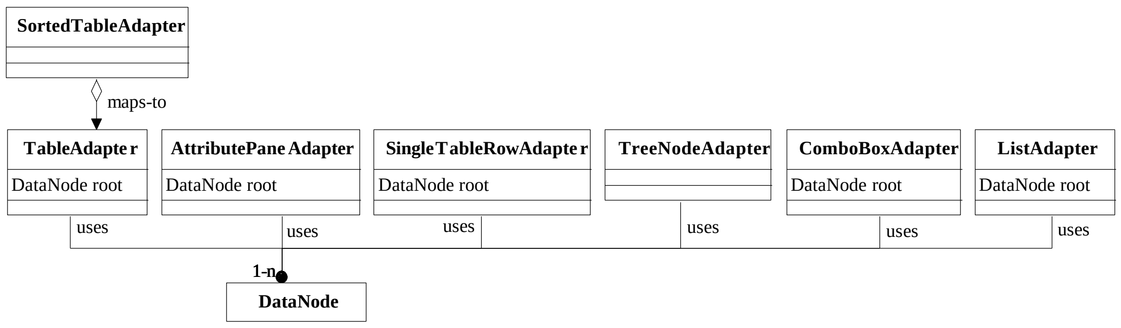


Figure 5-5. User interface component adapters using the interface of *DataObject*.

All adapter classes are designed to be completely compatible with regular Java Swing components. The role of the root node stored inside the adapters varies. If the user interface component is supposed to display the data of multiple objects, the child nodes under the root node are displayed. If the user interface component is expected to display the attributes of a single node, that node is the root node. Table 5-1 explains the purpose of all user interface component adapters.

Table 5-1. Model Presentation Framework component adapters and their roles.

Name:	**Adapter for:**	**Role:**
TableAdapter	JTable	Provides a regular table model for displaying all child nodes under a specified *DataNode* instance.
AttributePaneAdapter	JTable	Provides a two-column table presentation of attribute name and value pairs of the attributes under a specified *DataNode* (attribute names in the left column and values in the right column).
SingleTableRowAdapter	JTable	Like AttributePaneAdapter, except that the attribute names are displayed in the header of the table and the values horizontally in a single row of the table.
SortedTableAdapter	JTable	Encapsulates TableAdapter instance and implements column sorting on top of it.
TreeNodeAdapter	JTree	Adapts a *DataNode* into a single node in the tree. This adapter is unique as every tree node needs a dedicated adapter instance.
ComboBoxAdapter	JComboBox	Maps all child nodes under specified *DataNode* instance into a JComboBox.
ListAdapter	JList	Maps all child nodes under specified *DataNode* instance into a JList.

Assigning data to be displayed in different user interface components is fairly easy for the application developer using the Model Presentation Framework. In principle he inserts the needed user interface component into the user interface, encapsulates the suitable *DataNode* instance, which contains the reference to the desired *DataObject* instance, inside the component specific adapter. Then he passes the adapter as the model for the user interface component. If an existing user interface component is being replaced with another, there may be a need to make changes to the behaviour of the application as the concepts of the whole user interface might be affected by the change. Such changes should not however affect the domain specific model.

5.3.5. Customised User Interface Components

The idea of user interface component adaptation was to provide suitable models for each user interface component that is supported. All user interface component adapters in the Model Presentation Framework work with standard Swing user interface components. The problem is that standard components are not able to provide any additional functionality that supports the internal architecture of the framework. For example customised tree component can have default functionality notifying listeners about the selections or customised table component can implement sorting functionality.

Tree component (*JTree*) is the only component that supports navigation in the object hierarchy. In customised table and list components, navigation can be supported by default. The table component has a ready-made implementation that always encapsulates the table adapters inside

SortedTableAdapter and provides a working implementation for sorting the data. Browsing functionality does not require much custom functionality from the user interface components. Much harder requirements come from the fact that components must be able to support usable editing functionality. Next we will take a look at that.

5.3.6. Editing Support

Editing support has to be integrated to all layers of the Model Presentation Framework. The structure of layers may already give hints where such support is needed. The framework is designed so that implementing support for editing can be added later into it. This was reasonable for both clarity of the overall architecture and practical division of implementation into phases. Editing support layer is the most complex layer in the framework as the functionality cannot be totally separate from another layers.

A convenient way to study the editing support is to go from the top to the bottom layer by layer. Editing operations are designed for table component, although a similar approach is possible for trees. Many user interfaces contain windows and dialogs, which have multiple separate components for editing attributes individually. This type of modifications are not supported for the reason that you basically have to draw the line somewhere and the attribute pane is suitable for doing similar editing operations inside a single table component.

The Swing table component uses external components for rendering the contents. Swing defines a *CellRenderer* interface and some default implementations for that [Java2]. The architecture also allows you to make customised cell renderer classes [Java2]. Cell editing in tables and trees works similarly. The developer can use the default implementation of *CellEditor* interface or create a new cell editor class, if more specialised functionality is required [Java2, JavaTutorial].

Basically Swing architecture offered two choices to implement the editing support for the Model Presentation Framework. The easier solution would have been to implement editing by using default cell editors that come with Swing. This would have meant that once the Model Presentation Framework passes modified attributes to the application model classes, they always come in as strings without any specific type information. The application model classes should then implement the logic that interprets values correctly. A more complicated solution was to pass the attributes to the Model Presentation Framework after editing, so that the type information is still in place. To do this an attribute type aware cell editor was needed. This solution causes more work in the framework side and makes the implementation more complex, but a thing to remember is that the framework should reduce the work of an application developer, instead of turning it into another form. That is why the latter solution was chosen.

The most important part in the editing support layer is the class implementing cell editor functionality for the table component. It is required to handle every attribute type that exists in the Model Presentation Framework. The adapter class is telling table components whether some cell is editable or not. A mapping to the *editable* attribute declared in the *Attribute* class does this. If the attribute is editable the table will invoke the *TableCellEditor* [Java2] instance that is assigned for the table. Therefore the most convenient solution is to define a single editor class that is able to handle any parameter it receives. The default implementation in Swing is a good starting point as the existing functionality can be used for editing string attributes. Numeric values, booleans and enumerated types are a bit more problematic and specialised components must be assigned for them. The Model Presentation Framework utilises extended text field components from an internal Java platform [Platypus, 1999]. This design in the editing support layer takes a detailed responsibility of editing different types of attributes. As explained earlier in section 5.3.1, the attributes themselves may contain additional information and the customised cell editor classes can define their behaviour according to the additional attribute information. At the moment there is no design of such component in the customised user interface components layer that needs a special editing support, but existing components make it easier to implement editing functionality. When customised table component is used in the user interface, it can assign the customised cell editor component for itself by default. Therefore it is recommended for an application developer to deploy the user interface components provided by the Model Presentation Framework as much as possible. Doing that hides some implementation details and reduces the amount of user interface related code in the application.

In the user interface component adaptation layer, some adapter classes declare methods for setting the values. Their interface is already defined by Swing API [Java2], and they must be simply implemented to be compliant with the structure of the Model Presentation Framework.

The model object encapsulation layer is only for accessing the actual model objects behind it. Therefore there is no need to modify it to implement editing functionality. This is because the layer the adapters actually need to access is the model interface layer. As the application developer should implement the application model by inheriting and refining the classes in the model interface, it is his responsibility to build the editing support to them. Depending on which class or interface from the model interface layer is extended, the amount of methods that must be overwritten varies.

The application model class implementations have a very tight control of the things that are done in the user interface. Based on the attribute information retrieved by the application model classes the Model Presentation Framework is able to determine whether the attribute can be edited or not. If the attribute is editable the editing support layer selects the correct editor component according to the attribute type information. Once the user has made some changes in the user interface, the information about them is automatically passed to the model object in question. At this point the application has all the freedom to either accept the change or discard it. Naturally, if the change made by the user is discarded, the user should be informed about it. After the change has or has not been accepted the view component in the user interface is able to update itself automatically.

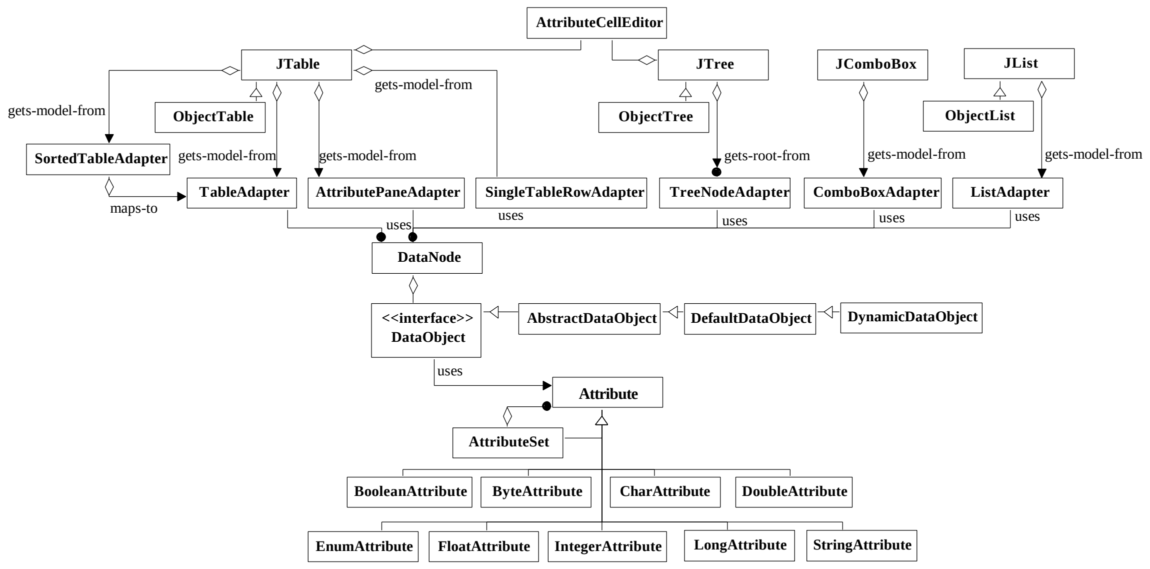


Figure 5-6. The overall class diagram of the Model Presentation Framework

When all the layers are combined into one diagram in Figure 5-6, we will notice a very clear separation between the presentation and actual data. The *DataNode* class defines the access point to the model connecting the two parts. The adapter and editor classes are able to call and use classes below the *DataNode,* but the view of the world they are seeing is always retrieved through the *DataNode*. This kind of separation is necessary in order to keep the architecture of the applications clear. When a common framework implements this separation, there should be no need to redesign it for every new application.

6. Utilising the Model Presentation Framework

The description of classes and object model gives a picture of the architecture, but it does not tell very well how the framework works in practise. This section explains how the Model Presentation Framework performs some operations and presents architecture of a simple test program that uses the framework.

6.1. Roles of the Model Presentation Framework in Applications

In principle, using lower layers of the Model Presentation Framework does not force developers to use higher layers. In this case, it can be argued whether there is any point in not using them. If the application is meant to display the data and allow user to modify it, there are very good reasons for building the functionality on top of the Model Presentation Framework services. However, some exceptional cases can be found. For instance if a very special approach in user interface design is applied, there surely will not be any support from the Model Presentation Framework in the form of user interface components. There can also be needs to strongly separate viewing and editing operations in the application. In such case the Model Presentation Framework could be used only for viewing purposes and editing functionality would be implemented separately. There are no restrictions for doing so, and as we will see later, a typical way of implementing add or delete operations is very similar to this approach.

The attribute encapsulation layer is a completely separate entity and can be used alone, whether there is any user interface or not. So, in some cases only parts of the Model Presentation Framework can be utilised.

6.2. Displaying the Data

When a client application with user interface is started, it does not necessarily need to have the whole model data in memory after start-up. Depending on user actions it may retrieve those parts of the model that are needed. This is why data nodes and objects in the Model Presentation Framework can be expanded. When the framework comes to a situation where internal data of a new object needs to be accessed for the first time, it tries to expand that object before passing it to the user interface. This ensures that all data, including children of the object are accessible.

The following sequence describes a typical flow of control when the controller initialises the model and the view to display the data:

1. Controller initialises the model.

2. Controller tells the view to show the model object by encapsulating the model object inside a *DataNode* instance and passing it to the view. The view creates the correct adapter for the UI component and puts the data node into adapter as a root object.
3. The view component decides to expand the displayed object. To do this it calls the component adapter.
4. User interface component adapter calls *expand* method of the *DataNode* instance. Inside the framework this causes an *expand* method call to the framework derived model object in the application.
5. When expand is done, the user interface component adapter has enough information to show object attributes or the children of the object. The user interaction may now begin.

When we study these steps we will see that only the first two steps are at the responsibility of the application. The rest of the sequence happens inside the framework itself. This is because the logic for expanding the object is quite generic and simple and can therefore have a default implementation. If the application needs to expand the object for some other reason there are no restrictions for doing it. The view component displaying the object will not notice that the object has been expanded before it needs to access the object in order to display the expanded data.

6.3. Modification through the Model Presentation Framework

The most simple editing operation for the Model Presentation Framework is a modification of an existing object. The domain specific model classes in the application must be able to handle modifications that are passed to them by the Model Presentation Framework. In addition the data in the user interface must be presented in a component that allows user to do editing operations. When these requirements are met, the rest of the editing is automatic. Figure 6-1 illustrates a control flow of a single modification in MVC++ application with the Model Presentation Framework.

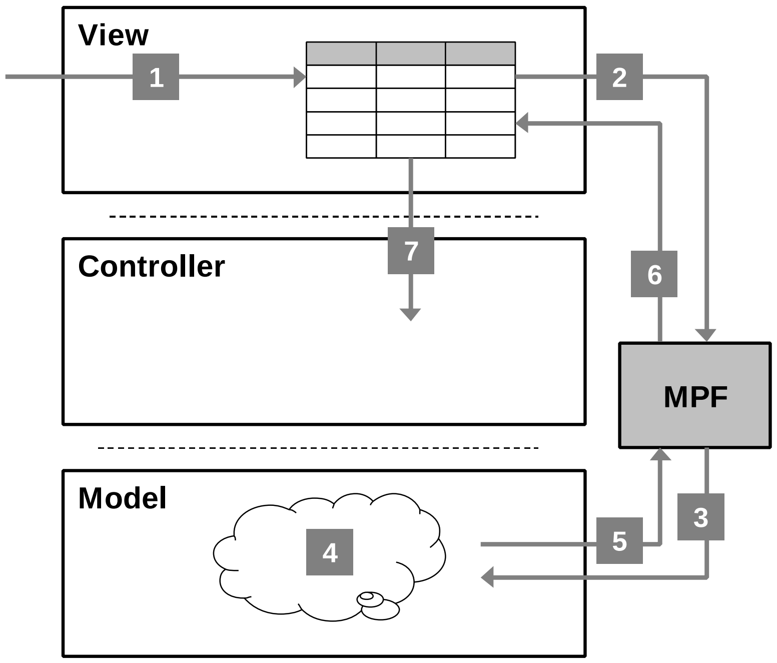


Figure 6-1. Modification going through the Model Presentation Framework.

1. User makes modification in the view part. In this example we assume that change was made in a table component.
2. View informs the Model Presentation Framework about the change in object information. With table component this means that method *setValueAt* in *TableAdapter* class gets called.
3. *TableAdapter* delegates changes to the framework compliant application model object. First it finds the correct *DataNode* instance and then retrieves the *DataObject* stored in it. Next the *TableAdapter* initialises an *AttributeSet* instance and puts an attribute that describes the change in it. Finally *TableAdapter* calls *DataObject* method *setAttributes* giving the constructed *AttributeSet* instance as parameter.
4. Model object analyses the change and applies it to its parameters. At this point the model object may also discard the changes if they are not valid.
5. Model object returns the results back to the Model Presentation Framework (OK / Not OK).
6. The Model Presentation Framework passes the results back to the view. After this step the view should refresh itself, so that the data is synchronised with the actual model data.

Optionally the view may inform the controller about modifications and the controller can then do some additional operations.

6.4. Creating new Objects

Creation of new objects requires much more information than changing the attributes of an existing one. The Model Presentation Framework does not know the exact type of the object or how the object should be created and initialised. Therefore it is better to execute the creation operation in traditional MVC++ manner through the controller to the model. The purist interpretation of MVC++ requires that it is the model layer that actually creates the new object. The role of the Model Presentation Framework in creation of the new object is only updating the changes to the user interface. Depending on the design and logic of the application, the updating may need some additional information or logic may be necessary in order to update the user interface correctly.

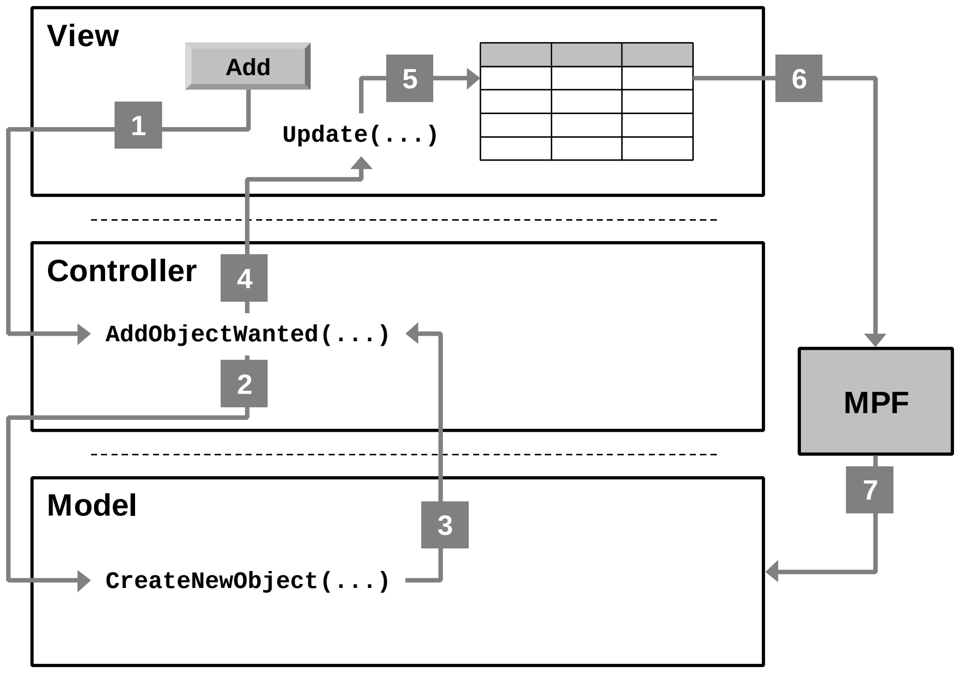


Figure 6-2. Creating a new object in the MVC++ manner and notifying framework about the changes.

Figure 6-2 illustrates the object creation procedure. The steps performed go as follows:

1. Add operation is invoked in the user interface, controller is called.
2. Controller handles the request and tells the model that new object should be created.
3. The object is created and the flow of control returns to the controller.
4. Controller asks the view to update itself.
5. View asks the correct UI component or components to update themselves.

6. View component calls the Model Presentation Framework to retrieve the changed data.
7. The Model Presentation Framework retrieves the updated data from the model classes and adapts it for the UI component that was asking it.

Note: Step 4 can also be an OK/Not OK result to the view part of the application. If so, the view must decide by itself whether any UI components need to be updated.

The same circumstances apply to object removal. Creation and removal require a fair amount of application logic and are therefore the weakest part of the Model Presentation Framework. While some demo applications were created on top of the Model Presentation Framework, it was clearly seen that it is very hard to understand how to perform these operations and therefore they need an extensive support in the framework documentation.

6.5. Sample Application

One of the test programs in the Model Presentation Framework demonstrates a simple example of an MVC++ application using the Model Presentation Framework. The test program has a simple MVC++ structure and working functionality in only 150 lines of code. It is therefore very suitable example for training material to demonstrate both application structure and hierarchical model structures. The test program simply models a hierarchy of integers being equal or greater than 0. At the first level in the hierarchy we find integers from 0 to 9. Under '2' we find integers from 20 to 29 and so on. The screenshot in figure 6-3 illustrates this. Theoretically there is no limitation for the number of integers presented in the tree.

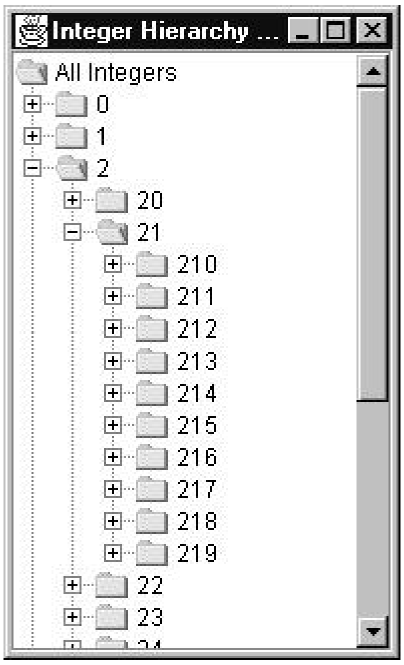


Figure 6-3. Test program for hierarchy of integers.

Similar functionality without the Model Presentation Framework was also implemented for training purposes. That also took about 150 lines of code, but a few issues should be taken into account. First, the model part in the implementation was user interface dependent and only compatible with the *JTree* component. If the model data should be presented in any other user interface component the model implementation should be divided into two classes and a new model class for another user interface component would be needed. Second, this example does not manage any parameters. If parameter handling is needed the amount of code in controller and view layers of regular implementation would rise significantly. With the Model Presentation Framework such changes would affect only the model layer and using the same data in other user interface components than *JTree* would require only a few additional lines of code.

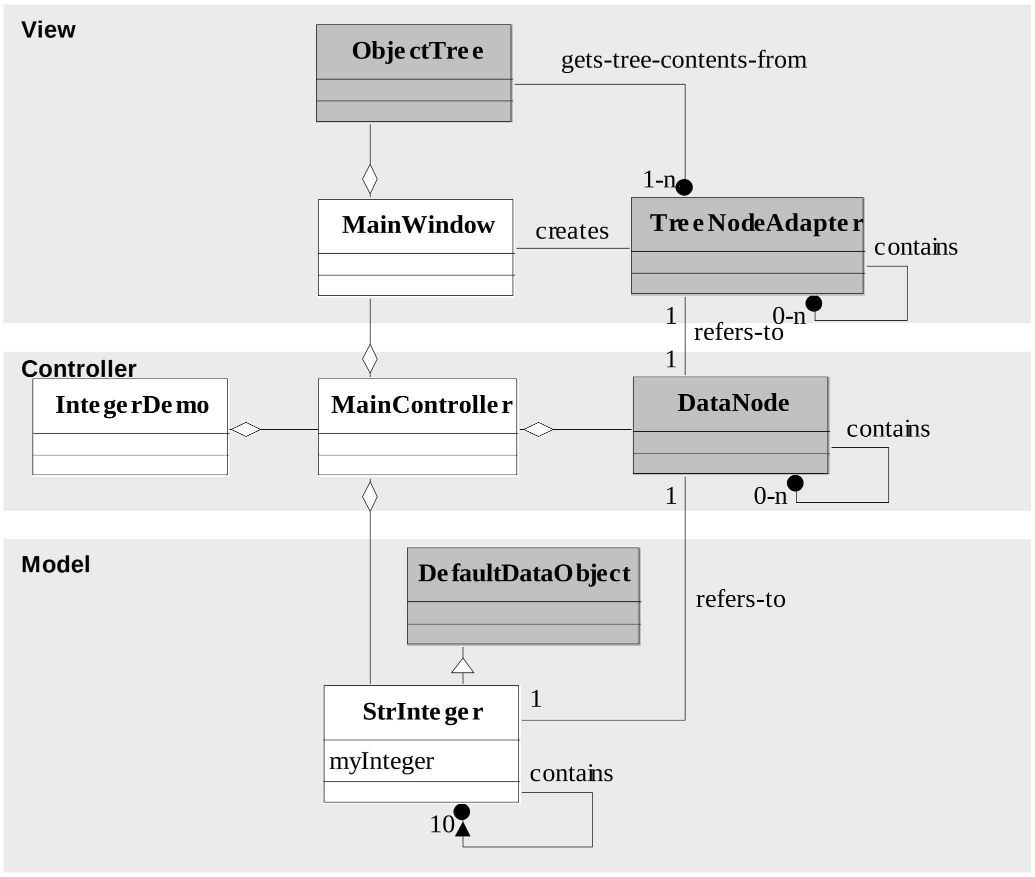


Figure 6-4. Test program class diagram.

The class diagram of the test program is presented in figure 6-4. Model Presentation Framework classes are drawn in grey colour. The only task of *IntegerDemo* class is to start the *MainController*, which then creates the *MainView* and the model. The model in this case means one instance of *StrInteger* class that serves as a root object for the data. When the *MainController* in the test program passes the root object to the view, it needs to be encapsulated inside *DataNode* instance. Once the view gets the *DataNode* object to be used as root of the hierarchy, it creates a suitable adapter for it. In this case *TreeNodeAdapter* is used and set as a model for the *ObjectTree* component coming from the Model Presentation Framework. At this point there is a connection to the actual model object through the framework classes.

In the next phase the framework classes get more active, because the tree component needs to have more data. First this is because the tree needs to know whether the visible nodes are leaf nodes or not. Later, also the user interaction causes more calls to the model layer. As the test program has only browsing functionality, all the calls to the model can be done through the Model Presentation Framework classes. This example is, of course, very simplified and a bit too optimistic, when compared to real application. However, it is able to give the general idea how the application controller layer is bypassed by utilising the framework classes.

6.6. Filtering and Sorting

Large sets of data are something that many applications have to cope with. Application users in turn want to work effectively and often making changes one by one and navigating to the next spot to be changed is far too slow. This brings up the issues of finding the desired data and targeting the modifications to the correct subset of data.

One way to access the correct set of data is to filter it so that only items that match with certain conditions are visible. It is not so easy to determine these conditions as they depend on the application data. Conditions that application may need are, for example, exact values, value ranges, strings matching to regular expression, and negations of all those. Designing a complete filtering engine that can handle almost any thinkable condition is not reasonable on the framework level. Therefore, filtering of data with the Model Presentation Framework is one of the key issues that must be explained with good examples in the framework documentation.

For a small convenience *AttributeSet* class offers a simple filtering support with two methods. Method *isSubsetOf* can be used to determine whether the *AttributeSet* instance has subset of identical parameters when compared to another *AttributeSet*. This means that attribute types and their values must be the same. Method *isDescriptionOf* works the same way, but allows the attribute in the set to be of different types. Attributes are considered identical if their string presentations match each other. Using these methods together with tailored filtering for some parameters it is possible to build very powerful filter capabilities into application.

In principle, sorting of data is a completely generic operation and can therefore be supported easily. The Model Presentation Framework provides basic services for sorting data in a table component. In the current design sorting is based on a single sorting key at a time. The sorting key can be any column of the table. Sorting is however *stable*, meaning that if objects are considered to be equal by the current sorting key, the relative order of the objects will not change. Basically this allows sorting with multiple keys, but this requires sequential sort operations to be run. Sorting the actual model data would be more efficient choice if it would be possible to implement a generic sorting using multiple keys. The implementation would be closer to the actual data, which decreases the overhead caused by object-oriented architecture of the framework. Both sorting solutions have their advantages as sorting on the table allows direct manipulation. Sorting using multiple keys would not be so easy to use, but it could prove to be more efficient with large-scale applications.

6.7. Status of the Model Presentation Framework

The existing implementation of the Model Presentation Framework was mainly a result of prototyping. A rough design was made along the way, but apart from the feasibility study [Vuorenmaa, 1999], no formal design documents were finished at the prototyping phase. The implementation follows very closely the design presented in this thesis, although some features have not been implemented yet. For instance editing support is not completed for all attribute types. All designed features have been studied to the level where it is sure that they are possible to implement.

The implementation had an iterative cycle controlled by several demo applications outlining real applications and smaller programs for testing the separate features of the framework. During the implementation a few large-scale changes were done to make the framework more feasible and easier to use. Undoubtedly more of that work lies ahead.

In the future the existing implementation must be moved on top of internal Java platform, Platypus that offers trace and error logging services and some user interface components [Platypus, 1999]. This was one of the reasons why some functionality was intentionally left unimplemented in the prototyping phase.

The existing implementation is able to work together with the MVC++ architecture, but it does not rely on it. This was an intentional design decision. The purpose of the Model Presentation Framework is to define a strict and fixed separation of the model data and its presentation for the applications. On top of that it offers a generic abstraction that allow the data to be displayed in multiple user interface components, without additional conversion in the view layer of the application. Ability to co-exist with the MVC++ was a requirement, but restricting the implementation only to work with the MVC++ architecture was not. So being, it should be possible to combine the Model Presentation Framework with other application architectures as well.

7. Applicability of MPF

This chapter addresses the feasibility and applicability of the Model Presentation Framework. After a short overview, we cover some experiences collected during the framework development and experimental prototyping. Then some Model Presentation Framework assets and downsides are summed up. After this two other frameworks are shortly introduced and compared to the Model Presentation Framework.

7.1. Achieved Benefits

The second chapter brought up the problems of the MVC++ architecture. They were large amount of classes in the implementation and the fact that simple changes, for example supporting new parameters, cause changes to several classes in all layers of the MVC++ architecture.

The Model Presentation Framework does not directly reduce the number of classes in the implementation. All layers of the MVC++ are still there and they need to be implemented as before. This is a consequence of the MVC++ architecture itself and designing a new framework on top of the architecture does not remove it. However, the Model Presentation Framework simplifies the structure of a typical MVC++ application by defining parts of the implementation.

In an application that uses the Model Presentation Framework the model layer must follow certain guidelines that make the model implementation a bit more difficult. This is a small cost when compared to the savings on controller and view layers. The controller and view layers do not have to know so much about the structure of the data. The view layer can use component adapters and possibly user interface components supported by the Model Presentation Framework. Because of this the view layer does not know what kind of data it has and does not need to know how it is modified. Navigation in model object hierarchies happens through the Model Presentation Framework, provided that used user interface component supports navigation. Modifications also go through the framework directly to the application model objects. When compared to creation and removal of objects, modifications work like magic. Although the user interface components do not know much about the attributes they are displaying, the editor components are type aware and access the additional attribute information. Editing functionality in the Model Presentation Framework is therefore able to assign a dedicated editor component for each attribute type.

As we learned before object creations and removals are at the responsibility of the application. This means that they are passed through the MVC++ layers of the application and therefore have to be designed for each application. The design can in turn utilise the framework services if this is considered as feasible.

An implication of the Model Presentation Framework at its best is that changes in domain object model cause changes only to the model layer of the application. A new object parameter means changes in the corresponding model class and naturally if support for new model object is needed a new model class has to be designed. Support for object creation can sometimes spoil the ideal case, and that may be one issue that has to be studied in the future. To sum it up the Model Presentation Framework does not reduce the number of classes in an MVC++ application, but makes their implementation much simpler by taking responsibility of defining generic functionality of the typical user interface application.

The Model Presentation Framework implements functionality that is reusable in application. This can not be achieved by using a simple user interface library. This should speed up the development and let the developer to concentrate more on the application specific functionality and problems, instead of implementing the basic logic of the user interface. When all data handling in the application code is done at the model layer, also the reusability of application code increases. With small changes the implementation of view and controller layers may be reused in another application. Of course this also requires a reuse of the previous user interface concept.

7.2. Drawbacks

The framework does not support adding and deleting objects, and thus the application must control these operations in a traditional MVC++ manner. The framework can offer some support for updating the user interface after those operations, but the application must still have the overall control of the operations. Defining a generic logic for adding and deleting objects in the future seems difficult, as there can be various dependencies in the application data that affect these operations.

Many applications handle dates and times in some manner. Naturally application can convert date or time into a *StringAttribute* in order to display it, but the framework does not have components for date and time input. This is definitely an issue that has to be studied further in the future. Needs to localise dates and times make the problem more complicated.

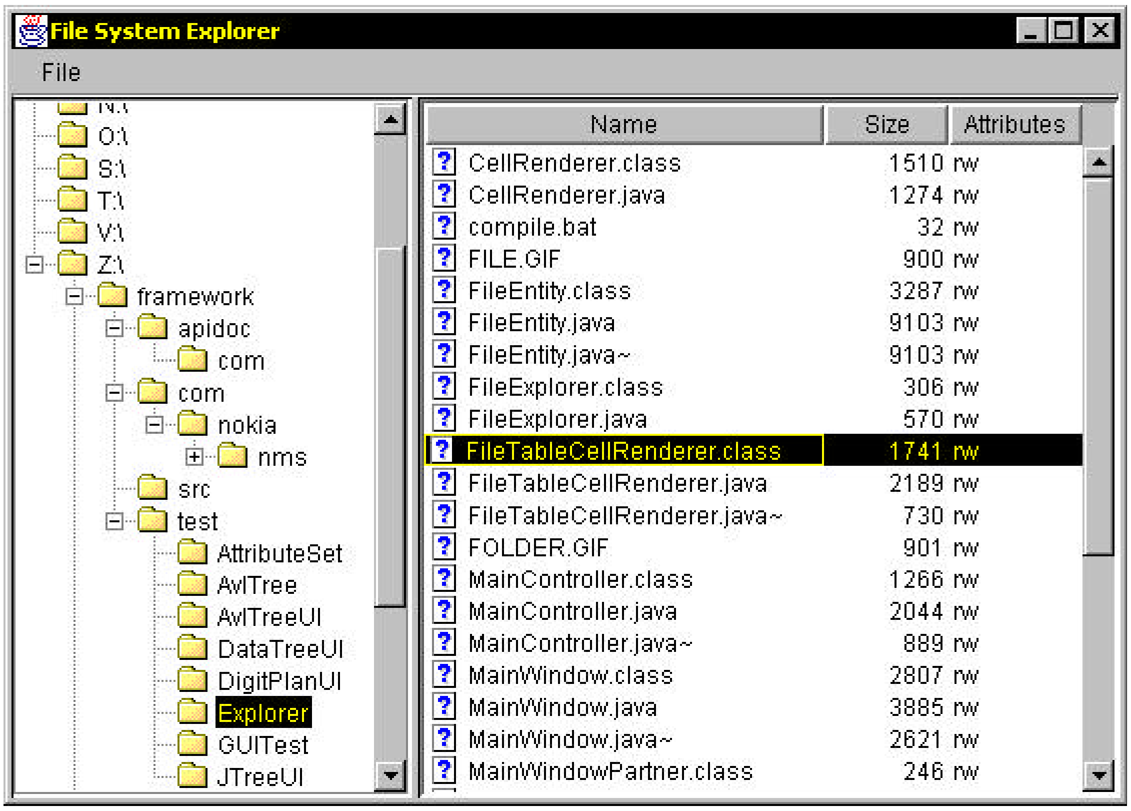
Strong object-orientation of the framework makes it modular and generally the functionality is divided into very small pieces. This hopefully makes the maintenance work easier, but it also causes the decrease of performance. To increase the performance some optimisation was done already at the prototyping phase. Mainly they were targeted to increase response times in a table component. *TableAdapter* class implements a short-term caching to reduce the number of calls to the application model classes. When attributes of a certain table row are asked for the first time, the *TableAdapter* accesses the application model and stores the values. As long as the row is not changed, the cached values are used and the application model is not accessed. Another need for optimisation was discovered when sorting of the table rows by column values was tested. This revealed both performance and memory problems. Once again, caching the values inside an internal *Comparator* [Java2] class inside the framework clearly improved the performance.

It was also discovered that the framework architecture allows an alternative way for implementing the application model classes. The model interface has the *getAttributes()* method that returns an instance of an *AttributeSet*. A traditional way to implement *getAttributes()* would be to construct this set from primitive type attributes stored in the model class and return it. Another possibility would be not to use primitive attribute instances at all. The model class could always keep the attributes inside the *AttributeSet*. This would consume more memory, but the attribute access would be faster, because there is no conversion step before the framework can access the attributes. The larger memory consumption of course sets lower limits to the amount of objects kept in memory. The practical knowledge in this area is still a bit vague, but it could be proposed that classes that have a large amount of instances should store their attributes as primitive types and construct the attribute sets only when they are needed. For the classes that are not designed to have so many object instances, storing the attributes in internal *AttributeSet* instance could be more useful. Mainly this is an optimisation problem between the memory usage and speed. Unfortunately the application developer has to solve it.

When these drawbacks and the general complexity of frameworks and utilisation are taken into account, it is clear that learning to use the Model Presentation Framework is not simple. The documentation must cover wide range of issues from basic learning steps to design tips in order to help the application developer work efficiently. The difficulty of tracking errors between the application and the framework has already been noticed in practise. This indicates that in wider use there should also be a contact person with some expertise for fast assistance.

7.3. Existing Usage Experiences

Several demos were built during the prototyping of the Model Presentation Framework. They indicated that building demos on top of the Model Presentation Framework is more prototyping the structure of data than with traditional user interface demos. This is because there must be a framework compatible model implementation before anything reasonable can be displayed. This creates a little more work for the first prototype, but it can also be turned into asset. Once the prototype model for an application domain has been created, making another demo with a different user interface concept is fairly easy and fast, because multiple user interface components are able to present the contents of the same model. An existing prototype may also have a very generic user interface implementation that may be able to serve as a skeleton for another demo user interface on a different application domain. In an ideal case, only the model classes need to be redesigned, but early experiences have shown that some adjustments are usually necessary.



Name	Size	Attributes
CellRenderer.class	1510	rw
CellRenderer.java	1274	rw
compile.bat	32	rw
FILE.GIF	900	rw
FileEntity.class	3287	rw
FileEntity.java	9103	rw
FileEntity.java~	9103	rw
FileExplorer.class	306	rw
FileExplorer.java	570	rw
FileTableCellRenderer.class	1741	rw
FileTableCellRenderer.java	2189	rw
FileTableCellRenderer.java~	730	rw
FOLDER.GIF	901	rw
MainController.class	1266	rw
MainController.java	2044	rw
MainController.java~	889	rw
MainWindow.class	2807	rw
MainWindow.java	3885	rw
MainWindow.java~	2621	rw
MainWindowPartner.class	246	rw

Figure 7-1. Test program mimicking Windows Explorer.

One of the Model Presentation Framework test programs is simple a file system explorer imitating the well-known Windows explorer as illustrated in figure 7-1. The program includes browsing functionality only, not any manipulation of files. The main window has a tree on the left side, displaying the drives and folders in the file system. On the right side lies a table that displays files and their sizes and attributes. This program has MVC++

structure working together with the Model Presentation Framework. This functionality is implemented in less than thousand lines of code. This test program was built in parallel with the framework functionality, and therefore it is hard give any reliable estimate of the actual development time.

In telecommunications network switch, there are many configuration objects that only have a numeric ID to identification. A simple user interface prototype was constructed to demonstrate the application that would be used the give names to one numerically identified object type. The prototype presents a hierarchy of configuration objects, which is based on some parameter values the objects have. The user is able to navigate in this hierarchy and add or modify the names for the objects. Based on the skeleton taken from the earlier file system explorer program this prototype was put together in four hours. When the domain area expert wanted to make some adjustments to model, the additional work took one more hour. As the logic and the support for editing name attributes came from the Model Presentation Framework, a working basic functionality was implemented in roughly 900 lines of code. The user interface initialisation took about 400 lines and the remaining 500 lines were the implementation of model class hierarchy and general MVC++ architecture.

The view and controller layer implementations of the file system explorer and this prototype are nearly similar. This case clearly demonstrated that sometimes redefining the model layer is almost enough to build a new prototype for different purpose.

As test programs and some prototypes were built on top of the Model Presentation Framework, it also became clear that knowing the internals of framework was something almost obligatory. Even though the framework became more and more reliable, locating errors was sometimes hard. Frequent switching of control between the framework and application code caused a few confusing situations where tracking the error was especially hard.

7.4. Comparison to other Frameworks

7.4.1. ET++

ET++ is an object-oriented application framework for building user interfaces with C++ for Unix environment [Weinand and Gamma, 1994]. With many sophisticated features [Pree, 1995] and extensive domain area and about three hundred classes [Weinand and Gamma, 1994], it is not possible to cover the finer details of ET++ here. ET++ is divided into layers, each having their own roles. The *Toolkit* layer has the most primitive parts of the framework, including the class *Object*, which a base class for the most of the ET++ classes

[Weinand and Gamma, 1994]. *Data Structures* layer implements many general classes, such as containers, that are also utilised in higher layers of the framework [Weinand and Gamma, 1994]. The *View System* layer defines characteristics of graphical objects and *User Interface Toolkit* provides graphical user interface components. The *Framework* and *Application Framework* layers define many higher level frameworks for building parts of the application or the general logic as a starting point for the application [Weinand and Gamma, 1994]. Other layers provide operating system level abstractions as well as services for investigating the internals of ET++ application while they are running [Weinand and Gamma, 1994].

ET++ is able to load and link classes dynamically to a running application by utilising virtual functions [Weinand and Gamma, 1994]. The *Object* base class provides an object input/output with streams or to be used with clipboard as well as change notification mechanism [Weinand and Gamma, 1994]. Nowadays similar features are becoming more popular in standard development tools and for example object serialisation and listeners can be found in Java API [Java2].

For graphical objects the *View System* layer in ET++ defines a *VObject* class that implements basic functionality of visual objects [Weinand and Gamma, 1994]. That class is the basis for visual objects and user interface components [Weinand and Gamma, 1994]. High level features of ET++ include for example building blocks for text editors with support for several text formats, grid views for tabular data and tree views for data hierarchies [Weinand and Gamma, 1994].

ET++ really is a large library of services and a collection of frameworks. It is suitable for building for example drawing editors, spreadsheet applications word processors [Weinand and Gamma, 1994] or hypertext systems [Pree, 1995]. Isolation from the system underneath makes ET++ applications easily portable between supported systems [Weinand and Gamma, 1994]. In short, ET++ serves as an excellent example of successful framework design. It helps the designer in many levels of the application, and when compared to the Model Presentation Framework, ET++ is far more extensive and advanced.

7.4.2. HotDraw

HotDraw is a framework for designing graphical editors [HotDraw]. It is versatile in many typical aspects of object oriented drawing tools [Johnson 1992] and has been used to implement for instance a case tool and a HyperCard clone [HotDraw]. Drawing model is two-dimensional and objects in a drawing may react to user commands, they can be animated as well as constraints can be specified between them [Johnson 1992].

In application utilising HotDraw, all elements in drawings are subclasses of *Figure* [Johnson 1992]. The framework contains ready-to-use subclasses of *Figure* that can be used when applicable and the application developer can specialise the *Figure* or any of its subclasses to implement his own drawing elements [Johnson 1992]. *CompositeFigure* is a subclass of *Figure* that consists of a set of other figures [Johnson 1992]. Therefore it is very usable when the customised figures need to be created [Johnson 1992]. The name *CompositeFigure* indicates that Composite design pattern [Gamma *et al.*, 1995] is applied in the design of figure class hierarchy. This solution is quite similar to the attribute encapsulation layer in the Model Presentation Framework (see section 5.3.1). Both of these cases have solved a problem of defining a container class that can contain references to other container classes. Using the Composite design pattern is a very natural choice here [Gamma *et al.*, 1995].

More advanced features in HotDraw are constraints, handles, tools and animations [Johnson 1992]. Constraint objects can be used to define dependencies between objects [Johnson 1992]. Constraints can change some attributes belonging to figures [Johnson 1992]. For example a constraint can keep the line connected to some drawing figure when the figure is moved [Johnson 1992]. Handle is small visible control that is attached to a figure [Johnson 1992]. A typical usage of handle is for example resizing of figure, but in principle a handle can be used to do any of operation to figure [Johnson 1992]. Tools are simply operations that are visualised in a tool palette [Johnson 1992]. This concept is used practically in any drawing editor. In HotDraw a tool can be used for example to go into a different drawing mode, manipulate a figure, or to perform some operation for entire drawing [Johnson 1992]. Animation features of HotDraw can be used to implement user independent actions in a drawing [Johnson 1992]. They can be for example automatic periodical adjustments to the drawing, executed by the editor application itself [Johnson 1992]. All things considered, HotDraw forms a very good customizable framework for editing and handling two-dimensional drawings in applications that utilize it.

What HotDraw does for graphical editors, the Model Presentation Framework does for model of the MVC++ client applications. It generalizes a part of basic MVC++ interaction towards model and using that it provides default functionality for its domain. The Model Presentation Framework is not able to provide working default application like HotDraw. This is mainly because of different framework domains. The Model Presentation Framework can be utilized for any kind of application and it can only define an interface towards model data, not the user interface the framework user wants to make.

In its own domain HotDraw goes further than the Model Presentation Framework. It defines many hot spots where the framework is flexible and customizable. The Model Presentation Framework practically has only one hot spot, which is the extensible model interface layer.

7.4.3. Comparison Summary

ET++ and HotDraw are advanced frameworks that offer a lot of functionality, and have clearly reached a level of maturity. ET++ is a framework for building user interface and HotDraw is more focused on building graphical editors. From that aspect, their domain is more fixed than the Model Presentation Framework domain. The Model Presentation Framework extends the MVC++ architecture and helps the developer to bring the application data to the user interface and handle it there. That task can be done by a fairly small framework, which the Model Presentation Framework is currently. Lower level framework layers provide very generic functionality, but on the other hand one must agree that the set of supported user interface components at the moment is quite limited.

The Model Presentation Framework will probably need some extensions to be more powerful in the future. It is not that easy to anticipate what layer is the first one that needs to be enhanced. The attribute encapsulation could have support for more attribute types and the model interface layer could support different operations, like sorting and searching model object hierarchies. Possibly, support for new user interface components is needed as well. These steps take the Model Presentation Framework more towards a black-box framework with ready-made components. Very likely ET++ and HotDraw have taken similar step in their process of growth. Due to its restricted domain alongside MVC++, the Model Presentation Framework will probably not ever grow to be a very large framework. Hopefully, it will be accepted as a complement for the current MVC++ architecture.

8. Conclusion

The purpose of the Model-View-Controller architecture is to divide the application functionality into logical and manageable parts. I have presented two weaknesses of the MVC++ architecture. Handling complicated model data, both classes and their parameters, requires lot of effort in all layers of the MVC++ application. Also the number of classes is fairly large in MVC++ applications, especially if the user interface consists of several windows and dialogs. It is hard to reduce the number of classes, as it is a direct implication of the MVC++ architecture. Therefore, I concentrated on finding an alternative way for handling the model data in MVC++ applications.

To find a generic solution to the problem, I studied the principles of object-oriented frameworks and their relationship to class libraries and design patterns. I also investigated the issues of object-oriented framework design, design tools, implementation, testing, documentation and maintenance. It seemed that object-oriented framework could provide a solution for more generic handling of model data in MVC++ applications.

Next, I presented the design of the Model Presentation Framework that provides an automatic and generic presentation and modification of model data. Although the Model Presentation Framework design was made for MVC++ applications, it does not depend on the MVC++ and should be usable in other application architectures as well. The framework functionality was demonstrated with operation descriptions and the class diagram of a simple test application.

The current state of the framework and test programs that have been built using it, indicate that the Model Presentation Framework can partly solve the problems of handling the data in the view layer of the MVC++ applications. Presentation is already supported in many user interface components, and editing the object attributes is possible in table based components. Application using the Model Presentation Framework has total control over the model classes, their attribute types and modifications, as long as the classes implement an interface defined by the framework. Unfortunately, problems in creation and deletion of model objects were not solved, and applications using the Model Presentation Framework still have to use normal MVC++ methods to implement these operations. In spite of that, the view and controller layers in Model Presentation Framework based application are smaller, because most of the object parameter handling can be performed inside the framework.

Other experiences with the Model Presentation Framework emphasise the importance of good design and documentation. Common problems in building and using frameworks were also faced in this case. Building prototypes on top of the framework made clear that the Model Presentation Framework is very data dependent. Even for prototypes, the structure data must be designed in order to get the framework to display something. In application development this can be turned into asset, but in prototyping it usually means just more work. Comparisons to ET++ and HotDraw frameworks showed that the Model Presentation Framework is still in quite an early phase of development and careful research and design is needed to find out how it should be developed in the future.

It has been shown that the handling of model data and parameters in MVC++ applications can be partly automated with a framework that extends the MVC++ architecture. At the moment it is difficult to say whether the community of developers will accept such a solution and what form it will eventually take.

References

[Bonnet, 1999] Stéphane Bonnet, Java MVC++ Framework for NMS GUI Applications, Master of science thesis, Tampere University of Technology, Dept. of Information Technology, 1999.

[Brugali *et al.*, 1997] Davide Brugali, Giuseppe Menga, Amund Aarsten, The Framework Life Span, *Communications of the ACM, Vol. 40, No 10,* October 1997.

[Fayad and Schmidt, 1997] Mohamed E. Fayad, Douglas C. Schmidt, Object-Oriented Application Frameworks, *Communications of the ACM, Vol. 40, No 10*, October 1997.

[Gamma *et al.*, 1995] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software,* Addison-Wesley Publishing Company, 1995.

[Hakala *et al.*, 1999] Markku Hakala, Juha Hautamäki, Jyrki Tuomi, Antti Viljamaa, Jukka Viljamaa, Kai Koskimies, Juha Paakki, Managing Object-Oriented Frameworks with Specialization Templates. April, 1999. Available at *http://www.cs.helsinki.fi/group/fred/reports/ecoop99.doc*, Last checked: December 21st, 1999.

[Holub, 1999a] Allen Holub, *Building user interfaces for object-oriented systems, Part 1*, JavaWorld, July 1999. Available at *http://www.javaworld.com/jw-07-1999/jw-07-toolbox.html*, Last checked: December 29th, 1999.

[Holub, 1999b] Allen Holub, *Building user interfaces for object-oriented systems, Part 2: The visual-proxy architecture,* JavaWorld, September 1999. Available at *http://www.javaworld.com/javaworld/jw-09-1999/jw-09-toolbox.html,* Last checked: December 29th, 1999.

[HotDraw] HotDraw Home Page Available at *http://st-www.cs.uiuc.edu/users/brant/HotDraw/HotDraw.html*, Last checked: 22.1.2000.

[Jaaksi, 1994] Ari Jaaksi, Implementing MVC Applications in NTC NMS, 1994.

[Jaaksi, 1995] Ari Jaaksi, Implementing Interactive Applications in C++, *Software Practice & Experience*, March 1995.

[Jaaksi *et al.*, 1999] Ari Jaaksi, Juha-Markus Aalto, Ari Aalto, and Kimmo Vättö, *Tried and True Object Development, Industry Proven Approaches with UML*, Cambridge University Press, 1999.

[Jacobson *et al.*, 1997] Ivar Jacobson, Martin Griss, Patrik Jonsson, *Software Reuse*, ACM Press, 1997.

[Java2] Java 2 SDK documentation. Available at *http://www.javasoft.com/products/jdk/1.2/docs/index.html.* Last checked: October 25th, 1999.

[JavaTutorial] The Java Tutorial, Available at *http://java.sun.com/docs/books/tutorial.* Last checked: November 3rd, 1999.

[Johnson and Foote, 1988] Ralph E. Johnson, Brian Foote, Designing Reusable Classes, *Journal of Object-Oriented Programming (JOOP),* June/July, 1988. Available at *ftp://st.cs.uiuc.edu/pub/papers/frameworks/designing-reusable-classes.ps.*

[Johnson and Russo, 1991] Ralph E. Johnson, Vincent F. Russo, Reusing Object-Oriented Design. *Technical Report UIUCDCS 91-1696,* University of Illinois, 1991. Available at *ftp://st.cs.uiuc.edu/pub/papers/frameworks/reusable-oo-design.ps.*

[Johnson 1992] Ralph E. Johnson, Documenting Frameworks Using Patterns, Proceedings of OOPSLA '92, Available at *ftp://st.cs.uiuc.edu/pub/patterns/papers/documenting-frameworks.ps,* Last checked January 30th, 2000.

[Johnson, 1997] Ralph E. Johnson, Frameworks = Components + Patterns, *Communications of the ACM, Vol. 40, No 10,* October 1997.

[Koskimies, 1997] Kai Koskimies, *Pieni oliokirja,* Suomen Atk-kustannus 1997.

[Koskimies and Mössenböck, 1995] Kai Koskimies, Hanspeter Mössenböck, Designing a Framework by Stepwise Generalization, *Proceedings of 5th European Software Engineering Conference,* Springer-Verlag, 1995. Available at *http://www.ssw.uni-linz.ac.at/Research/Papers/Moe95.html.* Last checked: January 18th, 2000.

[Krasner and Pope, 1988] Glenn E. Krasner, Stephen E. Pope, A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80, *Journal of Object-Oriented Programming (JOOP),* 1, 3, 1988.

[Lewis *et al.*, 1995] Ted Lewis, Glenn Andert, Paul Calder, Erich Gamma, Wolfgang Pree, Larry Rosenstein, Kurt Schmucker, André Weigand, John Vlissides, *Object Oriented Application Frameworks,* Manning Publications Co., 1995.

[Platypus, 1999] Platypus – Java GUI Platform, Designer's Guide. Nokia Internal Document.

[Pree, 1995] Wolfgang Pree, *Design Patterns for Object-Oriented Software Development,* Addison-Wesley Publishing Co., 1995.

[Taligent, 1993] Leveraging Object-Oriented Frameworks. A Taligent White Paper, 1993. Available at *http://www.ibm.com/Java/education/ooleveraging/index.html.*

[Taligent, 1994] Building Object-Oriented Frameworks. A Taligent White Paper, 1994. Available at *http://www.ibm.com/Java/education/oobuilding/index.html.*

[TFM, 1999] Traffic Flow Management, User's Guide, Nokia Internal Document.

[Viitanen, 1999] Discussions with Heli Viitanen, December 1999.

[Virtanen, 1999] Discussions with Jussi Virtanen, December 1999.

[Vuorenmaa, 1999] Markku Vuorenmaa, *Model Presentation Framework – Feasibility Study*, version 1.0, September 1999, Nokia Internal Document.

[Weinand and Gamma, 1994] Andre Weinand, Erich Gamma, ET++ - a Portable, Homogenous Class Library and Application Framework, *Proceedings of the UBILAB '94 Conference*, Zurich, 1994. Available at *http://www.ipd.hk-r.se/michaelm/fwpages/files/Wei94.ps.*