Chapter 32

Framework Theory

Learning Objectives

The learning objective of this chapter is primarily to define the terminology associated with the concept of object-oriented *frameworks*, as the implementation techniques have already been accounted for. Thus, the discussion will take a more theoretical standpoint.

32.1 Framework Definitions

If you look up the term "framework" in a dictionary you will find definitions like *a basic conceptual structure* or *a skeletal or structural frame.* If we look to the software engineering community, a number of authors have presented definitions of their own.

- A framework is: a) a reusable design of an application or subsystem b) represented by a set of abstract classes and the way objects in these classes collaborate (Fraser et al. 1997).
- A framework is a set of classes that embodies an abstract design for solutions to a family of related problems (Johnson and Foote 1988).
- A framework is a set of cooperating classes that make up a reusable design for a specific class of software (Gamma et al. 1995, p. 26).
- A framework is the skeleton of an application that can be customized by an application developer (Fayad et al. 1999a, p. 3).
- A framework defines a high-level language with which applications within a domain are created through specialization (Pree 1999, p. 379).
- A framework is an architectural pattern that provides an extensible template for applications within a domain (Booch et al. 1999, p. 383).

32.2 Framework Characteristics

If we look closer at these definitions they use different terminology and they have different perspectives. For instance, they use various terms, like "skeleton", "architectural pattern", "design", "high-level language", for basically the same underlying concept. Regarding perspective, Gamma et al. define frameworks in terms of what it *is* whereas Fayad et al. and Pree define it in terms of what is can be *used for*. However, they all try to express the characteristics that identify a piece of software as a framework as opposed to an application, a subsystem, an algorithm, or a library. These characteristics are:

- *Skeleton / design / high-level language / template*: that is, the framework delivers application behavior at a high level of abstraction. This is much in line with the general definition of a framework as *a basic conceptual structure.*
- *Application / class of software / within a domain*: that is, a framework provides behavior in a well-defined domain.
- *Cooperating / collaborating classes*: that is, a framework defines the protocol between a set of well-defined components/objects. To use the framework you have to understand these interaction patterns and must program in accordance with them.
- *Customize / abstract classes / reusable / specialize*: that is, a framework is flexible so that you can tailor it to a concrete context, as long as this context lies within the domain of the framework.
- *Classes / implementation / skeleton*: that is, a framework is reuse of working code as well as reuse of design.

Given these characteristics of a framework I can now state that what started as a pay station *application* for a single customer, Alphatown, has evolved into a configurable pay station *framework.* The pay station system fulfills all the characteristics mentioned above: it is a *skeleton* within a particular *domain*, namely pay stations, and consists of *collaborating classes*, some of which are *customized* for a particular product variant. And it comes with *implementation* and thus embodies both design as well as code reuse.

Exercise 32.1: Given these characteristics, determine whether Java Swing is a framework.

32.3 Types of Users and Developers

In normal software development, we classify stake holders as *developers* and *users*. Developers they, well, develop the software, hopefully listening to the opinion of the users which are the people that in the end will use the software to get their work done efficiently.

With regards to framework development, there are actually three types of stakeholders: *framework developers*, *application developers*, and *users*. The framework developers are the people that design and code the framework while the application developers are the programmers that customizes the framework to the particular needs of the users. Thus one can say that the application developers are actually the "users" of the framework. The application developers are the customers that framework developers must listen to in order to produce a good framework that will make them work efficiently.

Application developers on the other hand have to decide whether to use a framework or not. The things to consider are:

- the framework's domain must be sufficiently close to the domain/problem that the application developers are trying to address.
- the framework must be sufficiently flexible so the application developers can adopt it to the specific context. If the framework is lacking ways to customize it in places where the application developer needs to adjust the provided functionality then the framework is of course less suited or inappropriate for the problem at hand.
- the framework must deliver a design, functionality, and domain knowledge that otherwise is very expensive to acquire. This speaks in favor of frameworks of some complexity and size.
- the framework implementation must be reliable. Reusing a framework that is full of defects is definitely only "reused" once.

However if these properties are met, then a major development investment is saved: a good framework provides a quality design and reliable implementation of complex functionality. Thus there is a saving both in immediate development time due to the reuse as well as a saving in the following maintenance period as the framework code is thoroughly tested and less defects are expected to appear.

If the application developers have adopted a certain framework then a contract must be followed. Successful use relies on that the application developers:

- understand the protocols between the framework and the customization code they have to provide.
- understand the aspects that can (and cannot) be customized as well as the concrete techniques used for the customization.

If the framework's interaction patterns are not understood then the application developers are in big trouble, and the result is that developers "fight" the framework rather than follow its guidelines. This leads to bulky, error prone, and unstable code. In the early days of event-driven programming for window and mouse based applications, many programmers that were used to programming console-based applications had great difficulties in understanding the new guidelines leading to unreliable and slow applications.

Thus, a framework requires a substantial initial investment in training that should not be overlooked. Consider, for example, the number of books published about Swing, Enterprise JavaBeans, and other big frameworks.

32.4 Frozen and Hot Spots

A framework consists of *frozen spots* and *hot spots*, a terminology introduced by Pree (1994). The frozen and hot spots are the parts of the framework code that is fixed or variable.

Definition: **Frozen spot**

A part of framework code that cannot be altered and defines the basic design and the object protocols in the final application.

Definition: **Hot spot**

A clearly defined part of the framework in which specialization code can alter or add behavior to the final application.

Thus, the challenge for the framework designers is both to identify the overall architecture, the frozen parts, as well as define which parts that are allowed to be customizable, the hot parts. The latter of course involves defining the concrete programming techniques to allow application developers to insert their code to add or alter behavior.

The frozen term is good because it points to an important property of frameworks:

Key Point: Frameworks are not customized by code modification

A framework is a closed, blackbox, software component in which the source code must not be altered even if it is accessible. Customization must only take place through providing behavior in the hot spots by those mechanisms laid out by the framework developers.

Usually, frameworks are delivered as sealed components, for instance in the form of a binary library. As an example, MiniDraw is a Java jar file and customization is only done by defining new tools, putting image files in the right folder, and/or configuring the factory with proper implementations of MiniDraw's roles. Even though you can find the MiniDraw source code on the website and thus alter it, this is not the proper way to use it: remember *change by addition, not by modification.*

The term "hotspot" is not a very precise term. Other terms you may see is **hook method** or simply hooks, or **variability point**. The hook metaphor is a good one—think of the framework as "software with some hooks", you can then attach your own code to the hooks to alter the behavior of the framework. Many frameworks come with standard implementations for most or all of the hooks, so you only need to fill out a few to get something going. However, as I have used the term *variability point* during most of the book I will generally stick to it.

I will use the term **framework code** to identify the code that defines the framework whereas I use the term **application code** for the code that defines the specialization of the variability points and the "boilerplate" code for initializing the framework. Often the application code also contains code that is not related to the framework: as an example consider using MiniDraw to make a small UML class diagram editor that could generate Java classes from the class diagrams. The application code would have to define both MiniDraw customizations such as UML class box figures etc., as well as the code to generate Java classes.

32.5 Defining Variability Points

Given that framework code cannot be altered and an application developer has to customize it, he has to have mechanisms to "glue" his code into the frameworks variability points. The underlying techniques for object-oriented software have already been thoroughly treated in this book: the relation between frozen and hot spots and the TEMPLATE METHOD should be obvious. The template method defines the algorithm's structure as well as the fixed behavior, that is the frozen part, while hook methods are called that may add or alter behavior, the hot parts. Thus the most common way of defining variability points are either by subclassing or by delegation.

The next issue is how to tell the framework which concrete instances to use? One thing is to implement a HookInterface, the next thing is to give the framework the object reference to an instance of the implementing class. The answer is basically the *dependency injection* principle: let the client objects (those in the application) establish the dependencies between framework objects and objects implementing the variability points.

Key Point: Frameworks must use dependency injection

Framework objects cannot themselves instantiate objects that define variability points, these have to be instantiated by the application code and injected into the framework.

A more hands-on rule is that framework code must never contain a new statement on classes that have hot spot methods defined.

Exercise 32.2: List the techniques that the MiniDraw example applications used to inject dependencies to the application specific objects. Discuss them in light of the key point above.

Frameworks present a range of possibilities for defining the hot spot objects.

- *Existing concrete classes.* The framework comes along with a (large) number of predefined classes and your job is to compose these into something sensible. AWT and Swing are examples where you compose new graphical user interfaces from predefined components.
- *Subclassing an abstract class:* the framework contains abstract classes so most of a high quality implementation is already given, leaving you with the task of filling out the last details.
- *Implementing an interface:* the framework contains an interface, giving the application developer the opportunity to exercise full control over the hot spots.

The list above is of course also a spectrum of ease and speed versus control. It is easy and fast to configure a framework purely by injecting dependencies to the proper components but you have only a limited set of options. In the opposite end of the spectrum you can develop all the details in an object that simply implements a framework interface but this is of course much slower and there is a larger potential of defects. The latter also require you to have an intimate understanding of the framework protocol: in which order are which methods called and what are the pre and post conditions that must be satisfied?

This spectrum also describes a best strategy for defining variability point classes in a framework.

> **Key Point: Frameworks should support the spectrum from no implementation (interface) over partial (abstract) to full (concrete) implementation for variability points**
>
> *A framework provides the optimal range of possibilities for the application developer if all variability points are declared in interfaces, if these interfaces are partially implemented by abstract classes providing "common case" behavior, and if a set of concrete classes for common usages is provided.*

In this way the application developer has the full range of possibilities at his disposal.

Exercise 32.3: Review the MiniDraw framework code and find examples of MiniDraw following the above guideline.

Exercise 32.4: Analyze to what degree Swing graphical component classes obey the key point above.

It should be mentioned that of course *parameterization* can also be used to change framework behavior. Simple parameterization is for instance to set some parameters in a framework call. Advanced parameters can be given in property files or XML files whose format the framework defines.

Frameworks is not purely an object-oriented construct. At the machine language level, polymorphic method invocation is simply jumps via a jump table, so you can define frameworks in assembler or procedural languages using function pointers.

32.6 Inversion of Control

A prominent feature of frameworks is that they define the flow of control in the resulting application. This is also covered by some of the framework definitions mentioned that speak of "collaborating classes". The instances in the running framework call each other in predefined ways and occasionally "sneak out" into your code when they invoke a hotspot method. This is called the principle of **inversion of control**: the framework defines the flow of control, not you. A more colorful rephrasing is "the Hollywood principle," that is, "Don't call us, we'll call you." As an example, once you call `open()` on the MiniDraw editor, it does all the processing of mouse events and calls your tool and draws your images at the appropriate times.

Exercise 32.5: Relate the inversion of control property to the use of TEMPLATE METHOD in frameworks.

Thus, frameworks are reusable pieces of code that are very different from traditional libraries. This is shown in Figure 32.1. A traditional library is shown on the left and an application reusing code from the library maintains the overall control during execution; occasionally calling a method in the class library. An example of a library in Java is *java.lang.Math*: it contains a lot of code to calculate cosine, logarithms, etc., but it does not dictate the flow of control in your application. In a framework, shown on the right, the overall control during execution remains in the framework that occasionally calls a method in some application specific code via a hotspot.

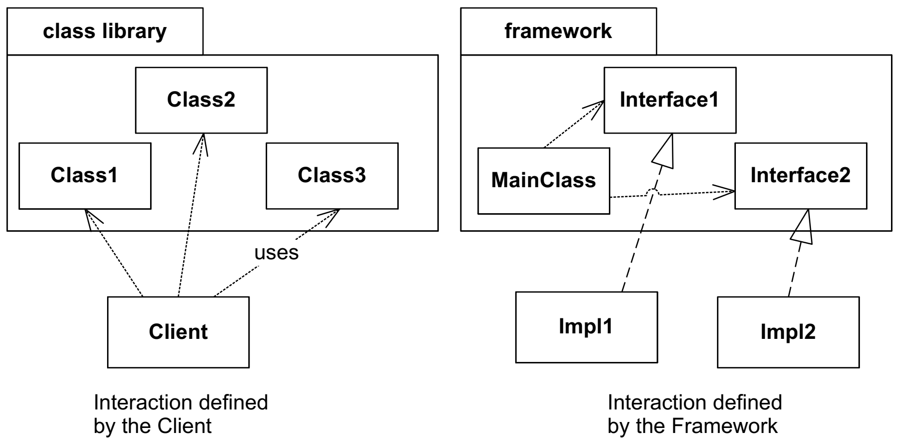

Figure 32.1: Inversion of control in frameworks.

32.7 Framework Composition

Frameworks are a way of reusing software and thus speed up development and save a lot of money at the same time. In many large scale applications it is therefore interesting to use two or more frameworks as many types of domains may be covered in a single system. For instance it may be obvious to reuse a graphical framework, a framework for distributed objects, a framework to handle transactions, etc.

If the frameworks are based upon subclassing of abstract classes then application developers will run into a **framework composition problem**. This is illustrated in Figure 32.2. We are faced with the problem that in the same class we must redefine hotspots from *two* different classes, one from each framework, but both frameworks define the classes as abstract. In Java we have no way of multiple inheriting from two abstract classes. In this situation we are quite stuck and have to either forget about using both frameworks or come up with some very clumsy adaptation code.

The situation could have been avoided if the framework developers had followed the ① *program to an interface* principles as shown in Figure 32.3. Here interfaces have been used and we have no problem in Java as we can inherit all the contracts defined by the interfaces. The figure also shows that the customization class can itself inherit from an application specific class, a thing that was also ruled out in the previous figure. This technique is employed to compose MiniDraw with Swing as explained in Section 30.5.2. Note that the concrete class can still reuse functionality provided by

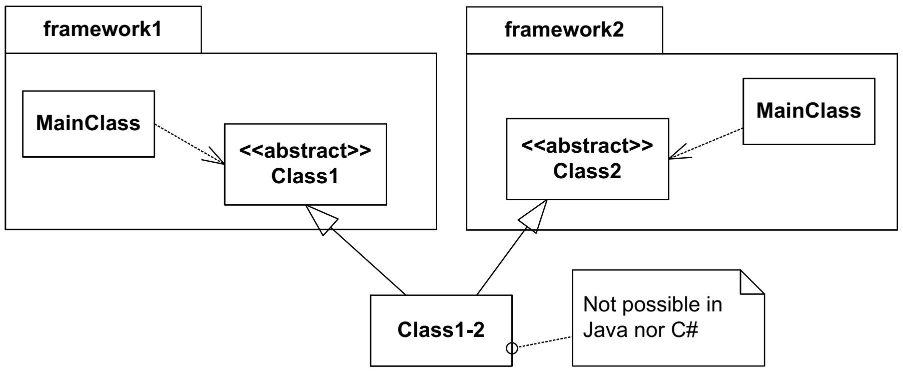

Figure 32.2: Composition problems in inheritance based frameworks.

the frameworks by delegating to instances defined therein, as is shown by Class1-2 delegating requests to Class1 from framework 1.

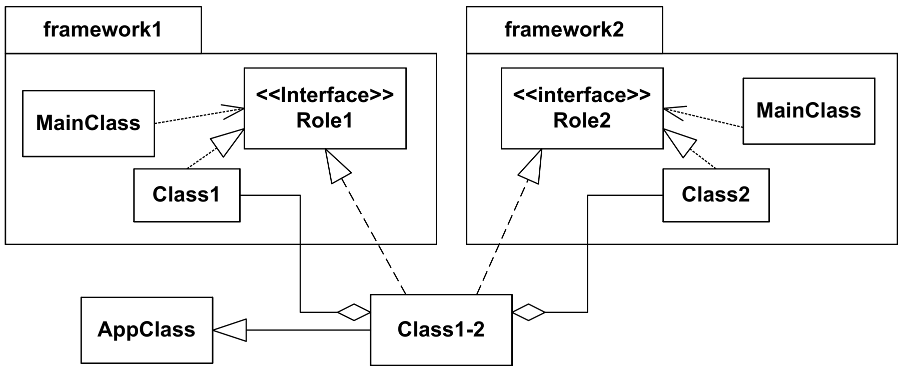

Figure 32.3: Avoiding the composition problem in delegation based frameworks.

Some frameworks pose additional problem of composition. Some frameworks insist on having the user event processing loop: Java Swing is an obvious example. If two frameworks both insist on having the event processing loop then combining them becomes highly problematic. Similar problems may occur if one framework is multi-threaded by design and the other inherently single-threaded. These problems can be very tricky to tackle and lead to the glue code being very difficult to overview and understand.

32.8 Software Reuse

The reason that frameworks are interesting is that it is one way of *reusing* software. Software reuse is the appealing idea of using existing software instead of writing it from scratch. This idea is of course attractive but the seemingly obvious potential has turned out to be rather complex to realize in practice.

Sidebar 32.1: MiniDraw Reuse Numbers

MiniDraw is approximately 1,600 lines of code (LOC) including comments, and the jar file about 30KB. The numbers for the three demonstration applications are: puzzle 55 LOC, rect 200 LOC, and marker 125 LOC. Thus if I calculate the percentage of lines of code that is reused in the final applications I get:

Puzzle: 97 %   Rect: 89 %   Marker: 93 %

Object-oriented frameworks have, however, gained quite some success in delivering the promise of software reuse. You can find some numbers on MiniDraw in sidebar 32.1. High reuse is primarily possible because A) a framework limits its functionality to a certain domain and B) it dictates the way it should be tailored. This allows the developers of the framework to cope with the problems that traditionally hinder software reuse, namely that the reusable software component does not fit the context of the software that it is going to be reused in. For frameworks, this problem is solved simply by the framework developers defining the context a priori.

Reuse can appear at many different levels in a software development project, for instance one may reuse code (by copy and paste, by using libraries of code made earlier), or one may reuse design (reusing a good solution for a known problem, like algorithms or design patterns). Frameworks are quite unique in this respect.

Key Point: Framework reuse is reuse of both design and code

A framework is a tangible unit of software, thus it is code reuse. However, due to the inversion of control property, it also defines the flow of control, object protocols, and clearly defines the variability points, and thus bundles a quality design as well.

A framework dictates how you must design your program—and if you do not follow these guidelines you are in a big mess. That is, the framework is reuse of a design—a way to structure a particular type of application or system. For instance Java AWT and Swing define a rigid way of structuring a graphical application: you must create a Frame instance, register listeners for window close events, add graphical components, etc. Thus you cannot structure a Swing application the same way as a console based one.

But a framework also comes with working code that you can use right away. For instance Swing comes with a large set of predefined graphical components that you can use and most standard graphical applications can be made just by reusing the provided components without modification.

32.9 Software Product Lines

The process that has been going on with the pay station—new customers with new requirements and my ambition to address these requirements in a way that keeps cohesion high, coupling low, and the production code under testing control to ensure reliability—has gradually transformed the production code to a framework. Frameworks are an important aspect of the larger context known as software product lines.

Definition: **Software product line**

A *software product line* is a set of software intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way. (Bass et al. 2003, p. 353)

A software product line is similar to product lines in more traditional manufacturing like car production where a given product usually comes in different variants: standard model, deluxe, convertible, etc. It is the same basic software system but with a managed set of features, that is, variants.

However, a framework is just the technical, software development, aspect. A software product line must also consider the managerial, economic, strategic, and market aspects. These aspects each pose their own and complex sets of problems, but is beyond the scope of this book. Just to mention a few aspects, it does not help a company to develop a framework if the sales people are not trained so they know which features that are easy to make (i.e. have a well-defined variability point in the framework) or difficult to make (i.e. is not addressed by the framework): odds are they will not negotiate to neither the customer's nor the company's advantage. Another aspect is the organizational aspects like "lost empires": if product line reuse will lead to a 50% cut in testing compared to building applications from scratch, the testing department may fight this idea vigorously.

32.10 Summary of Key Concepts

A framework is a reusable design together with implementation for a specific class of applications. The users of frameworks are application developers that tailor the framework so it fits the needs of the end users. The advantage of using a framework instead of building the application from scratch is that you get a reliable implementation with few defects, development time is cut due to the code you get from the framework, and you learn a high quality design for building applications within the domain. The disadvantage is the time and effort invested in learning the framework.

At the code level, a framework is considered to consist of *frozen spots* and *hot spots*. The frozen spots cannot be altered whereas the hot spots define the parts of the code where the application developer may alter or add behavior. Frameworks are customized by *dependency injection*, that is, the objects defining the exact behavior of the variability points are injected into the framework. A special characteristic of frameworks is the *inversion of control* which means the framework decides flow of control and invokes the application developer specified variability points at well-defined spots. A framework is a special case of software reuse, and is unique in that it combines reusing design as well as concrete implementation. *Software product lines* are frameworks combined with an organization that considers the managerial, economic, strategic, and market aspects of delivering highly flexible software systems.

32.11 Selected Solutions

Discussion of Exercise 32.1:

Given the characteristics, Java Swing is a framework: it is a *skeleton within a domain*, graphical user interfaces, and consists of a large number of *collaborating classes*. You generally *customize* it by composition, that is you create a dialog by composing the right set of buttons, list boxes, panels, etc. And of course, Swing comes with an immense amount of code.

Discussion of Exercise 32.2:

Concrete objects that were all injected into MiniDraw by the example applications are:

- ImageFigure instances with specific images of e.g. the nine puzzle pieces were injected through following the convention detailed by MiniDraw of how to automatically load GIF images.
- A factory was injected that itself injected the type of drawing, drawing view, and status field to use.
- New tools were defined and injected by a simple setTool method call on the editor.
- New figure types were injected simply by adding them to the drawing's collection of figures.

Discussion of Exercise 32.3:

The MiniDraw **Figure** role has the full spectrum. As seen in Figure 30.5 there is the Figure interface, an AbstractFigure (which is reused to create the RectangleFigure in the "Rect" application), as well as the concrete ImageFigure defined by MiniDraw.

The **Tool** role also has interface, abstract class, as well as default concrete classes.

Discussion of Exercise 32.4:

Generally, the visible components of both the AWT and Swing are concrete classes that are rooted in an abstract class: *java.awt.Component* or *javax.swing.JComponent* respectively. This makes it impossible to make a dialog where a JLabel is replaced by a third party graphical label without modifying the dialog code: you cannot inject another type of label .

Discussion of Exercise 32.5:

They are two sides of the same coin. It is the template method that defines the flow of control through its definition of algorithm structure and only call out into the application developer's code when hook methods are called. Of course a framework is not a single template method but a large set of ordinary and template methods that execute in the framework thereby calling a lot of different hook methods defined by many different hook interfaces or abstract methods in the framework.

32.12 Review Questions

What is a framework? Mention some of the characteristics of frameworks and relate them to for instance the pay station system, MiniDraw, Java Swing, or other frameworks.

Mention some of the considerations an application developer must consider when deciding to use a framework or not.

Define and explain the concept: Frozen spot, hot spot, variability point, and inversion of control.

Relate the TEMPLATE METHOD design pattern to the inversion of control property.

Describe implementation techniques to define variability points and discuss each technique's benefits and liabilities.

Describe the problems that may arise when several different frameworks must be combined in a single application.

Explain why A) copy a piece of code from the last project and paste it into my new project; B) enter an algorithm I have read in a magazine; C) call methods in the *java.lang.Math* package; are not considered framework reuse.