A SURVEY OF FRAMEWORKS

Juha Hautamäki

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF TAMPERE

REPORT A-1997-3

UNIVERSITY OF TAMPERE
DEPARTMENT OF COMPUTER SCIENCE
SERIES OF PUBLICATIONS A
A-1997-3, MARCH 1997

A SURVEY OF FRAMEWORKS

Juha Hautamäki

University of Tampere
Department of Computer Science
P.O. Box 607
FIN-33101 Tampere, Finland

ISBN 951-44-4131-1
ISSN 0783-6910

A Survey of Frameworks

Juha Hautamäki

FRED project
University of Tampere
Department of Computer Science
March 1997

Abstract

A framework is a tool to improve the software developing process. It is a reusable design expressed as a set of abstract classes and the way their instances collaborate. With frameworks, application developers don't have to start from scratch each time they write an application. Instead they specialize framework's behavior to suit the current problem by extending and refining its classes and operations. The purpose of this paper is to give an overview of frameworks as well as tools and methods supporting their use and design. This paper is one of the preliminary reports of the FRED (Framework Editor for Java) project. Other preliminary reports are "Application Frameworks in the Java Environment" [VilA97] and "Tools Supporting the Use of Design Patterns in Frameworks" [VilJ97]. The project home page is http://www.cs.Helsinki.FI/~viljamaa/fred.

1 Introduction

Software development has been motivated largely by the quest to help developers produce software faster and to deliver more value to end-users. This goal has moved the software industry to embrace object-oriented technology because of its potential to significantly increase developer productivity [Tal93]. Object-oriented frameworks are one way to use this technology in a most efficient way.

Roberts and Johnson [RoJ96] define frameworks as reusable designs of all or part of a software system described by a set of abstract classes and the way instances of those classes collaborate. A framework captures the programming expertise necessary to solve problems in a particular problem domain; it hides the parts of the design that are common to all applications in that domain, and makes explicit the pieces that need to be customized. A typical framework usually provides the design for only a part of a system, such as its user interface, though application specific frameworks sometimes describe an entire application. A framework is most valuable when it preimplements the most difficult part of the problem domain.

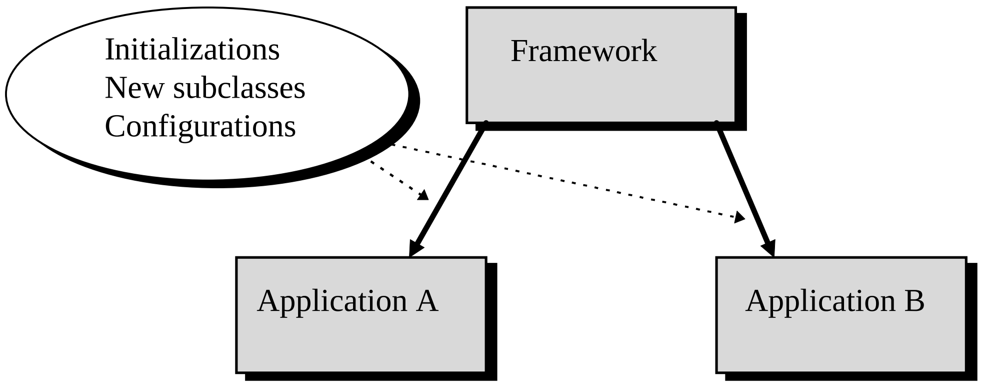


Figure 1. A framework and two specialized applications.

An object-oriented framework is a kind of class library. However, frameworks are more than well written class libraries. With frameworks the flow of control is bi-directional between the application and the library; that is, the library can call user written code and vice versa. This feature is achieved by dynamic binding in object-oriented languages where an operation can be defined in a library class but implemented in a subclass in the application [Mat96a]. Also, creating local extensions with a simple class library is more difficult than with a framework.

Johnson [Joh96] says that large-scale reuse of object-oriented libraries requires frameworks. The framework provides a context for the components in the library to be reused. However, Taligent Inc. [Tal94] has noticed that some libraries exhibit framework-like behaviour, and some frameworks can be used like class libraries. This can be viewed as a continuum, with traditional class libraries at one end and sophisticated frameworks at the other.

Frameworks are discussed by number of other authors as well. The most of the references used in this paper have been found from the Internet; e.g., Mattsson’s [Mat96b] bibliography has been helpful. The goal of this paper is to give an overview of this widely discussed area. First the basic concepts are introduced in the second chapter. Then, framework experiences and tools are discussed in the third chapter. The design of frameworks is discussed in the fourth chapter. Some existing frameworks are presented in the fifth chapter. We assume that the reader already knows the principles of object-oriented programming.

2 Basic Concepts

In this chapter some common concepts dealing with frameworks are introduced. It is assumed that the reader already knows object-oriented concepts like inheritance, polymorphism, dynamic-binding, superclass, subclass, etc. However, abstract classes and concrete classes are briefly discussed in the first subchapter; they are essential in order to understand frameworks. Frameworks can be divided into two categories (white-box, black-box); this is discussed in the second subchapter. Some important commonly used terms (hot spots, cookbooks, and the Hollywood principle) are introduced in the third subchapter. Design patterns are briefly discussed in the fourth subchapter; they are useful when designing and documenting frameworks.

2.1 Abstract Classes and Concrete Classes

An *abstract class* is a design for a single object, whereas a framework is a design of a set of collaborating objects with a set of responsibilites. By definition an abstract class is a class with at least one operation left unimplemented (note that here the term ”operation” means the same as ”function” or ”method”). Because some operations are unimplemented, an abstract class has no instances and is used only as a superclass. Thus, it is designed to be used as a template for specifying subclasses rather than objects. Some object-oriented programming languages provide no direct support for abstract classes. E.g., early versions of C++ had no support for abstract classes, but currently C++ allows abstract operations to be declared as ”pure virtual” [Str91], and Java lets them be declared as ”abstract” [ArG96]. Abstract classes have three kinds of operations:

- *Abstract operations* are not implemented by the abstract class but this task is left to its subclasses.
- *Template operations* are abstract algorithms defined in terms of one or more abstract operations. The subclass specializes the template operations by implementing the abstract operations. Thus template operations are partially implemented.
- *Base operations* are fully implemented.

Why do we need abstractions? Strict modeling of the real world leads to a system that reflects today’s realities but not necessary tomorrow’s [GHJ95]. Due to their nature frameworks must be flexible and the abstractions found during the design process are key elements to making the design flexible. However, finding these abstractions is difficult. In general, it seems that an abstraction is usually discovered by generalizing from a number of concrete examples. An experienced designer can sometimes invent an abstract class from scratch, but only after having implemented concrete versions for several other projects. One sign that a good abstraction has been found is that code size decreases, indicating that code is being reused.

Classes that are not abstract are *concrete classes*. Concrete classes are implementations of abstract classes where their interfaces have been refained [CIM92]. A concrete subclass of an abstract class will provide an implementation for any operation that needs one, and will inherit the implementations of the other operations. A framework usually comes with abstract classes and some preimplemented concrete classes; together they form the skeleton of a working application.

2.2 White-Box and Black-Box Frameworks

White-box frameworks

A framework is called *white-box* if it is based on inheritance. An application developer customizes a white-box framework by deriving new subclasses and overriding member functions. The term ”white-box” refers to visibility; due to inheritance the internals of parent classes are often visible to subclasses. Object inheritance uses the ”IsA” relationship (see, e.g., [Dei94]) where the derived class creates a new kind of object whose type is the original class.

Gamma *et al.* [GHJ95] enumerate advantages and disadvantages of white-box frameworks: Class inheritance is defined statically at compile-time and is straightforward to use. It also makes it easier to modify the implementation being reused. However, one can’t change at run-time the implementations inherited from parent classes. Also, the implementation of a subclass is bound up with the implementation of its parent class. This dependency limits flexibility and ultimately reusability. It is also said that inheritance breaks encapsulation. One cure is to inherit only from abstract classes, since they usually provide little or no implementation.

There is no strict line between a white-box framework and a simple class hierarchy; every class hierarchy offers the possibility of becoming a white-box framework [JoF88]. In its simplest form, a white-box framework is a program skeleton, and the user-derived subclasses are additions to that skeleton. Usually the development process starts with the white-box approach; white-box frameworks should be seen as a natural stage in the evolution of a system. As the system evolves, the designer will see which parts of the framework should be specialized.

Black-box frameworks

Black-box frameworks are based on object composition. That is, new functionality is obtained by assembling or composing objects. An application developer then uses these assembled components to specify the application behavior. Object composition is known to be a more flexible mechanism than inheritance, precisely because object compositions can be changed dynamically, whereas inheritance structures are static compile-time concepts. Object composition uses the ”HasA” relationship (see, e.g., [Dei94]), in which one object uses another object by reference. Object composition requires that the objects being composed have well-defined interfaces, so that the user needs to understand only the external interface (e.g., function names) of the components.

Gamma *et al.* [GHJ95] discuss the advantages of object composition. Unlike with inheritance, no internal details of component objects are visible and any object can be replaced at run-time by another object of compatible type. Object composition helps keeping each class encapsulated and focused on one task. Also, classes and class hierarchies will remain small. One should be able to get all the functionality one needs just by assembling existing components through object composition. However, in practice the set of available components is never rich enough. Reuse by inheritance makes it easier to build new components that can be composed with old ones. Hence, often inheritance and object composition work together.

Johnson and Foote [JoF88] have noticed that white-box frameworks tend to evolve into black-box ones. This is due to the fact that the design of a particular framework system gradually becomes better understood, which leads to components with higher functionality. However, this process is not obvious; many frameworks will not complete the journey from white-box skeleton to black-box framework during their lifetime. Black-box relationships are an ideal towards which a system should evolve. However, a white-box framework can be a finished product, too.

2.3 Hot spots, Cookbooks, and the Hollywood Principle

Hot spots

Succesful framework development requires the identification of domain specific *hot spots* [Pre94]. These are the places where the framework will be extended by the application developers. Thus, hot spots are kind of slots in the application domain; it is the developer’s task to fill these slots with his or her own solutions. Hot spots make the framework flexible. For instance, a language implementation framework has an abstract class representing language structures, where the operations defining the actual syntax and semantics of the current language structure are left unimplemented; by overriding these abstract operations the user defines the particular target language.

Cookbooks

A document containing instructions that describe typical ways to use a framework is often called a *cookbook* [Pre94]. Correspondingly, individual instructions in a cookbook are called *recipes*. Hence, a cookbook contains numerous recipes which describe in an informal way how to use a framework in order to solve specific problems. The framework's internal design and implementation details are usually omitted in these kind of instructions.

The Hollywood principle

An important characteristic of a framework is that the methods defined by the user to tailor the framework will often be called from within the framework itself, rather than from the user's application code [JoF88]. This mechanism is often called ”the Hollywood principle” or ”Don't call us, we'll call you”.

2.4 Design Patterns

Originally architect Christopher Alexander developed the idea of *design patterns* to enable people to design their own homes and communities [AIS77]. His pattern language was a set of patterns, each describing how to solve a particular kind of design problem. The pattern language starts at a very large scale, explaining how the world should be broken into nations and nations into smaller regions, and goes on to explain how to arrange roads, parking, shopping, places to work, homes, etc. The patterns focus on finer and finer levels of detail. Each pattern was written in a particular format, leading into the next one(s).

Correspondingly, a framework can be designed and documented in terms of patterns, where each pattern describes how to solve a small part of the larger design problem. Each pattern describes a problem that occurs over and over again in the problem domain of the framework, and then describes how to solve that problem. Instead of having to choose from an almost infinite number of possible combinations of actions, patterns allow the solution of problems by providing time-tested combinations that work [Pre94]. Thus, the idea is that if someone has solved a specific problem once and documented the solution as a pattern, we can use this information in order to solve same kind of problems.

Gamma *et al.* [GHJ95] have enumerated several commonly used design patterns. They have found that, in general, a pattern has four essential elements:

- *Pattern name*. The name individualizes the pattern. A good name is vital, because it will become part of design vocabulary.

- *Problem*. The problem describes when to apply the pattern. It explains the problem and its context.
- *Solution*. The solution describes the elements that make up the design, their relationships, responsibilities, and collaborations in an abstract level. There are several ways to describe this information; e.g., when giving a graphical representation of the classes in the pattern we can use techniques like the Object Modeling Technique (OMT) presented by Rumbaugh *et al.* [RBP91].
- *Consequences*. The consequences section describes how the pattern supports its objectives. They are the result and trade-offs of applying the pattern. They should also tell what aspect of system structure can be varied independently.

What is the relationship between frameworks and patterns? Frameworks are software whereas patterns are knowledge about software [Mat96a]. Thus, design patterns are more abstract than frameworks. Design patterns are also smaller architectural elements than frameworks; a typical framework contains several design patterns.

3 Using Frameworks

A typical framework is an extensible skeleton which provides the basic functionality in some specific problem domain. The application developer fills in the needed extra functionality by extending the framework's functionality. This is done by deriving new classes and overriding member functions (white-box frameworks), and/or by introducing different combinations of objects (black-box frameworks). In this chapter frameworks are discussed from the user's point of view. Advantages and problems are discussed in the first and second subchapter. Usually a framework comes with a set of tools in order to make the application development easier; tools are discussed in the third subchapter.

3.1 Benefits

The role of a framework is to provide the general flow of control, while the developer's code waits for a call from the framework (the Hollywood principle). This is a significant benefit since developers do not have to be concerned with the details, but can focus their attention on their particular problem. By providing an infastructure, the framework decreases the amount of standard code that the developer has to program, test, and debug. The developer writes only the code that extends or specifies the framework behavior to suit the program's requirements [Tal93]. For instance, IBM's Commercial Shareable Frameworks (project San Francisco) provide core business processes packaged as high-level extendible business frameworks. With these frameworks the application developer can create solutions for a broad range of industrial sectors such as wholesale, retail distribution, finance, etc. These frameworks contain the basic application structure that application programmers previously had to develop on their own [IBM96a].

Frameworks store experience; problems are solved once and the business rules and designs are used consistently. This allows an organization to build from a base that has been proven to work in the past. Another advantage is that different applications created with a particular framework share the same basic structure. The resulting system is better integrated from a user's point of view, while requiring less work by developers to get their programs to work together [Tal93]. Maintaining multiple applications with a single framework is also easier than maintaining multiple applications with multiple design solutions.

An important benefit of frameworks is that they help coordinate people working on the same project. One way for this is by providing standard interfaces that can be widely reused. Another way frameworks help coordinate a project is by providing a way to divide work; a project can be divided into those improving, extending, or developing frameworks and those using them for a particular application [JoR91]. Frameworks enable rapid prototyping, too [CIM92].

A framework's algorithms and data structures are automatically reused by every instantiation of the framework. However, a framework is not just a collection of classes but it also defines a generic design. When using a framework, its design will be reused as well. Because of the bi-directional flow of control a framework can contain much more functionality than a traditional class library.

The following list is a summary of benefits presented in this chapter:

- With frameworks application developers can focus their attention on their particular problem domain.
- Frameworks decrease the amount of code that the developer has to program, test, and debug.
- Frameworks store experience.
- The resulting system is better integrated.
- Maintaining multiple applications created with a particular framework is easier.
- Frameworks help coordinate people working on the same project.
- Frameworks allow rapid prototyping.
- Algorithms, data structures, and the design of frameworks are all reused.

3.2 Problems

The main obstacle to using frameworks is the learning curve [Mat96a]. Developers must learn how to design and develop objects. They must also understand what functions frameworks provide, and how to extend and customize them for their own applications [IBM96b]. This is never easy if the domain is non-trivial.

Pure white-box frameworks can be difficult to use because they require application developers to write a substantial amount of code. Johnson and Foote [JoF88] say that the major problem with such a framework is that every application requires the creation of many new subclasses. While most of these new subclasses are simple, their number can make the task difficult for an unexperienced programmer. On the other hand, black-box frameworks based on object composition are generally easy to use (once you have learned them), but they can be limiting [Tal94].

Demeyer *et al.* [DMN97] present more drawbacks concerning understandability of inheritance and object composition. They say that a classical difficulty in comprehending an object-oriented design is that the inheritance hierarchy of an application tells suprisingly little about its architecture.

This is due to the fact that inheritance describes relationships between classes, not objects. Object composition fares little better, but not much, since the object structures can be built and modified at runtime, similarly making the architecture of a framework hard to understand.

Documenting frameworks is difficult because they are more abstract than most software. However, Johnson [Joh92] says that patterns are a good way to describe frameworks because novice users of a framework will usually not want to know exactly how it works, but will only be interested in solving a particular problem (recall cookbooks in chapter 2.3). As long as the patterns are powerful enough to describe most initial uses of the framework, it will meet the needs of the application developers. However, Johnson continues, nobody understands a framework until he/she has used it, so using a framework is more important than studying the theory behind it. Johnson believes that theory should follow practice, that a discussion of the theory behind a framework is only understandable once the framework itself is understood, and that the best way to get a general understanding of a framework is to use it.

The following list is a summary of problems presented in this chapter:

- The main obstacle to using frameworks is the learning curve.
- White-box frameworks can be difficult to use because they require clients to write a substantial amount of code to produce interesting behavior.
- Black-box frameworks can be limiting.
- Frameworks are more abstract than most software, which makes their documentation difficult.
- A framework must be used in order to understand it.

3.3 Tools and Toolkits

A framework without supporting tools is often hard to use; it provides no mechanisms to reduce the learning curve and to manage the framework features. Therefore a framework usually comes with tools supporting application development. But what is a tool or toolkit? Johnson and Foote [JoF88] define a toolkit as a collection of high level tools that allow a user to interact with an application framework to configure and construct new applications.

Hence, a tool can be a big application with high level functionality, or it can be a single executable file taking few parameters. A tool can have a graphical user interface, or it can be executed from the command line prompt. In other words a tool is something that makes the application developers task easier. To accurately list all possible tools and toolkits is impossible because the needs of the frameworks vary. However, the following kinds of tools are often used with frameworks:

- Browsers are used to view information. A typical browser presents the framework's class hierarchy, project organization, etc.
- Editors are used to modify and combine framework elements. E.g., it is typical to open an editor for a feature when its name is double clicked in a browser.
- Help tools offer guidance. This guidance can be, e.g., a hypertext document.
- Test tools can be used to test the resulting applications. This enables also simulation.
- Documentation tools help to create documentation for the resulting applications. This kind of tool is, e.g., the Javadoc tool [ArG96] which generates documentation from Java source files.

Often there is a browser for viewing and managing data, like a class hierarchy, and editors for editing individual classes and for handling components. The following kinds of tools are planned in IBM's San Francisco project: A tool for visual representation of the framework's processes, a tool for viewing tasks used by the business process, an editor for business processes, a palette of existing frameworks that can be included using a drag-and-drop metaphor, and tools for business modelling and simulation. The project also considers help tools which would guide developers through the development process by identifying hot spots (recall chapter 2.3) and which would provide hints/tips for performance improvement [IBM96b].

Ideally, one should be able to construct an application almost without programming, for instance, by selecting icons representing standard components and application structures, connecting them graphically and letting the system generate an executable program. One such a tool is Vista [Mey90a, Mey90b], a graphical editor for object-oriented applications. It has being developed as part of the ITHACA (Intelligent Tools for Highly Advanced Commercial Applications) ADE (Application Development Environment) by the Centre Universitaire d'Informatique of the University of Geneva [ANM90]. Vista is based on C++ and X, using the Motif user interface toolkit. Vista supports the selection of components, the graphical linking of components, immediate execution of scripts and the saving of scripts [FNP92].

Mey and Nierstrasz [MeN93] describe the user interface of Vista: It is composed of three windows: message window, control window and tool window. The message window displays system messages which supply information or error messages to the user. The control window contains a scroll list of the plug-compatible software components that can be used or edited. There is also an area to select link presentation (color, style, width). The tool window contains the command menu and the drawing area. The drawing area is where the user instantiates components.

The command menu is used to execute an operation on a component or a link. The user interface of the tool, the components and the component ports are sensitive to user input. This approach allows both the internal behaviour of applications, as well as their user interface, to be directly edited and manipulated. For visual composition to work, component sets and composition models must be available. A component set is a collection of software components from which applications may be constructed. A composition model defines the allowable composition interfaces for a given component set and the possible connections between components.

The main point is that high level tools reduce both coding and remembering efforts. Black-box frameworks are better at serving as the foundation of a toolkit [JoF88]; a tool can let the user choose preimplemented components and connect them together. This is easier than deriving new subclasses and overriding operations.

4 Designing Frameworks

Though there are numerous design techniques for object-oriented programming, developing object-oriented frameworks is significantly more difficult than developing individual applications. Developers have noticed that framework designing is an iterative process which requires both domain and design expertise. A framework must be simple enough to be learned. On the other hand it must embody a theory of the problem domain, and provide enough features to be useful. The problem is to find the reusable design and the hot spots making the framework flexible enough. Design patterns can be helpful in this process. Gamma *et al.* [GHJ95] say that a framework which uses design patterns is far more likely to achieve high levels of design and code reuse than one that doesn't use them. An added benefit comes when the framework is also documented with the design patterns it uses. Of course one has to know these design patterns in order to use them.

The following subchapters summarize some papers dealing with framework design. The first subchapter concentrates on practical issues, whereas the second one contains more systematic aspects. The text of these papers is reorganized, shortened, and sligthly modified. Matters dealing with framework design are emphasized.

4.1 Practical Viewpoints

There is a number of papers dealing with framework design. In chapter 4.1.1 experiences collected by Taligent Inc. are presented; their paper ”Building Object-Oriented Frameworks” [Tal94] is a practical one containing valuable development tips. Johnson's papers ”How to Design Frameworks” [Joh93] and ”Documenting Frameworks Using Patterns” [Joh92] are summarized in chapter 4.1.2. Chapter 4.1.3 summarizes the paper ”Reusing Object-Oriented Design” [JoR91] written by Johnson and Russo; they emphasize that designing a framework is an iterative process where the dialog between the framework users and the framework developers plays an important role.

4.1.1 Building Object-Oriented Frameworks

Taligent is a subsidiary of IBM. As an important center for object technology, Taligent provides key software components for several IBM development tools. They have noticed that a framework should be easy to use for client programmers (i.e., for application developers). From the client's perspective, an easy-to-use framework performs useful functions with no extra effort. The framework works with little or no client code, and it supports small, incremental steps from the default behavior to sophisticated solutions. If clients don't understand how the framework works and how to use it, they'll develop their own solution. Booch [Boo94] says that the most profoundly elegant framework will never be reused unless the cost of understanding it and then using its abstractions is lower than the programmer's perceived cost of writing the application from scratch. Hence, one must strike a balance between the simplicity of the client interface and the power of the framework. A completely flexible framework can be difficult for the clients to learn, and difficult for the developers to support.

One approach is to build a very flexible, general framework from which one can derive additional frameworks for narrower problem domains. The overall framework provides generalized components and constraints to which the derived frameworks conform. Derived frameworks introduce additional components and constraints that support more specific solutions. These additional frameworks provide the default behavior and built-in functionality, while the general framework provides the flexibility. To be successful, one should design the frameworks to be:

- *Complete*. Frameworks support features needed by clients and provide default implementations and built-in functionality where possible. One shall provide concrete derivations for the abstarct classes in the frameworks and default member function implementations to make it easier for the clients to understand the framework and to allow them to focus on the areas that they need to customize.
- *Flexible*. Abstractions can be used in different contexts.
- *Extensible*. Clients can easily add and modify functionality. The developers shall provide hooks so that clients can customize the behavior of the framework by deriving new classes.
- *Understandable*. Client interactions with the framework are clear, and the frameworks are well-documented. The developers shall follow standard design and coding guidelines and provide sample applications that demonstrate the use of each framework. If the developers who build frameworks and those who use them follow the same guidelines, it facilitates reuse in both directions.

When one begins to develop a framework, many small iterations on the design and initial implementations must be made. As the framework matures and the clients begin to rely on it, the changes become less frequent and development moves into a much larger maintenance cycle. This process can be divided into four general tasks:

Identify and characterize the problem domain

- *Outline the process.* The first step in developing a framework is to analyze the problem domain and to identify the frameworks needed. That is, once the problem domain has been identified, the problem is broken down into a collection of subframeworks that can be used to build a solution. To determine what frameworks are needed, one should think in terms of families of applications rather than individual programs:

    - One should look for repeatedly build software solutions, particularly in key business areas.
    - One must identify what the solutions have in common and what is unique to each program. The common pieces, the parts that are constant across programs, become the foundation for the frameworks. These pieces are then factored into small, focused frameworks.

- *Examine existing solutions.*
- *Identify what parts of the process the framework will perform.* For each framework, one should identify the process it models. Once the process has been outlined abstractions can be detected.
- *Identify the key abstractions.* If the framework developers are familiar with the problem domain, they can recall their past experience and former designs to identify abstractions and begin designing the framework. If they are not experts in the problem domain, or haven't developed any applications for it, they should examine applications written by others and consider writing applications in the domain. Then they should factor out the common pieces and identify the abstractions.
- *Get input from clients and refine solutions.*

Define the architecture and design

- *Design how clients interact with the framework.* Developers must determine the classes and member functions the client will use. They should simplify the client's interaction with the framework to help prevent usage errors. It must be made as clear as possible in both framework's interfaces and documentation what's required of the clients. Developers should continually look for ways to reduce the amount of code the client must write:
    - Provide concrete implementations that can be used directly.
    - Minimize the number of classes that must be derived.
    - Minimize the number of member functions that must be overridden.
    - Use illustrative class and function names.
- *Provide tools to make client's tasks easier.*
- *Apply recurring design patterns.*
- *Get input from clients and refine solutions.*

Implement the framework

- *Implement the core classes.*
- *Test the framework.*
- *Ask clients to test the framework.*

- *Iterate to refine the design and add features*. Building a framework is an iterative process. Beginning with the initial design, one should work with clients to determine how the framework can be improved. Developers should continually look for ways to refine the framework by adding more default behavior, and additional ways for users to view and interact with the data. They must implement features, test them, and verify them with their clients. During this process, developers will go back and reanalyze the problem domain and refine their design using testing, client feedback, and their own insights.

Deploy the framework

- *Provide documentation*. Code comments and documentation are a necessary part of any programming project, but they are especially important when developing frameworks. If other developers don't understand the framework, they won't use it. Hence, one must make clear what classes can be used directly, what classes must be instantiated, and what classes must be overridden. Clients are interested in solving particular problems; usually the details of the framework implementation are not important. It is best to provide a variety of documentation:
    - At minimum, provide sample programs. Examples make frameworks more concrete and make it easier to understand the flow of control. In the process of developing and testing the framework, one has to develop applications that use the framework. Often these can provide a foundation set of samples. One should provide a variety of samples that demonstrate how to use the framework in different contexts.
    - Diagrams of the framework architecture.
    - Descriptions of the framework.
    - Descriptions of how to use the framework.
- *Establish a process for distribution*. Developers must also plan how the finished product will be distributed and supported. To use the frameworks, other developers need to know that they exist and know how to access them.
- *Provide technical support for clients*. To realize the benefits frameworks can provide, resources must be committed to support them.
- *Maintain and update the framework*. Early in its life, a framework will probably require routine maintenance to fix bugs and respond to client requests. Over time, even the best framework will probably need to be updated to support changing requirements. One must be able to assist clients and respond to their problems and requests. When a framework is updated, the impact on the clients should be minimized. A constantly changing framework is difficult, if not impossible, to use. It's better to add new classes instead of changing the existing class hierarchy, or to add new functions instead of changing or removing existing ones.

Frameworks shouldn't get too big. Developers should look for ways to break big frameworks down into small, focused frameworks. If frameworks are designed to interoperate, small frameworks are more flexible and can be reused more often. By breaking down the original framework into a set of small frameworks, the resulting frameworks can be used in other contexts. Also, one should enhance portability by isolating platform-dependent code to make it easier to port the framework. Designing for portability reduces the impact porting has on the clients.

Projects can often be factored into a number of separate frameworks and assigned to small teams. If it takes more than three or four programmers to produce a framework, it should probably be split into a set of smaller frameworks. Teams of two to four are usually more effective than teams of one, unless the single programmer is both an experienced framework developer and a domain expert. Working with several small teams introduces additional challenges:

- *Programmers are focused on one aspect of a large project and might not understand all of the interrelationships and client implications.* Appoint a project architect who maintains the ”big picture” and ensures that the frameworks ultimately work together.
- *Architectural consistency must be maintained across teams.* Follow standard design and coding guidelines.
- *Dependencies between frameworks can create bottlenecks.* Decouple the frameworks by isolating the dependencies in intermediary classes.

Often when one framework requires the services of another framework, the connection can be implemented through an interface or server object. Then, only one object is dependent on the other framework. Until the other framework can support the necessary operations, the intermediary class provides stub code that allows the rest of the framework to be tested. Loosely-coupled frameworks are generally more flexible from the client’s perspective, too.

Frameworks are a long-term investment; the benefits gained from developing frameworks are not necessary immediate:

- Framework designers need more time to create a framework than to create a procedural library.
- Clients need more time to learn a framework than to learn a procedural library.

4.1.2 Documenting and Designing Frameworks

This chapter is based on Johnson’s papers ”How to Design Framework” [Joh93] and ”Documenting Frameworks Using Patterns” [Joh92]. He has found that a typical way to develop a framework is the following: An application in a particular problem domain is developed in an object-oriented language. Then the application is divided into reusable and nonreusable parts. Then a second application is develop reusing as much software of the first application as possible. It will be noticed that the framework obtained from the first application is not very reusable, so it must be fixed. Then a third application is developed reusing as much software as possible. Again it is usually noticed that the framework is still not completely reusable. The framework is fixed and the iteration continues.

According to Johnson, a good way to develop a framework is to pick two similar applications that need to be developed and that are obviously in the application domain. These applications determine the framework. To find them start with similar applications illustrating features that the framework must support. They will provide a base for generalization. Find common abstractions, figure how to decompose problem into standard components, parametrize. The project group should also include peoples who have already developed some applications in the current domain.

Divide the project into a framework group and two application groups. The framework group considers how other applications would reuse the framework and develops documentation and training for the framework. Application groups reuse as much software as possible. They will give feedback to the framework group. The project groups should also look for examples that will break the framework; perhaps they will reveal limits of the framework and lead to generalization. However, it is not practical to have too many special cases.

Abstractions are crucial for developing frameworks. Design patterns can help to find good abstractions. Usually it is best to design an abstract class by generalizing from concrete subclasses. To find out abstract classes and to generalize concrete classes create an empty superclass and move common code/variables to it. Rename functions to give classes the same interface. Usually code in the subclasses is almost, but not quite, the same. Abstract out differences, and move the rest to a superclass. The result is that subclasses have more, but smaller, methods. Look for commonalities, represent each idea once.

Many reusability bugs can be fixed by a small set of transformations. It is good to break large functions into smaller ones, e.g., similar code sequence inside functions can be turned into a new function in the same class. Similarly, large classes should be broken into components or subclasses. Multiple inheritance can always be simulated by dividing an object into pieces; sometimes, not always, this improves a design, too. It is also important to use descriptive names for classes, functions and variables.

The documentation should show how the framework can be used to build applications. It is common that framework users want to know as little as possible about the framework. This means that they are not interested in a description of the design of the framework, but want a kind of cookbook (recall chapter 2.3), giving detailed instructions for using the framework. This can be done by structuring the documentation as a set of patterns, sometimes called a pattern language, in which each pattern describes the problems it is meant to solve.

In the paper ”Documenting Frameworks Using Patterns” [Joh92] Johnson has used the above described documentation method for the HotDraw graphic editor framework. In his approach each pattern is composed of the following parts:

- *Problem description.*
- *Detailed discussion of the different ways to solve the problem*. This includes also examples and references to other patterns. Examples play a key role in the framework documentation. They make frameworks more concrete, make it easier to understand the flow of control, and help the reader to determine whether he or she understands the rest of the documentation.
- *Solution summary*. This summarizes the solution, and includes also references to patterns needed to complement the current pattern.

The first pattern should describe the framework’s application domain. It is usually hard to specify a problem domain precisely, but a small set of examples usually makes the general area clear. These examples are not intended to show how to use the framework to build applications, nor to explain the design of the framework, but rather to show what the framework is good for. In addition, the first pattern introduces the rest of the patterns in the language, and it will usually tell which patterns should be studied next. Thus, the first pattern acts both as a catalog entry for the framework and as a road map. The internal structure of a set of patterns minimizes the amount that has to be read to solve a problem, and also provides places to store design information.

A good documentation describes the framework’s design, too. This includes the different classes in the framework and the way that instances of these classes collaborate. Although programmers can usually interconnect objects without completely understanding how they work, and can even make subclasses following a cookbook, a framework is most useful to someone who understands it in detail. In general, detailed design information should be hidden as long as possible from the user, because most users are not interested in seeing it. On the other hand, it is necessary to make sure that the patterns contain the information somewhere and that the information is not duplicated.

Johnson says that documentation for a framework has three purposes, and patterns can help fulfill all of them. Documentation must describe:

- The purpose of the framework
- How to use the framework
- The detailed design of the framework.

4.1.3 Frameworks and Iteration

This chapter is based on the paper ”Reusing Object-Oriented Design” [JoR91] written by Johnson and Russo. They say that the framework’s range of applicability depends heavily on the examples on which it is based. Each example that is considered makes the framework or abstract class more general and reusable. Abstract classes are small, so it is easy to generate lots of examples on paper and reduce the chance of iteration. Frameworks are large, so it is too expensive to look at many examples, and paper designs are not sufficiently detailed to evaluate the framework.

Because frameworks require iteration and deep understanding of the application domain, it is hard to create them on schedule. Thus, framework design should never be on the critical path of an important project. On the other hand, framework design must be closely associated with the application developers; building applications with a framework shows which parts of the framework need to be improved. This is because changes are almost always motivated by trying to reuse the framework. A use of a framework validates it when the use does not require any changes to the framework, and helps improve the framework when it points out weaknesses in it. Thus, designers of a framework should collaborate closely with application developers.

Why is the iteration necessary? Clearly, a design is iterated only because its authors did not know how to do it right the first time. Perhaps this is the fault of the designers; they should have spent more time analyzing the problem domain, or they were not skilled enough. However, lack of experience is not the only reason for iteration. The main reason that framework design iterates is because frameworks are supposed to be reusable; all software requires iteration before it becomes reusable. This follows from the general observation that software never has a desirable property unless the software has been carefully examined and tested in terms of the property. The ultimate test for whether the software is reusable is to reuse it. It is not possible to reuse software until it has been written and it is working, so iteration is inevitable. The only exception is that software that is a reimplementation of an existing reusable software might not need iteration. This is because the new software is actually just a version of the old one, and the iteration took place already when the old version of the software was designed.

A common mistake is to start using a framework for important projects while its design is still iterating. It is better to first use the framework for some small pilot projects to make sure that it is sufficiently flexible and general. If not, these projects will be good test cases for the framework developers. A framework should not be used widely until it has proven itself; the more widely a framework is used, the more expensive it is to change it later.

The best frameworks comprise the work of many people. Multiple points of view are needed to learn which parts are likely to change. The dialog between users and developers of a framework plays an important role in its development. This does not mean that frameworks should be designed by a committee. A good framework has a conceptual integrity that is usually achieved only by a single person or a small group.

4.2 Systematic Approaches

Many authors have noticed that there are hardly any formal techniques for the design of frameworks. Many experts even believe that frameworks cannot be the result of systematic design, but that they rather evolve in a bottom-up fashion. However, in this chapter two more or less theoretical methods are introduced. In the first subchapter we introduce a pattern language suggested by Roberts and Johnson [RoJ96]; this language offers guidance in framework design. The second subchapter summarizes the paper ”Designing a Framework by Stepwise Generalization” [KoM95] written by Koskimies and Mössenböck; it presents a two-phase framework design method.

4.2.1 A Pattern Language for Developing Frameworks

In the paper ”Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks” [RoJ96] Roberts and Johnson present a pattern language which can be used during the framework's development process. The language contains patterns related to each other. Each pattern has a name, context, problem description, forces affecting the pattern, solution and its rationale, implementation instructions, examples and related patterns. Here we summarize those patterns; the pattern name is in bold face and references to the other patterns of the pattern language are underlined. Note that most of of the patterns are overlapping; sometimes one goes first, sometimes the other.

- **Three Examples:** How to start designing a framework? People develop abstractions by generalizing from concrete examples. Develop three applications that you believe that the framework should help you build. The general rule is: build an application, build a second application that is slightly different from the first, and finally build a third application that is even more different than the first two. Common abstractions will become apparent. There are two approaches to developing these applications. In the first approach, the applications are developed in sequence by a single team. This allows the team to begin reusing design immediately. In the second approach, the applications are developed in parallel by separate teams. This approach allows different points of view at the expense of the time it will take to unify these applications in the future. The most obvious way to follow this method is to simply build three applications in succession, making sure that both code and people are carried over from one project to the next. Another way is to prototype several applications without building industrial strength versions of any of them. Note that no special object-oriented design techniques are needed when building these applications. Just use standard techniques, and try to make the system flexible and extensible. The initial versions of the framework will probably be *white-box frameworks*.

- **White-box Framework:** You have started to build your second application. Some frameworks rely heavily on inheritance (white-box), others on polymorphic composition (black-box). Which should you use? Polymorphic composition requires knowing what is going to change, hence, use inheritance first. Build a white-box framework by generalizing from the classes in the individual applications. Use patterns like Template Method and Factory Method [GHJ95] to increase the amount of reusable code in the superclasses from which you are inheriting. At this point, you should not be worrying about the semantics of inheritance, just the ability to reuse existing code. Once you have a working framework, you can start using it. This will show you what is likely to change and what is not. Later, immutable code can be encapsulated and parameterized by converting the framework into a *black-box framework*. While developing the subsequent applications, whenever you find that you need a class that is similar to a class that you developed in a prior application, create a subclass and override the methods that are different (programming-by-difference [JoF88]). After you've made a couple of subclasses, you will recognize which parts you are consistently overriding and which parts are relatively stable. At that point, you will be able to create an abstract class to contain the common portions. Also, while overriding methods, you will probably discover that certain methods are almost the same in all the subclasses. Again, you should factor out the parts that change into a new method. By doing this, the original methods will all become identical and can be moved into the abstract class. As you develop additional applications, you should begin to build a *component library*. A *black-box framework* addresses the same problem, but in a different context.
- **Component Library:** You are developing the second and subsequent examples based on the *white-box framework*. How do you avoid writing similar objects for each instantiation of the framework? A framework with a good library of concrete components will be easier to use than one with a small library. Up front, it is difficult to tell which components users of the framework will need. Some components are problem-specific while others occur in most solutions. Start with a simple library of the obvious objects and add additional objects as you need them. These objects are the concrete subclasses of the abstract classes that make up the framework. Abstract classes generally lie in the framework. Concrete classes generally lie in applications. The component library can be created by accumulating all the concrete classes created for the various applications derived from the framework. In the long run, a class should only be included in the component library if it is used by several applications, but in the beginning, you should put all of them in. If a component gets used a lot, it should remain in the library. If it never gets reused, it gets thrown out. Many components will get refactored into smaller subcomponents by later patterns and disappear that way. As components get added to the library, you will begin to see recurring code that sets of components share. You should look for *hot spots* in your framework where the code seems to change from application to application.
- **Hot Spots:** You are adding components to your *component library*. As you develop applications based on your framework, you will see similar code being written over and over again. How do you eliminate this common code? Ideally, the varying code should be encapsulated within objects whenever possible, since objects are easier to reuse than individual methods. With the code encapsulated, variation is achieved by composing the desired objects rather than creating subclasses and writing methods. By gathering the code that varies into a single location (object) it will both simplify the reuse process and show users where the designers expect the framework

to change. To encapsulate the hot spots, you will often have to create *finer grained objects*. Often these fine-grained objects will cause your framework to become more *black-box*.

- **Pluggable Objects:** You are adding components to your *component library*. Most of the subclasses that you write differ in trivial ways; e.g., only one method is overridden. How do you avoid having to create trivial subclasses each time you want to use the framework? New classes, no matter how trivial, increase the complexity of the system. If the differences between subclasses is trivial, creating a new subclass just to encapsulate the small change is overkill. Adding parameters to the instance creation protocol provides for reuse of the original class without resorting to programming. Creating pluggable objects is one way to encapsulate the *hot spots* in your framework. The parameters can be automatically supplied by a *visual builder*.
- **Fine-grained Objects:** You are refactoring your *component library* to make it more reusable. How far should you go in dividing objects into smaller objects? Continue breaking objects into finer and finer granularities until it doesn't make sense to do so any further. Since frameworks will ultimately be used by domain experts (non-programmers) you will be providing *language tools* to create the compositions automatically. Therefore, it is more important to avoid programming. Anywhere in your component library where you find classes that encapsulate multiple behaviors that could possibly vary independently, create multiple classes to encapsulate each behavior. Wherever the original class was used, replace it with a composition that recreates the desired behavior. This will reduce code duplication, as well as the need to create new subclasses for each new application. As the objects become more fine-grained, the framework will become more *black-box*.
- **Black-box Framework**: You are developing *pluggable objects* by encapsulating *hot spots* and making *fine-grained objects*. Some frameworks rely heavily on inheritance, others on polymorphic composition. Which should you use? Use inheritance to organize your component library and composition to combine the components into applications. Essentially, inheritance will provide a taxonomy of parts to ease browsing and composition will allow for maximum flexibility in application development. When it isn't clear which is the better technique for a given component, favor composition. Convert inheritance relationships to component relationships. Pull out common code in unrelated (by inheritance) classes and encapsulate it in new components. By organizing the component library in this manner, we support the creation of a *visual builder* that allows the library to be browsed and compositions to be created graphically.
- **Visual Builder:** You have a *black-box framework*. You can now build an application entirely by connecting objects of existing classes. The behavior of your application is determined entirely by how these objects are interconnected. A single application consists of two parts. The first part is the script that connects the objects of the framework together and then ”turns them on”. The second part is the behavior of the individual objects. The framework provides most of the second part, but the application programmer must still provide the first part. The connection script is usually very similar from application to application with only the specific objects being different. How do you simplify the creation of these scripts? Make a graphical program that lets you specify the objects that will be in your application and how they are interconnected. It should generate the code for an application from its specification. Since the code is basically just a script, the tool can generate it automatically. The tool will also make the framework more user-friendly by providing a graphical interface to it. At this point, domain experts can create applications by simply manipulating images on the screen. Only in rare cases should new classes have to be added to the framework. Now you have developed a visual programming language. Note that this implies that you will need *language tools*, just like any other language.

- **Language Tools:** You have just created a *visual builder*. It creates complex composite objects. How do you easily inspect and debug these compositions? Create specialized inspecting and debugging tools. Find the portions of your framework that are difficult to inspect and debug. Create specialized tools to navigate and inspect the compositions.

4.2.2 Stepwise Specialized Frameworks

In their paper ”Designing a Framework by Stepwise Generalization” [KoM95] Koskimies and Mössenböck suggest that the design of a framework shall proceed in two phases. The first phase is called problem generalization. It starts from a single example problem which is then generalized in a sequence of steps into the most general (sensible) form. The first phase makes use of the following questions to find the next generalization:

1) Which concepts of the problem domain exist in variants and should be treated uniformly? For example, a graphical user interface has buttons, check boxes and sliders, all of which should be displayable, moveable, resizeable, etc. Instead of distinguishing between these variants, the framework should generalize them into the abstract concept of a user interface item that represents them all.
2) Is it possible to find a concrete concept that can be generalized into a more abstract one, thus making the framework reusable in situations for which it was not originally intended?

The second phase is called framework design. During this phase the generalization levels of the original example are considered in reverse order leading to a framework implementation for each level. In other words, a framework for the most generalized level of the original example is implemented first. Now the implementation of the framework for level *i* can be obtained by using the framework for the next general level *i+1*, like in figure 2.

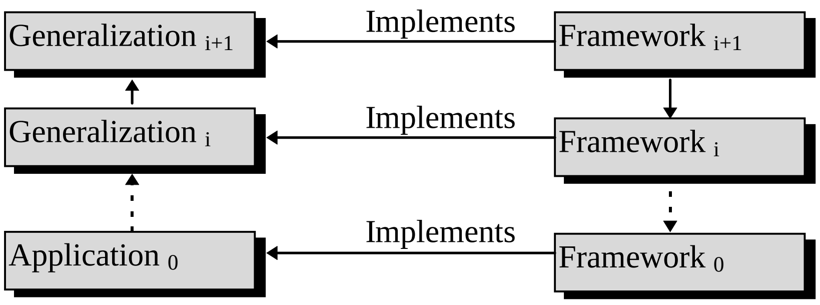


Figure 2. Obtaining frameworks by stepwise specialization.

The result is a hierarchy of more and more refined frameworks. The second phase makes use of general design experience and domain knowledge to find the hot spots in each framework. The following questions can be asked for each obtained example level:

3) Which parts of the system might change? For instance, some users might want to replace the formatting strategy of a text editor. This change can be anticipated by adding a formatter class with a default implementation to the framework. The default formatting can then be replaced by customized formatting strategies if desired, without changing any interfaces.
4) Where might a user want to hook custom code into the framework?

The last step in the second design phase is to implement the original example problem with the resulting framework, thus demonstrating that the framework is applicable in this representative case. The two-phase method avoids early commitment to application-specific classes and architectures, making it possible to employ general design patterns at early development stages.

5 Some Existing Frameworks

In this chapter some of the existing frameworks are introduced in order to illustrate how different and wide subject a framework can be. Some of the frameworks presented here are commonly known, like ET++, whereas some are less known, like the D2D extension framework. Of course this chapter is not a complete description of these frameworks; such a presentation would not be possible here. There is also a large number of frameworks which are not mentioned here at all.

MVC

The Smalltalk-80 user interface framework called Model/View/Controller (MVC) was developed around 1980 [Gol84, KrP88, LaP91]. It was the first widely used framework and it showed that object-oriented programming is well-suited for implementing graphical user interfaces. MVC uses a paradigm in which the input, the actual system core, and the visual feedback are explicitly separated and handled by three types of object. The model object manages the application specific data and behavior. The view object manages the output views. The controller object interprets the mouse and keyboard inputs from the user, controlling the model and/or the view to change as appropriate. The MVC behavior is then inherited, extended, and modified as necessary to provide a flexible and powerful system [Bur92].

FOIBLE

Frameworks can be built on top of other frameworks by sharing abstract classes. FOIBLE, for example, is built on top of Model/View/Controller (MVC) framework discussed above. FOIBLE is a framework for building device programming systems in Smalltalk; it lets the user edit a picture consisting of a collection of interconnected devices. By editing the picture the user actually decides how the devices are handled. FOIBLE uses the MVC framework to implement the editor, but adds tools and foibles to implement the semantics of the picture and the visual representation of components [Eri87, JoF88].

ET++

ET++ is a homogeneous object-oriented class library integrating user interface building blocks, basic data structures, and support for object input/output with high level application framework components. The work on ET++ started in 1987 at the University of Zurich. In 1990 the developers of ET++ moved to UBILAB (Union Bank of Switzerland’s Informatics Laboratory). The main goals in designing ET++ have been the desire to substantially ease the building of highly interactive applications with consistent user interfaces following the well known desktop metaphor, and to combine all ET++ classes into a seamless system structure. Experience has proven that writing a complex application based on ET++ can result in the reduction in source code size of 80% and even more when compared to the same software written on top of a conventional graphic toolbox. ET++ is implemented in C++ and runs under UNIX(TM) and either SunWindows(TM), NeWS(TM), or the X11 window system [WGM89, EgG92]

ET++ SwapsManager

One example of a domain specific framework is the ET++ SwapsManager developed by the Union Bank of Switzerland. It is based on the ET++ application framework and it was developed as a calculation engine for financial trading software. The ET++ SwapsManager provides a model for swaps trading that demostrates the advantages of applying framework technology to financial engineering. Exploratory prototyping was used during the development process. This was because the developers didn't know much about financial engineering and the trading specialists didn't know what kind of tools they want. One goal in the ET++SwapsManager project was also to acquire experience with the idea of design patterns [EgG92].

ITHACA

Intelligent Tools for Highly Advanced Commercial Applications (ITHACA) is an Esprit II Project, started in January 1989 [ANM90]. The goal of ITHACA was to produce a complete object-oriented application development environment that can be easily adapted to various application domains. The Ithaca Application Development Environment (ADE) is being developed as an integrated environment where the Software Information Base (SIB) contains reusable components and development information, and where a set of development tools supports the application developer in finding and combining components to produce a specific application. These tools include a browser for SIB, a debugger for classes, a requirements collection and specification tool (RECAST), and a visual scripting tool (Vista, recall chapter 3.3). In ITHACA, it is considered that application engineers have stocked the SIB with a library of classes and application frames clustered by application domains. The application developer is expected to proceed in the following steps to produce a specific application [FNP92, MeN93]:

1) Select an application frame: Using only a rough sketch of the application requirements, the developer searches and browses the SIB to find an application frame corresponding to the application domain and to the requirements of the application being designed.
2) Select useful classes: The application frame drives requirement collection and specification according to preexisting, generic specifications and designs, thus guiding the developer in the selection of reusable classes.

3) Tailor classes: The selected classes are incrementally modified using design suggestions given by the specification classes; tailoring occurs by supplying parameters or by modifying class behavior through inheritance.
4) Script application: The selected design classes are linked together by means of a script that specifies how the objects will cooperate to implement the required application.
5) Monitor behavior and develop continuously.

Choices

Choices is an operating system framework developed at the University of Illinois at Urbana-Champaign [CRJ87]. This system-level framework composes of subframeworks for process management, virtual memory management, file systems, and networking. For example, the virtual memory framework of Choices [RuC89] provides an abstract design of a virtual memory system that can be customized to make a particular concrete system. Here the virtual memory is a logical technique that provides the illusion of having more memory than is actually on the computer by using physical memory (e.g., a hard disk) as a cache. The Choices virtual memory framework, like all the other parts of Choices, has changed several times. Typically, a subframework was reorganized to simplify it and to make it easier to understand and reuse it [JoR91].

San Francisco

IBM’s Commercial Shareable Frameworks, also known as the project San Francisco, is a set of flexible application business frameworks. San Francisco is being developed in collaboration with several hundred international Independent Software Vendors (ISVs); they are working with IBM to design, develop, and validate frameworks, to create development tools, and to develop integrated applications on top of the San Francisco frameworks. The goal of the project is to reduce the complexity, expense, and time-to-market for ISVs to build customized multi-platform business applications. San Francisco provides a set of related classes of business objects that provide the infrastructure and the core business processes in an application area; software providers will extend this model with their user interface, country- and industry-specific requirements, business rules, competitive differentiators, and complementary application functions. The extension points (hot spots) at which application providers will add or replace business logic are carefully defined during the framework’s design phase. The combination of the prefabriciated common business objects and the core business processes will approximate 40% of a typical working application, the rest is added by the user [IBM96a, IBM96b].

MacApp

MacApp is an application framework for creating Macintosh applications. Programmers can use MacApp to build applications while freely inheriting characteristics common to all Macintosh applications [Mac96]. There are over a thousand applications for Macintosh developed with MacApp. The domain of these applications vary from drawing tools to network management software.

POSC

The Petrochemical Open Systems Consortium (POSC) was formed by oil and gas companies who realized that their competitive strengths lay in core business areas, not in computing technology. So far petroleum companies have used disparate data formats, different database systems, in-house developed and purchased applications that do not communicate with each other, and workstations with differing operational requirements and application interfaces. These problems affect software and hardware vendors as they develop and market to this industry segment. They face difficult economic trade-offs of what languages, communication protocols, software and hardware interfaces to develop and/or support. As a result, petroleum companies are cooperatively developing a framework that will allow third party vendors to build custom applications to a standard business model [Tal94, POS96].

SEMATECH CIM Framework

A Computer-Integrated Manufacturing (CIM) framework has been developed under contract from SEMATECH (SEMiconductor Manufacturing TECHnology). The goal is to reduce cost to develop, integrate, modify, and support manufacturing systems in multi-industry manufacturing areas. The project was driven by the need to reduce cycle time and cost, and increase flexibility in the production process. With the CIM framework, suppliers can provide custom plug and play applications that extend the framework’s model [Tal94, SEM96].

D2D extension framework

D2D is a two-dimensional weather forecaster workstation and it is the central element of the WFO-Advanced forecaster system [MaW96]. With the D2D extension framework local forecasters and programmers in National Weather Service offices (NWS) can develop their own extensions to the display system. Such extensions enable D2D users to display special kinds of data, interactively locate information in geographic databases, compute storm directions, etc. An advantage of using a framework is that extension developers do not need to make architectural decisions for their extensions. Instead they derive new classes and override the virtual methods as necessary [Kel97].

DOVER

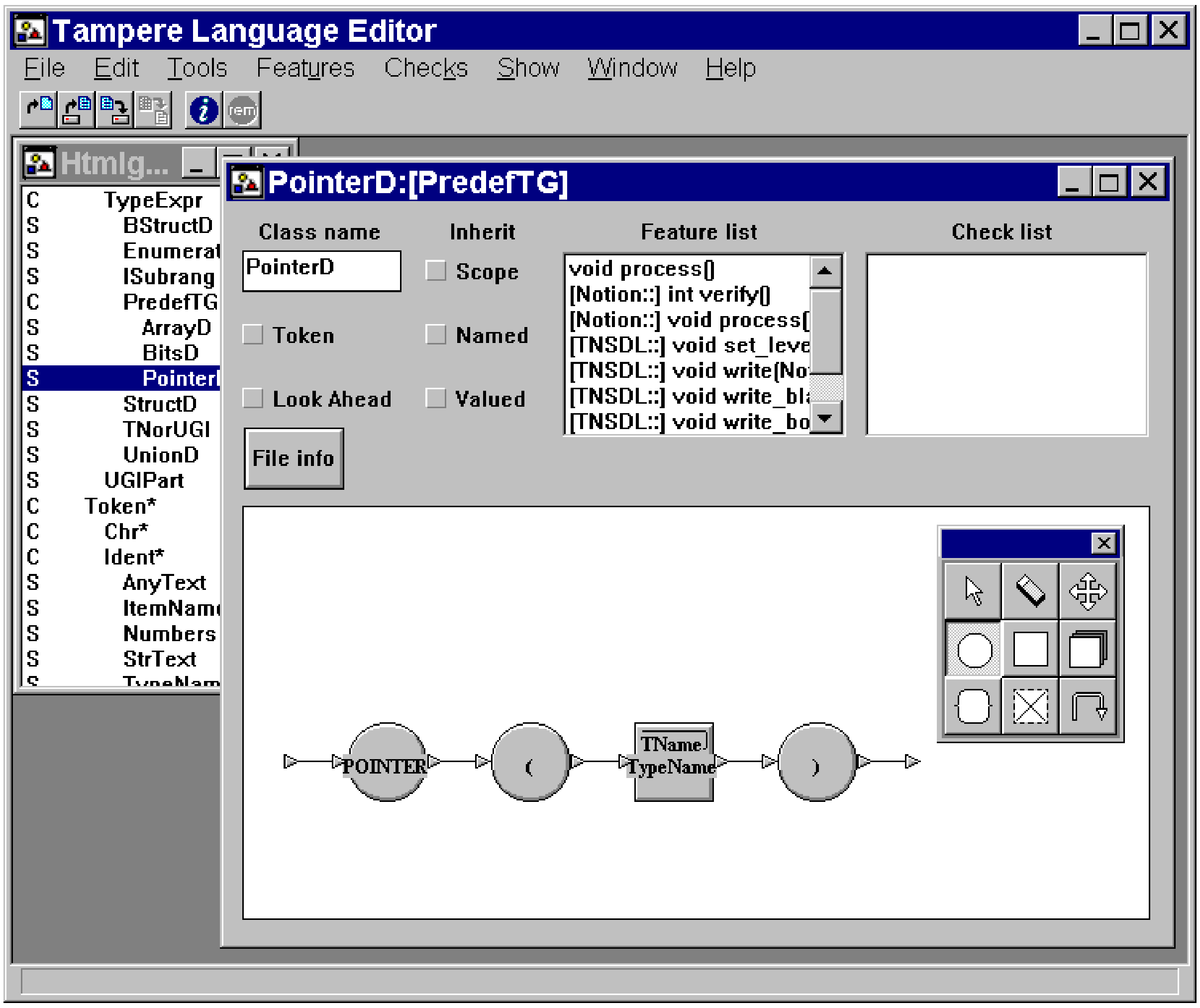
The system development tradition at Karlskronavarvet has been to start the development process almost from scratch in every new development project, leading to high development costs. For that reason, development of an object-oriented framework for vessel control systems (DOVER) has been constructed. The resulting framework consists of basic, reusable building blocks, suitable for designing control systems in this particular application area. The framework has lead to better productivity and better estimation of development time. This has been observed when developing several slightly different prototype systems using the framework. The code size for some systems has been reduced by almost 40 %. Also, for the development of a new system 80 % of the code can be reused. The learning curve for developers and maintenance teams is reduced, too; this is because now the developers only have to know about one base system. The programming is done faster and the readability of the code is improved. This has reduced software development costs. With the framework a significant part of the system has already been tested before, which in turn provides for higher quality [Kar96].

FACE

Framework Adaptive Composition Environment (FACE) is an approach for building and using frameworks. It is under construction in the Software Composition Group of the University of Berne. The goal of the project is to build a visual software composition environment where developers can create applications via graphical user interface. That is, prefabricated components are presented to the user and they can be linked in specific ways. FACE includes also a typechecking mechanism, which prevents incorrect component compositions. In the FACE approach developing an application covers two levels, namely composing a set of objects and composing a ”schema” for an application. A schema descripes possible structures and cooperations of run-time objects in the target application. A basic element of a schema is a class-component. The class-component is basically an object that has a certain composition interface defining how this particular component can be connected to other components. All FACE components are black-box entities, thus, a schema is specialized through parametrization and composition by the application developer. The chosen approach is application domain independent, however, the current choice of granularity is focused on realizing small to medium-size applications. FACE could be, e.g., used for building graphical design tools, editors, and simulators. Note however, that real experiences in this direction are still missing [Mei96, Rie96].

TaLE

Tampere Language Editor (TaLE) is a framework system for developing language implementations in an object-oriented programming environment [Jär95, JKN95, Hau96]. In TaLE individual language structures are implemented as C++ classes and they are accessible indirectly through high level graphical tools, like in figure 3. TaLE generates most of the needed C++ code automatically depending on the user selections, while the user defines the syntax and the semantics of his/her target language. Then the generated language implementation is compiled with a C++ compiler. The result is an executable program, called language processor that can make user-defined actions based on the given input text. Currently TaLE has been used to implement a source-to-source converter translating programs from PL/M to C [Nii95b]. Also a simple language implementation converting the type definitions of TeleNokia Specification and Description Language (TNSDL) into HTML has been implemented. This HTML coded WWW-page can then be viewed with a WWW-browser, like Netscape [Hau96].



C	TypeExpr
S	BStructD
S	Enumerat
S	ISubrang
C	PredefTG
S	ArrayD
S	BitsD
S	Pointer
S	StructD
S	TNorUGI
S	UnionD
S	UGIPart
C	Token*
C	Chr*
C	Ident*
S	AnyText
S	ItemNam
S	Numbers
S	StrText

Class name: PointerD

```
void process()
[Notion::] int verify()
[Notion::] void process(
[TNSDL::] void set_leve
[TNSDL::] void write(No
[TNSDL::] void write_bl
[TNSDL::] void write_bo
```

Figure 3. PointerD class is opened and the user defines the syntax for that class.

6 Conclusion

Frameworks are reusable designs used to improve application developers’ productivity and target application’s maintainability. Frameworks have many advantages making them attractive; less code to program, test, and debug, faster development cycle, etc. The main obstacle of using a framework is the learning curve. To reduce this curve good documentation and effective tools should be provided with the framework.

A framework can make the application development much easier and cheaper. However, developing a framework itself is not easy; it is an iterative process which needs both domain and design experience. The development process usually starts with a few examples which are then generalized to find out similarities. Abstractions and hot spots must be found in order to make the framework flexible. Design patterns can be used to ease this process; they provide abstractions proved to work in the past. They can also be used to document the framework.

References

[AIS77] Alexander C., Ishikawa S., Silverstein M., Jacobson M., Fiksdahl-King I., Angel S.: A Pattern Language. Oxford University Press, New York, 1977.

[ANM90] Ader M., Nierstrasz O., McMahon S., Müller G., Pröfrock A-K.: The ITHACA Technology: A Landscape for Object-Oriented Application Development. In: Proc. of ESPRIT'90 Conference, pp. 31-51. Kluwer Academic Publishers, 1990.

[ArG96] Arnold K., Gosling J.: The Java Series: The Java Programming Language. Addison-Wesley, 1996.

[Boo94] Booch G.: Designing an Application Framework. Dr. Dobb's Journal, vol. 19, no.2, February 1994.

[Bur92] Burbeck S.: ”Application Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC).” [http:// sw1.informatik.uni-hamburg.de/ ~strunk/ smalltalk/ mvc.html], 1992.

[CIM92] Campbell R., Islam N., Madany P.: Choices, Frameworks and Refinement. Computing Systems, vol. 5, no. 3, 1992.

[CRJ87] Campbell R., Russo V., Johnston G.: The Design of a Multiprocessor Operating System. In: Proc. of USENIX C++ Workshop, 1987.

[Dei94] Deitel H.: C++ How to Program. Prentice-Hall, 1994.

[DMN97] Demeyer S., Meijler T., Nierstrasz O., Steyaert P.: Design Guidelines for Tailorable Frameworks. Draft manuscript, submitted to the Communications of the ACM October'97 Issue on ”Object-Oriented Frameworks”, January 1997.

[EgG92] Eggenschwiler T., Gamma E.: ET++ SwapsManager: Using Object Technology in the Financial Engineering Domain. In: Proc. of OOPSLA'92, ACM SIGPLAN Notices, Vol. 27, No. 10, 1992, pp. 166-177.

[Eri87] Ericson S.: FOIBLE: A Framework for Object-Oriented Interactive Box and Line Environments. Master's thesis, University of Illinois at Urbana-Champaign, 1987.

[FNP92] Fugini M., Nierstrasz O., Pernici B.: Application Development Through Reuse: The ITHACA Tools Environment. ACM SIGOIS Bulletin, vol. 13, no. 2, August 1992, pp. 38-47.

[GHJ95] Gamma E., Helm R., Johnson R., Vlissides J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[Gol84] Goldberg A.: Smalltalk-80: The Interactive Programming Environment. Addison-Wesley, 1984.

[Hau96] Hautamäki J.: Language Implementation with TaLE. Master’s thesis, Department of Computer Science, University of Tampere, October 1996.

[IBM96a] IBM Inc.: ”IBM Shareable Frameworks.”, [http:// www.softmall.ibm.com/ sf/ overview.html]. August 1996.

[IBM96b] IBM Inc.: ”IBM Shareable Frameworks Technical Overview.” [http:// www.softmall.ibm.com/ sf/ techov.html]. October 1996.

[JKN95] Järnvall E., Koskimies K., Niittymäki M.: Object-Oriented Language Engineering with TaLE. Object Oriented Systems, vol. 2, 1995, pp. 77-98.

[JoF88] Johnson R., Foote B.: Designing Reusable Classes. Journal of Object-Oriented Programming, vol. 1, no. 2, June 1988, pp. 22-35.

[Joh92] Johnson R.: Documenting Frameworks using Patterns. In: Proc. of OOPLSA ‘92, ACM SIGPLAN Notices, vol. 27, no. 10, 1992, pp. 63-76.

[Joh93] Johnson R.: How to Design Frameworks. In: Tutorial Notes of OOPSLA’93, 1993. [ftp:// st.cs.uiuc.edu/ pub/ papers/ frameworks/ OOPSLA93-frmwk-tut.ps].

[Joh96] Johnson R.: ”Frameworks Home Page.” [http:// st-www.cs.uiuc.edu/ users/ johnson/ frameworks.html]. August 1996.

[JoR91] Johnson R., Russo V.: Reusing Object-Oriented Design. Technical Report UIUCDCS 91-1696, University of Illinois, 1991.

[Jär95] Järnvall E.: Olioperustainen kielentoteutusjärjestelmä TaLE. Lisensiaatin tutkimus, Tietojenkäsittelyopin laitos, Tampereen yliopisto, syyskuu 1995.

[Kar96] Karlskronavarvet AB: ”Development of An Object-Oriented Framework for Vessel Control Systems.” [http:// www.fend.es/ services/ aeneid/ docs/ 10496.htm]. 1996.

[Kel97] Kelly S.: ”An Object-Oriented Framework for Local Extensions to the WFO-Adcanced Forecaster Display Workstation, D2D.” [http:// www-sdd.fsl.noaa.gov/ ~fxa/ publications/ 13th_IIPS_97/ KellyExt.IIPS97.html]. To be presented at the 13th International Conference on Interactive Information and Processing Systems (IIPS) for Meteorology, Oceanography, and Hydrology, February 1997, Long Beach, CA, USA.

[KoM95] Koskimies K., Mössenböck H.: Designing a Framework by Stepwise Generalization. In: Proc. of ESEC’95, Lecture Notes in Computer Science 989, Spinger-Verlag, 1995, pp. 479-497.

[KrP88] Krasner G., Pope S.: A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. Journal of Object-Oriented Programming, vol. 1, no. 3, 1988, pp. 26-49.

[LaP91] LaLonde W., Pugh J.: Inside Smaltalk, Volume II. Prentice Hall, 1991.

[Mac96] Apple Computer Inc.: ”MacApp: The Industrial-Strength Framework.” [http:// www.devtools.apple.com/ macapp/ MAdescription.html], 1996.

[Mat96a] Mattsson M.: ”Frameworks FAQ’s.” [http:// www.ide.hk-r.se/ ~michaelm/ fwpages/ fwfaqs.html]. 1996.

[Mat96b] Mattsson M.: ”An Object-Oriented Framework Bibliography.” [http:// www.pt.hk-r.se/ ~michaelm/ fwbibl.html]. 1996.

[MaW96] MacDonald A., Wakefield J.: WFO-Advanced: An AWIPS-like Prototype Forecaster Workstation. In: Proc. of Twelfth International Conference on Interactive Information and Processing Systems (IIPS) for Metorology, Oceanography, and Hydrology, 1996. [http:// www-sdd.fsl.noaa.gov/ ~fxa/publications/ 12th_IIPS_96/ MacDonald-Wakefield.IIPS96.html].

[Mei96] Meijler T.: ”FACE.” [http:// iamwww.unibe.ch/ ~scg/ ComponentModels/ face.html]. March 1996.

[MeN93] Mey V., Nierstrasz O.: The ITHACA Application Development Environment. In: Visual Objects (ed. D. Tsichritzis), Centre Universitaire d’Informatique, University of Geneva, July 1993, pp. 267-280.

[Mey90a] Mey V.: Vista User’s Guide. ITHACA Report ITHACA.CUI.90.E.4.#2, December 1990.

[Mey90b] Mey V.: Vista Implementation. ITHACA Report ITHACA.CUI.90.E.4.#1, December 1990.

[Nii95b] Niittymäki M.: Automated Construction of Source-to-Source Translators. Licentiate thesis, Department of Computer Science, University of Tampere, September 1995.

[POS96] Petrotechnical Open Software Corporation (POSC): ”POSC Frequently Asked Questions: What is POSC?” [http:// posc.org/ faq_whats_posc.html], August 1996.

[Pre94] Pree W.: Design Patterns for Object-Oriented Software Development. Addison-Wesley, 1994.

[RBP91] Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorenson W.: Object-Oriented Modeling and Design. Prentice Hall, 1991.

[Rie96] Rieger M.: ” An Implementation of the FACE Datamodel.” [http:// iamwww.unibe.ch/ ~rieger/ liz.html]. September 1996.

[RoJ96] Roberts D., Johnson R., Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks. In: Proc. of PLoP’96, Third Annual Conference on the Pattern Languages of Programs, 1996. [http:// www.cs.wustl.edu/ ~schmidt/ PLoP-96/ final.html].

[RuC89] Russo V., Campbell R.: Virtual Memory and Backing Storage Management in Multiprocessor Operating Systems Using Class Hierarchical Design. In: Proc. of OOPSLA’89, ACM SIGPLAN Notices, vol 24., no. 10, 1989, pp 267-278.

[SEM96] SEMATECH Inc.: ”CIM Framework.” [http:// www.sematech.org/ public/ cim-framework/ home.html], August 1996.

[Str91] Stroustrup B.: The C++ Programming Language. Addison-Wesley, 1991, Second Edition.

[Tal93] Taligent Inc.: ”Leveraging Object-Oriented Frameworks.” [http:// www.taligent.com/ Technology/ WhitePapers/ LeveragingFwks/ LeveragingFrameworks.html]. A Taligent White Paper, 1993.

[Tal94] Taligent Inc.: ”Building Object-Oriented Frameworks.” [http:// www.taligent.com/ Technology/ WhitePapers/ BuildingFwks/ BuildingFrameworks.html]. A Taligent White Paper, 1994.

[WGM89] Weinand A., Gamma E., Marty R.: Design and Implementation of ET++, a Seamless Object-Oriented Application Framework. Structured Programming, Vol. 10, No. 2, July 1989, pp. 63-87.

[VilA97] Viljamaa A.: Application Frameworks in the Java Environment. Report, Department of Computer Science, University of Helsinki, 1997.

[VilJ97] Viljamaa J.: Tools Supporting the Use of Design Patterns in Frameworks. Report, Department of Computer Science, University of Helsinki, 1997.