Framework Documentation
A Minimalist Approach

Universidade do Porto
Faculdade de Engenharia
FEUP

Department of Electrical and Computer Engineering

September 2003
Framework Fundamentals

An object-oriented framework is a cohesive design and implementation artifact. Frameworks serve to implement larger-scale components, and are implemented using smaller-scale classes [Riehle, 2000].

Since its creation at the end of the 1980s, the concept of object-oriented frameworks has attracted a lot of attention from researchers and software engineers, resulting in many frameworks being developed in industry and academia, covering different application domains. The benefits from frameworks include reduced time to market and improved compatibility and consistency [Taligent Press, 1994; Fayad and Schmidt, 1997a; Fayad et al., 1999].

This chapter presents the fundamental characteristics of the framework concept. It reviews common terminology, associated object-orientation concepts, the key benefits of reusing frameworks, and the role of frameworks in the context of object-oriented software architecture. The chapter concludes with a brief history of frameworks, from the early Simula frameworks (1960s) till present.
2.1 What Is a Framework?

A framework is a reusable design together with an implementation. It consists of a collection of cooperating classes, both abstract and concrete, which embody an abstract design for solutions to problems in an application domain [Johnson and Foote, 1988; Deutsch, 1989; Campbell et al., 1991; Cotter and Potel, 1995; Gamma et al., 1995; Lewis et al., 1995; Fayad and Schmidt, 1997b; Fayad et al., 1999].

But frameworks are more than just collections of classes. Frameworks are also architectural. A framework defines the overall application structure, its partitioning into classes and objects, the key responsibilities thereof, how the classes and objects collaborate, and the thread of control. So, frameworks dictate the architecture of the applications we build with them, but still leave enough design space to accommodate particular solutions. By predefining design parameters that are invariant in an application domain, frameworks help application developers get the key architectural aspects right from the beginning, letting them concentrate on the specifics of their applications.

When using a framework, we reuse not only analysis and design but also implementations. With a framework, developers can build applications by extending or customizing only some parts, while reusing framework implementations of the rest and retaining the original design.

2.2 Frameworks and Reuse

The simple yet powerful vision of software reuse was introduced in 1968 [Naur and Randell, 1968] as a means to reduce the time and effort required to build and maintain high-quality software systems. In a broad sense, software reuse is the process of creating new software systems starting from existing software artifacts rather than building them from scratch.

Reuse does not happen by accident. We need to plan to reuse software, and to look for software to reuse. Reuse requires the right attitude, tools and techniques [Johnson and Foote, 1988]. Object-oriented frameworks are one reuse technique, actually a powerful one that enable large-scale reuse.

2.2.1 Reuse Techniques

There are several techniques for software reuse, each possibly using different artifacts. Reusable software artifacts include source code fragments, design structures, abstract specifications, and documentation, to mention a few. Tools and techniques to support software reuse are usually categorized in
compositional and generative approaches. While compositional approach is based on reusing software artifacts, the generative approach is based on reusing software development processes, often embodied in tools that help automate them. Source code components and application generators are just two examples of such approaches, respectively [Krueger, 1992].

A reuse technique must support one or all of the four important activities of software reuse, namely: abstracting artifacts; selecting artifacts, which includes classifying, finding and understanding them; specializing artifacts; and integrating artifacts [Biggerstaff and Richter, 1989].

The determination of the best reuse technique is often difficult to do, as it depends a lot on the specificities of the project at hands. As an intuitive gauge to compare the effectiveness of different reuse techniques, Krueger used the notion of cognitive distance [Krueger, 1992]. He informally defines it as the amount of intellectual effort that must be expended in developing a software system to go from the initial conceptualization to a specification expressed in abstractions of the reuse technique, and from these to an executable system.

An ideal reuse technique should let us quickly find components that exactly fit our needs, are ready to use without being customized, and don’t force us to learn how to use them. The developer’s ability to reuse software is limited primarily by his ability to reason in terms of the abstractions used by the reuse technique. In other words, the cognitive distance between informal reasoning and the abstract concepts of the technique must be small. Natural, succinct and high-level abstractions describing artifacts in terms of “what” they do, rather than “how” they do it, are thus very important for effective software reuse. From all the existing reuse techniques, the reuse of software architectures is probably the technique that comes closest to this ideal.

2.2.2 How Object-Orientation Leverages Software Reuse?

Historically, in the 1960s, reusable software components have been procedural libraries. In 1967, with their language Simula 67 [Dahl et al., 1970], Dahl and Nygaard have introduced most of the key concepts of object-oriented programming, namely objects, classes and inheritance, and with these concepts one of the main paradigms of programming have started, the object-oriented programming.

Simula concepts have been important in the discussion of abstract data types and models for concurrent program execution, starting in the early 1970s. Alan Kay’s group at Xerox PARC used Simula as a platform for their development of the first language versions of Smalltalk, in the 1970s, extending object-oriented programming importantly with the integration of
Object-oriented programming is today becoming the dominant style for implementing complex applications involving a large number of interacting components. Among the multitude of object-oriented languages are Smalltalk, Object Pascal, C++, Common Lisp Object System (CLOS), Eiffel, BETA, and SELF. In particular, the Internet-related Java (developed by Sun) has rapidly become widely used in the last 1990s.

With the integration of data and operations into objects and classes, reusability has increased. The classes were packaged together into class libraries, often consisting of classes for different data structures, such as lists and queues. Class libraries were further structured using inheritance to facilitate the specialization of their classes. As a result, class libraries became capable of delivering software reuse beyond traditional procedural libraries.

Object-oriented programming languages combine features, such as data abstraction, polymorphism and inheritance, that encourage the reuse of existing code instead of writing new code from scratch. These features are detailed later in Section 2.4.1 (p. 29).

Taking advantage of these features, object-oriented languages promote the development of class libraries, which, like the procedural libraries, are mainly focused on reuse of code. But code reuse has limitations, working best when the domain is narrow, well understood, and the underlying technology is very static. In the long run, reusing the design of an application is probably more beneficial in economical terms than reusing the implementation of any of its components [Biggerstaff and Richter, 1987], because design is the main intellectual content of software and it is by far more difficult to create and re-create then code [Deutsch, 1989].

Although object-orientation has started with object-oriented programming languages, it is more than object-oriented programming. Object-orientation covers also earlier phases of programming, such as analysis and design. Using a small set of concepts (objects, classes, and their relationships) developers can model an application domain (analysis), define an architecture to represent that model on a computer (design), and implement that architecture to let a computer execute the model (programming) [Booch, 1994].

As a whole, object-orientation introduced in software development more qualities that favor software reuse, namely, problem-orientation, resilience to
evolution, and domain analysis.

- **Problem-orientation.** The object-oriented models produced during analysis are all described in terms of the problem domain, which can be mapped directly to object-oriented concepts, such as classes, objects and relationships. This seamlessness from analysis to programming models makes them simpler to communicate between users and developers, and enables the delivery of better software products [Hoydalsvik and Sindre, 1993].

- **Resilience to evolution.** In an application domain, processes change more often than the entities. As object-oriented models are structured around the entities, they are more stable to changes and therefore less resilient to evolution [Meyer, 1988].

- **Domain analysis.** Object-oriented analysis is naturally extensible to domain analysis, a broader and more extensive kind of analysis that tries to capture the requirements of the complete problem domain, including future requirements [Schafer et al., 1994].

Taking advantage of all these qualities, reusability appeared as one of the great promises of object-orientation, based on the reuse of code through inheritance. But the efforts only provided reuse at the level of small-scale components, usable as primitive building blocks of new applications. Neither object-orientation nor class libraries made possible the reuse of large-scale components. This understanding led to the conception of object-oriented frameworks, a kind of large and abstract application specially designed to be tailored for the development of concrete applications in a particular domain. In the beginning of the 2000s, object-oriented frameworks represent the state-of-the-art in terms of object-oriented reusable products.

### 2.2.3 The Power of Frameworks

Since its conception at the end of 1980s, the appealing concept of object-oriented framework has attracted a lot of attention from many researchers and software engineers. During the 1990s, frameworks have been built for a large variety of domains, such as user interfaces, operating systems, and distributed systems.

A framework can be shortly defined as a reusable design of an application together with an implementation [Johnson and Foote, 1988; Campbell et al., 1991; Lewis et al., 1995; Fayad and Schmidt, 1997b; Fayad et al., 1999]. The definitions for a framework are not consensual and vary from author to author. In few words, a framework can be defined as a semi-complete design and implementation for an application in a given problem domain.
As mentioned before, frameworks are firmly in the middle of the reuse techniques. They are more abstract and flexible (and harder to learn) than components, but more concrete and easier to reuse than a raw design (but less flexible and less likely to be applicable). Frameworks are considered a powerful reuse technique because they lead to one of the most important kinds of reuse, the reuse of design. When compared to other techniques for reusing high-level design, such as templates [Spencer, 1988] or schemes [Katz et al., 1989], frameworks have the advantage of being expressed in a programming language, thereby resulting easier to learn and apply by programmers.

Frameworks and components are cooperating technologies. Software components are “binary units of independent production, acquisition, and development that interact to form a functioning system, with explicit interfaces and context dependencies only. A software component can only be deployed independently and is subject to composition by third parties.” [Szyperski, 1998]. Frameworks provide a reusable context for components, in the form of component specifications and templates for their implementation, thereby making it easier to develop new components.

Frameworks and design patterns are concepts closely related as well, representing two different categories of high-level design abstractions [Johnson, 1992]. A single framework typically encompasses several design patterns. Patterns provide an intermediate level of abstraction between the application level and the level of classes and objects.

A design pattern is commonly defined as a generic solution to a recurring design problem that might arise in a given context [Alexander et al., 1977; Gamma et al., 1995; Buschmann et al., 1996]. The relationships between individual patterns unfold in the application domain naturally and form a high level language, called a pattern language [Alexander 1977]. A pattern language represents the essential design knowledge of a specific application domain, i.e. the experience gained by many designers in solving a class of similar problems. Design patterns and pattern languages are particularly good for documenting frameworks because they capture design experience and enclose meta-knowledge about how flexibility was incorporated. Pattern languages help document the application domain of the framework, the design of the framework in terms of classes, objects and their relationships, and also the specifications of important framework classes.

The combined use of frameworks with patterns and components is very effective, significantly helping to increase software quality and reduce development effort [Fayad et al., 1999].

The benefits of frameworks stem primarily from the levels of code and design reuse being much higher than what is possible with other reuse
technologies, such as code generators and class libraries. In addition to the reusability benefits, other advantages are due to the inversion of control, the modularity and the extensibility that frameworks provide to developers.

In general terms, the benefits from frameworks include higher development productivity, shorter time-to-market and higher quality. However, framework benefits are not necessarily immediate, but only gained over time. As significant productivity gains usually start appearing after multiple uses of the technology, frameworks must be considered a medium-to-long term investment.

The benefits from frameworks impact in many phases of application development, from analysis to maintenance and evolution. During the analysis phase, frameworks help developers reduce the effort usually required to understand the overall application domain, and enable them to focus on the details of the application at hands.

It is during the design, coding, testing and debugging of applications that frameworks have more advantages over traditional application development. Most of the benefits result from the high levels of design and code reuse provided, and also the inversion of control (Section 2.4.3) possible with frameworks:

- provide guidance on application architecture;
- improve programming productivity and quality by reducing the amount of code to design and write;
- improve modularity and understandability by encapsulating volatile implementation details behind stable interfaces;
- promote the development of generic solutions reusable across an application domain;
- and improve application integrability and interoperability due to shared architecture and design.

The benefits of frameworks are not less important at maintenance and evolution phases. The reusability and extensibility possible with frameworks help to decrease the effort of maintenance due to its amortization over many application specific parts. Framework-based applications are also easier to evolve without sacrificing compatibility and interoperability because frameworks provide explicit hook methods that allow applications to modify or extend framework’s behavior.

In summary, through design and code reuse, frameworks help us reduce the amount of design we must create and the lines of code we must write, therefore significantly improving productivity. As a result, not only we can
build applications faster, but also build applications that are easier to maintain and more consistent, because they share a similar structure at all levels of software design.

As software systems evolve in complexity, object-oriented application frameworks are being successfully applied in more application domains and therefore becoming more important for industry and academia. Application frameworks offer software developers an important vehicle for reuse and a means of capturing the essence of successful architectures, patterns, components and programming mechanisms.

Perhaps the best evidence of the power of object-oriented frameworks is reflected on the well-known success of many examples of popular frameworks, such as: Model-View-Controller (MVC) [Goldberg, 1984], MacApp [Schmucker, 1986], ET++ [Weinand et al., 1989], Interviews [Linton et al., 1989], OpenDoc [Feiler and Meadow, 1996], Microsoft Foundation Classes (MFCs) [Prosise, 1999], IBM’s SanFrancisco [Monday et al., 2000], several parts of Sun’s Java Foundation Classes (RMI, AWT, Swing) [Drye and Wake, 1999], many implementations of the Object Management Group’s (OMG) Common Object Request Broker Architecture (CORBA), and Apache’s frameworks (Cocoon, Struts) [Apache, 1999]. Despite the existing difficulties of reusing frameworks, all the above examples of frameworks are playing, directly or indirectly, a very important role in contemporary software development.

### 2.3 Object-Oriented Software Architecture

Software design, and system design in general, take place at different levels. Each level has components, both primitive and composite, rules of composition guiding the construction of non-primitive components, and rules of behavior providing semantics for the system. For software, at least three design levels are usually identified [Shaw and Garlan, 1996]:

- an **architecture level**, where the design issues involve the overall organization of a system as a composition of components, the definition of global control structures, and the assignment of functionality to design elements;

- a **code level**, where the design issues involve algorithms and data structures;

- and an **executable level**, where the design issues involve memory maps, call stacks, register allocations, and machine code operations.
As the size and complexity of software systems increase, the most important design problems are no longer the design of the algorithms and data structures, but instead the design and specification of the overall system structure.

### 2.3.1 What is Software Architecture?

Software architecture is an emergent field of study in software engineering specifically concerned with software design at the architecture level. Its importance to software engineering practitioners and researchers has significantly increased during the 1990s, in response to the growing need for exploiting commonalities in system architectures, on making good choices among design alternatives, and describing high-level properties of complex systems.

According to Webster’s Dictionary, architecture is “the art or practice of designing and building structures...”. The main concern of software architecture is the design and building of structure, and not the individual building blocks that bring the structures into existence.

Abstractly, software architecture describes the components from which systems are built, and the interactions among those components—the connectors. Software architecture may also describe the rules and mechanisms that guide the composition of components and eventual constraints on those rules.

Components at the architecture level can be things such as clients, servers, databases, filters, and layers of a hierarchical system. Examples of connectors range from simple procedure calls to complex protocols, such as client-server protocols, database-accessing protocols, event multicast, and pipes.

### 2.3.2 Architectural Levels

Object-oriented software architecture is particularly interested on the architecture of object-oriented systems, that is, on architectures having objects and classes as their primitive building blocks. Current practice suggests four levels of granularity to describe an object-oriented system: the class level, the pattern level (micro-architecture), the framework level (macro-architecture), and the component level.

**Class level**

At the smallest level of granularity, a system is designed as a set of classes, whose instances cooperate to achieve some sophisticated behavior otherwise impossible with a single object.

A class represents a well defined concept or entity of the domain. An object is an instance of a class, has a state, exhibits some well-defined behavior, and
has a unique identity. The structure and behavior of similar objects are defined in their common class. Whereas an object is a concrete software entity that exists in time and space, a class represents only an abstraction, the essence of an object [Booch, 1994].

For small systems, objects and classes are sufficient means for describing their architecture. However, as a system becomes bigger, more and more classes get involved in its architecture, and higher-level abstractions are needed to help developers cope with the complexity of designing and implementing such systems.

![Figure 2.1 Design elements of object-oriented architectures](image)

Pattern level Immediately above the level of classes, we can use patterns to describe the micro-architectures of a system. A pattern names, abstracts, and identifies the key aspects of a design structure commonly used to solve a recurrent problem.
Succinctly, a pattern is a generic solution to a recurring problem in a given context [Alexander et al., 1977]. The description of a pattern explains the problem and its context, suggests a generic solution, and discusses the consequences of adopting that solution. The solution describes the objects and classes that participate in the design, their responsibilities and collaborations.

The concepts of pattern and pattern language were introduced in the software community by the influence of the Christopher Alexander's work, an architect who wrote extensively on patterns found in the architecture of houses, buildings and communities [Alexander et al., 1977; Alexander, 1979; Lea, 1994].

Patterns help to abstract the design process and to reduce the complexity of software because patterns specify abstractions at a higher level than single classes and objects. This higher-level is usually referred as the pattern level.

There are different kinds of patterns, of varying scale and level of abstraction, being usually classified in architectural patterns, design patterns, and idioms [Buschmann et al., 1996].

- **Architectural patterns** express fundamental structural organization schemes for software systems.

- **Design patterns** are medium-scale tactical patterns that reveal structural and behavioral details of a set of entities and their relationships. They do not influence overall system structure, but instead define micro-architectures of subsystems and components.

- **Idioms** (sometimes also called coding patterns) are low-level patterns that describe how to implement particular aspects of components or relationships using the features of a specific programming language.

Patterns represent useful mental building blocks for dealing with specific design problems of software system development.

**Framework level**

Object-oriented systems of medium size typically involve a large number of classes, some patterns, and few layers of cooperating frameworks. Frameworks are used to describe a system at an higher level than classes and patterns.

The concepts of frameworks and patterns are closely related, but neither subordinate to the other. Frameworks are usually composed of many design patterns, but are much more complex than a single design pattern. In relation to design patterns, a framework is sometimes defined as an implementation of a collection of design patterns.

A framework can also be seen as a representation of a specific domain under the form of a reusable design together with a set of implementations often
reusable and ready to instantiate.

A good framework has well-defined boundaries, along which they interact with clients, and an implementation that is usually hidden from the outside. Frameworks are a key part of medium to large-scale development, but even them have an upper limit to cope with high levels of complexity [Bäumer et al., 1997].

Component level

On the highest level of granularity, a system can be described as a set of large-scale components that work together to support a cohesive set of responsibilities. A component is defined in [Szyperski, 1998] as a “unit of composition with contractually specified interfaces and explicit context dependencies only; (...) (it) can be deployed independently and is subject to composition by third parties”. Examples of large-scale components are domain components, which are collections of related domain classes covering a well-defined application domain or a part of. Large components may or may not have been built from one or more object frameworks [Wegner et al., 1992], but in the case of an object-oriented system they typically are.

2.4 Definition of Concepts

An object-oriented framework is a reusable software architecture comprising both design and code. Although this statement is generally accepted by most authors, there are a number of different definitions for object-oriented frameworks that emphasize other aspects of the framework concept.

The most referenced definition is perhaps the one found in [Johnson and Foote, 1988], which says that: “a framework is a set of classes that embodies an abstract design for solutions to a family of related problems”. This definition captures the essential aspects of the object-oriented framework concept, namely: (1) a framework comprises a set of classes; (2) a framework embodies a reusable design; and (3) a framework addresses a family of problems in a domain.

Other definitions present other aspects of frameworks, which altogether help us get a better understanding of the concept. For example, Deutsch states that “a framework binds certain choices about state partitioning and control flow; the (re)user (of the framework) completes or extends the framework to produce an actual application” [Deutsch, 1989]. The first part of this definition emphasizes (4) the structural aspect of a framework, by stating that architectural design decisions have been taken. The second part explicitly describes the main purpose of a framework, which is (5) to be adapted to the problem at hands, namely by extending or completing some of its parts.
In [Gamma et al., 1995] a framework is defined as “a set of cooperating classes that make up a reusable design for a specific class of software”, which is based on the two definitions above mentioned.

The definition given in [Cotter and Potel, 1995] concisely presents almost all the aspects previously presented (all but the (4)): “A framework embodies a generic design, comprised of a set of cooperating classes, which can be adapted to a variety of specific problems within a given domain”.

In the following definition, in [Johnson, 1997], a framework is defined as “(...) the skeleton of an application that can be customized by an application developer”. This definition reinforces the structural aspect of a framework, and that future applications will conform to them by customizing parts of the framework. The activity of framework “adaptation” is referred in this definition as framework “customization”, but the essential meaning of both terms are similar. In the same reference, a framework is also defined as “(...) a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact”. This definition indicates that a framework doesn't necessarily need to cover a complete problem domain, but possibly only smaller parts of it, thereby suggesting the possibility of composing several frameworks together to build concrete applications. The wording “set of abstract classes” may suggest that the extension of a framework has to be done through inheritance, but this is not completely true as there are other ways of extending a framework, namely by composition.

Therefore, using a more complete and longer definition, we can define a framework as a software artifact:

- encompassing a set of cooperating classes, both abstract and concrete;
- expressed in a programming language, providing reuse of code and design;
- and specially designed to be customized, by inheritance or composition, for the construction of concrete solutions (systems, or applications) for a family of related problems within a specific problem domain.

Shortly, a framework emphasizes the more stable parts of an application domain, as well as their relationships and interactions, and provide customization mechanisms that let application developers solve their particular problems in the that domain.

2.4.1 Object-Orientation Concepts

Much of the reuse power of object-oriented frameworks comes from the
most distinguishing characteristics of object-oriented programming languages: data abstraction, inheritance, and polymorphism.

Data abstraction

Class definitions in an object-oriented language are primarily a data abstraction mechanism that enables the unification of data together with the procedures that manipulate them. Through abstraction and encapsulation, classes enable the separation of interfaces from implementations, and thus the change of implementation details without affecting its clients. As a result, classes can often serve as fine-grained reusable components.

Inheritance

In object-oriented languages, classes can be organized along hierarchies supporting different kinds of inheritance. Class inheritance allows the properties and behavior of a class to be inherited and reused by its subclasses. Inheritance in programming languages can be seen as a built-in code sharing mechanism that, without polymorphism and dynamic binding, won’t be much different from several module import mechanisms of traditional languages.

Polymorphism

This is a feature of object-oriented languages that enables a variable to hold objects belonging to different classes. When combined with overloading and dynamic binding, polymorphism becomes a powerful feature of object-oriented languages that enables to mix and match components, to change collaborators at runtime, and to build generic objects that can work with a wide range of components. Overloading makes it possible for several classes to offer and implement many operations with the same name, being up to the compiler or runtime environment to disambiguate references to a particular operation.

The combination of these features allows for a greater flexibility in programming. Due to polymorphism, a single variable in a program can have many different types at run-time. Inheritance provides a way of controlling the range of types a variable can have, by allowing only type mutations within an inheritance tree. Finally, dynamic binding enables delaying until run-time the determination of the specific operation implementation (method) to be called in response to an operation request, when the actual types of the variable and operation parameters are known [Meyer, 1988; Booch, 1994].

Two of the most distinguishing features of the framework concept rely heavily on the use of dynamic binding: the extensive use of template and hook methods, and the inversion of control flow.

2.4.2 Template and Hook Methods

Frameworks are designed and implemented to fully exploit the use of dynamically bound methods. To illustrate this, we will present a simple
example of a hypothetical single class framework for unit testing.

The framework consists of a single abstract class named `testCase`. This class has three operations named `setUp`, `runTest` and `tearDown`. In order to implement tests for database connection operations, or mathematical operations, for example, the framework is supposed to be extended with concrete subclasses, such as `DBConnectionTest` or `MoneyTest`.

As different tests usually have different ways of being setup, executed, and terminated, the framework, i.e. the `testCase` class, doesn't provide implementations for these operations, being up to framework users to provide them. Although the details of concrete test implementations may differ, the overall running of a test is always the same, consisting of: the setup of the test, the running of the concrete test, and its finalization.

To capture this commonality between different test implementations, the framework implements a generic operation to run tests. This generic operation is named `run`, and its implementation is responsible for handling the invocation of the `setUp`, `runTest` and `tearDown` operations. In other words, to run any test case, it is only needed to invoke the `run` operation of `testCase` being up to `run` to do the rest. An illustrative implementation of the `testCase` class using the Java programming language is shown in Figure 2.2.

Concrete subclasses of `testCase`, such as `MoneyTest`, should provide implementations for the `setUp`, `runTest` and `tearDown` operations. Due to the mechanism of dynamic binding, when the `run` operation is called on an instance of `MoneyTest`, it is the `run` operation of `testCase` that will be used (if not overridden in `MoneyTest`). The method `run` of `testCase` will then invoke the `setUp`, `runTest` and `tearDown` operations implemented in the `MoneyTest` class.

```
abstract public class TestCase {
    public void run(){
        setUp();
        try {
            runTest();
        } finally {
            tearDown();
        }
    }
    abstract protected void setUp(){ }
    abstract protected void runTest(){ }
    abstract protected void tearDown(){ }
}
```

Figure 2.2 The class `testCase`.  

A Minimalist Approach to Framework Documentation
The point here deserving attention is the fact of an operation in a superclass, the `run` operation of `TestCase`, being able to call operations in subclasses, and therefore (the superclass) having control over the execution flow of the overall test sequence. The `run` operation is often called a template method, and the `setUp`, `runTest` and `tearDown` operations are called hook methods.

Template and hook methods are two kinds of methods extensively used in the implementation of frameworks. These terms are commonly used by several authors in [Wirfs-Brock et al., 1990; Pree, 1991; Gamma et al., 1993; Pree, 1995].

*Template methods* are implemented based on hook methods, and call at least one other method. A *hook method* is an elementary method in the context which the particular hook is used, and can be either an abstract method, a regular method, or another template method. An *abstract method* is a method for which only the interface is provided, and thus lack an implementation. A *regular method* is a method that doesn’t call hook or template methods, but only provides a meaningful implementation.

Generally, template methods are used to implement the frozen spots of a framework, and hook methods are used to implement the hot spots. The *frozen spots* are aspects that are invariant along several applications in a domain, possibly representing abstract behavior, generic flow of control, or common object relationships. The *hot spots* of a framework are aspects of a domain that vary among applications and thus must be kept flexible and customizable.

The difficulty of good framework design resides exactly on the identification of the appropriate hot spots that provide the best level of flexibility required by framework users. More hot spots offers more flexibility, but results in a framework more difficult to design and use, so somewhere in between resides a balanced design.

In our simple testing framework example, the `run` template method implements the overall execution of a test case (a frozen spot), which consists on preceding the execution of the test case with a setup, followed by a tear down operation responsible to release any resources eventually used during the test. All these operations are supposed to be provided by the hook methods `setUp`, `runTest` and `tearDown`, which are abstract methods. The class `TestCase` is considered an abstract class because it has at least one abstract method (actually it has three).

Template and hook methods can be organized in several ways. Although they can be unified in a single class, as in our example, in most of the situations it is better to put frozen spots and hot spots into separate classes. When using separate classes, the class that contains the hook method(s) is
considered the hook class of the class containing the corresponding template method(s)—the template class. We can consider that hook classes parameterize the corresponding template class. The hook methods on which a template method is based can also be organized in different ways. They can be defined all in the same class, or in separate classes, in a superclass or subclass of the template class, or in any other class.

In [Pree, 1995] are identified several ways of composing template and hook classes, and presented under the form of a set of patterns, globally called meta-patterns. Meta-patterns categorize and describe the essential constructs of a framework, on a meta-level. Design patterns provide proven solutions to recurrent design problems and are extremely useful to design object-oriented frameworks.

In our framework example, template and hook methods are unified in a single class, because the object providing the template method is not separated from the objects providing the hook methods, actually being an instance of MoneyTest (an instance of MoneyTest is also an instance of TestCase). This organization of template and hook methods is classified as the Unification meta-pattern, which corresponds to the simplest way of organizing template and hook methods. In Figure 2.3, this meta-pattern is represented attached to the classes of our example.

![Unification meta-pattern](image)

**Figure 2.3** The Unification meta-pattern attached to the testing framework.

With this unification meta-pattern, the developer must provide a subclass to
adapt the behavior of running a test, and this can’t be done at run time. Organizations that separate template classes and hook classes are called *Connection meta-patterns*, which allow the modification of the behavior of a $T$ object by composition, that is, by plugging in specific $H$ objects. The more sophisticated way of separating template and hook classes, called *Recursive meta-patterns*, occurs when the template class is a descendant of the hook class, which enables the composition of whole graphs of objects. In Figure 2.4 we show the basic differences of unification, connection and recursive meta-patterns.

**Figure 2.4** Unification, connection, and recursive meta-patterns.

With the single class framework example we have illustrated the usage of inheritance and dynamic binding for operations in one single class. By scaling up the example to a larger framework, with more abstract classes, more template and hook methods organized according to more powerful meta-patterns, we can have a better idea of the potential reuse power that well-designed frameworks can deliver to their users.

### 2.4.3 The Flow of Control in Framework-based Applications

The development of applications reusing frameworks leads to an inversion of control between the application and the software on which it’s based. When we use a class library, we write the main body of the application and call the code we want to reuse. When we use a framework, we reuse the main body and write the code it calls [Gamma et al., 1995]. By consequence, the code to be written must satisfy particular names and calling conventions defined by the framework, what reduces the design decisions we need to do. This inversion of control is characteristic to frameworks and is referred as the *Hollywood Principle*, meaning “Don’t call us, we’ll call you” [Cotter and Potel, 1995; Bosch et al., 1999].
This inversion of control flow in programs is an idea that has evolved over years of application development, passing by different ways of structuring programs: from procedural programs, to event-loop programs, and then to framework programs (Figure 2.5).

**Procedural programs**
In *procedural programs*, all the code for control flow is provided by the programmer. The program is executed sequentially, always under the programmer's control, and when necessary calls procedures from libraries provided by the operating system. The system takes action only when it is called by the program.

**Event-loop programs**
When using graphical user interfaces, sequential control flow is no longer appropriate, as end users may select when and which actions to perform. A solution to this problem led to the concept of *event-loop programs*, which let the user choose the order in which events happen, through the interaction with input devices (mouse, keyboard, etc.). These programs have an event loop that is responsible to sense user events and call the corresponding parts of the program configured to handle them, remaining programmer’s responsibility the flow of control within these parts.

**Framework programs**
Framework-based applications turn over control to the user, as happens with event-loop programs, and then to the original framework developers. The framework code assumes almost all flow of control, calling application code only when necessary. Calls are however not made exclusively in one direction: application code often calls framework code too. As a result of this two-way flow of control, it is not needed to design and write the control code required by event-loop programs or other code common to many applications that can be written once and reused many times afterwards. Ideally, with frameworks we design and write only a small part of the total
The shifting of control flow is a question of degree, and not absolute. We can say that a program exists on a scale somewhere between 0% and 100% framework-owned control flow. When developing applications using frameworks the goal is to shift the control flow as much as possible to the framework.

Back to our example, we will now analyze the oscillation of the flow of control between the framework and the application. As described before, the framework of this very simple example consists of a single class (TestCase) which is only customizable by inheritance. The application (MoneyTest) customizes the framework by providing implementations for the abstract hook methods setUp, runTest and tearDown.

The flow starts in the main method of the application's code. A MoneyTest object is created, and its run method is called. Due to the mechanism of dynamic binding, the run method selected to be executed is the one implemented in TestCase, the superclass of MoneyTest, and thereby the control flow is transferred to the framework. The run method then starts and calls the setUp method, declared as abstract in TestCase and implemented in MoneyTest. Now, the dynamic binding mechanism selects to be executed the setUp method implemented in MoneyTest, and thereby the control flow is returned back to the application. When the setUp method terminates, the control flow turns back to the framework, and then to the application again in order to execute the runTest method implemented in MoneyTest, and so on until the end of the main method.

The Figure 2.6 graphically describes how the control flow have oscillated back-and-forth from the application to the framework, until the moment of calling the runTest method.

The mechanism used by this framework to call application-specific code relies on deriving application-specific classes (MoneyTest) from the base classes provided by the framework (TestCase), and on overriding their methods (setUp, runTest, tearDown).

While this customization mechanism focus on inheritance, there are other mechanisms that rely on composition. Both kinds of mechanisms have advantages and drawbacks. The most significant difference is on how they trade-off flexibility of customization with run time adaptability. Inheritance based mechanisms offer a good extension flexibility but don't support adaptation at run time. Composition based mechanisms requires explicit definition of points of flexibility but support adaptation at run time.

Inheritance and composition based mechanisms lead to two broad categories of frameworks: black-box and white-box frameworks.
2.4.4 Classifying Frameworks

Frameworks are typically classified according to the extension techniques provided and their scope of work.

White-box and black-box frameworks

Based on the extension techniques provided, frameworks can be classified in a range along a continuum from white-box frameworks to black-box frameworks [Johnson and Foote, 1988], as illustrated in Figure 2.7.

Figure 2.6 Illustration of control flow in the framework example.

Figure 2.7 Classification of frameworks based on the extension technique

White-box frameworks rely heavily on inheritance and dynamic binding in order to achieve extensibility. Although white-box reuse is the hardest way to
use a framework, it is by far the most powerful.

Black-box frameworks are the easiest to use, because they are structured using object composition and delegation rather than inheritance. On the other hand, black-box frameworks are the most difficult to develop, because they require the definition of the right interfaces and hooks able to anticipate a wide range of application requirements.

Most real-world frameworks combine black-box and white-box characteristics, being thus called gray-box frameworks. They allow extensibility both by using inheritance and dynamic binding, as well as by defining interfaces. Gray-box frameworks are designed to avoid the disadvantages of black-box frameworks and white-box frameworks.

In addition to the classification above, frameworks can also be classified according to their scope of work. In [Fayad and Schmidt, 1997b] is proposed a classification for frameworks based on their scope which consists of three categories: system infrastructure frameworks, middleware integration frameworks, and enterprise application frameworks [Fayad and Schmidt, 1997b].

System infrastructure frameworks aim to simplify the development and support of system infrastructure areas such as operating systems, user interfaces, communications, and language processing. Graphical user interface (GUI) frameworks, Java Foundation Classes (JFC), Microsoft Foundation Classes (MFC), or MacApp, are examples of frameworks used as underlying frameworks for other applications.

Middleware integration frameworks are usually used to integrate distributed applications and components. Examples of middleware integration frameworks include ORB frameworks, message-oriented middleware, and transactional databases.

Enterprise application frameworks address large application domains, such as telecommunications, banking, or manufacturing, and can provide a substantial return on investment as they support directly the development of end-user applications. A famous example of an enterprise framework is the IBM SanFrancisco Project.

These kinds of frameworks are related, as they layer up on top of each other. Middleware integration frameworks usually includes a system infrastructure in its underlying layer. Similarly, an enterprise framework includes both a middleware integration framework and a system infrastructure in the underlying layers.
2.5 History of Frameworks

Although the framework concept reached popularity recently (1990s), the history of frameworks dates back to the 1960s. The first examples of the framework concept found in the literature were designed to solve mathematical problems in Simula (1960s) and Smalltalk (1970s).

2.5.1 Early frameworks

The Simula programming language, created more than 30 years ago (1967), has precipitated the invention of the concepts of object-oriented programming. Simula is particularly important for framework technology because it was specifically designed to support frameworks, or application-oriented extensions, as they were then called. Simula was designed as a minimal addition to Algol, extending it with the basic concepts of object-oriented programming: objects, classes, inheritance, virtual methods, references, and a type system. The object concepts first introduced by Simula have percolated into most current object-oriented languages, such as C++ or Java.

It is generally accepted that the most significant distinction between a framework and a mere class library of classes depends on the presence of inverted control. In other words, the possibility that code in the framework may call code in the user part. In primitive languages this is implemented with callbacks, that is, procedure parameters. In most object-oriented languages, inversion of control is achieved through virtual procedures, which are declared and invoked by the framework code, but whose implementations can be redefined by the user code. Simula has virtual procedures, but also has an inner mechanism, which has the same characteristics of a framework calling user code. Beta [Madsen et al., 1993] is the only other language also having this mechanism [Hedin and Knudsen, 1999].

Simula provides a standard library containing two frameworks, Simset for list handling and the Simulation for discrete-event simulation. Each framework consists of a single packaging class that contains all the component classes, procedures, and variables. An application program obtains these capabilities by using the framework name as a prefix to the program. Simset framework implements two-way circular lists. Simulation is a framework that allows the language to handle the discrete event Simulation [Birritch, 1979]. By means of these two object-oriented frameworks, Simula provides superb facilities for simulation, namely pseudo-parallelism, real time abilities, and simulation of complex systems. In addition, with Simula it is particularly easy to combine quite diverse frameworks.
2.5.2 GUI frameworks

In the late 1970s, the emerging interactive paradigm of Graphical User Interfaces (GUI) based systems made windows and events to stand up as a new challenging domain for programmers, for which they need help to write software.

The difficulty of coding GUI applications directly on top of the complex procedural application programming interfaces (APIs) provided by the most popular GUI systems (Macintosh, X Window System, and Microsoft Windows) started a growing demand for finding better ways of developing software solutions.

The Smalltalk-80 user interface framework, named Model-View-Controller (MVC) and developed in the late 1970s, was perhaps the first widely used framework [Goldberg, 1984; Krasner and Pope, 1988]. MVC showed at that time (and continues showing today) that object-oriented programming is well suited for implementing GUIs. MVC divides an user interface into three kinds of components working in trios: a view and a controller interacting with a model.

One of the first user interface frameworks influenced by MVC was MacApp, which was developed by Apple Inc. to support the implementation of Macintosh applications [Schmucker, 1986]. MacApp was followed by user interface frameworks from universities, such as Interviews from Stanford University [Linton et al., 1989], and ET++ from the University of Zurich [Weinand et al., 1989].

MacApp, InterViews, and ET++ became very popular during the 1980s. These frameworks provided useful, generic abstractions for drawing views and windows, and offered an event-handling mechanism based on the MVC concept. Most importantly, with any of these frameworks, the writing of an application became much easier, and resulted in a more stable code base, than directly using the APIs provided by the respective GUI systems.

But frameworks are not limited to user interfaces, being applicable to basically any area of software design. They have been applied to the domains of operating systems [Russo, 1990], very large scale integration (VLSI) routing algorithms [Gossain, 1990], hypermedia systems [Meyrowitz, 1986], structured drawing editors [Vlissides and Linton, 1990; Beck and Johnson, 1994], network protocol software [Hueni et al., 1995], and manufacturing control [Schmidt, 1995], to mention a few.

2.5.3 Taligent frameworks

In 1992, Apple and IBM have founded Taligent as a joint venture, which was
joined by Hewlett-Packard in 1994. Taligent goal was to develop a fully object-oriented operating system and portable application environment, which shipped in July 1995 as the CommonPoint Application System. CommonPoint was a set of tools for rapid application development consisting of more than a hundred small object-oriented frameworks [Andert, 1994; Cotter and Potel, 1995] running on top of OS/2, Windows NT, AIX, HP-UX, and a new Apple OS kernel.

CommonPoint was most similar in scope and portability to Sun’s subsequent Java environment, but based on C++ and without a virtual machine and a new object programming language (Java). The CommonPoint development environment was a visual component-based incremental development environment akin to the now-familiar IBM’s VisualAge or Borland’s JBuilder IDE’s. The CommonPoint user interface paradigm known as “People, Places, and Things”, extended the personal computer desktop metaphor to collaborative, distributed, task-centered workspaces that anticipated today’s web-based environments. In terms of framework technology, Taligent’s approach for CommonPoint made a shift in focus away from large monolithic frameworks to many fine-grained integrated frameworks.

In 1996, IBM took over sole ownership of Taligent, and in 1998 formally merged Taligent into IBM. During these two years, Taligent was an important center for object technology, providing key software components to IBM development tools, and licensed other key Java and C++ technologies to other industry partners, such as Sun, Netscape, Oracle and others. After 1998, Taligent engineering teams continued their development of object technologies and products at IBM.

2.5.4 Frameworks today (2000s)

The influence of the new GUI frameworks and the Taligent’s innovative technological approach have attained a lot of interest to the framework concept, and both widely promoted frameworks in larger communities.

At present, in the 2000s, frameworks are important and are becoming even more important as software systems increase in size and complexity. Component systems such as OLE, OpenDoc, and Beans, are frameworks that solve standard problems of building compound documents and other composite objects. Frameworks like Microsoft Foundation Classes (MFCs), many parts of Sun’s Java Development Kits (AWT, Swing, RMI, JavaBeans, etc.), implementations of the Object Management Group’s (OMG) Common Object Request Broker Architecture (CORBA), IBM’s WebSphere, SanFrancisco, Apache’s frameworks (Struts, Turbine, Avalon, etc.), Eclipse framework for integrated development environments, and JUnit testing framework, are all very important in contemporary software development.
Since its appearance in 1995, Sun’s Java has been one of the most successful, innovative and evolving language and frameworks (JavaBeans, Java Foundation Classes, Enterprise JavaBeans components, JavaOS, etc.). Java supports many platforms, from very small ones as smartcards, thin and thick clients to large mainframe installations. Java 2 Enterprise Edition (J2EE) has become one of the most successful frameworks for the web and enterprise technology.

In 2001, a new framework called .NET has emerged from Microsoft. It has many similar features to J2EE and will probably be one of the closest competitors of J2EE. Although both frameworks stand on a same foundation of programming languages, object models and virtual machines, they are different when considering the design goals of their runtime environment, namely the portability of code to different platforms: .NET uses a common intermediate language, and J2EE uses bytecode for a virtual machine. These two dominating frameworks promise to compete very closely in the next few years to come.