CODEN:LUTEDX(TETS-5231)/1-146/(1995)&local 31

Development of Object-Oriented Frameworks

Authors

Niklas Landin
Axel Niklasson

Tutors

Grace Bosson
Ericsson Software Technology, Frameworks
Ronneby, Sweden

Björn Regnell
Department of Communication Systems
Lund Institute of Technology, Lund University
Lund, Sweden

ABSTRACT

An object-oriented framework is a reusable software component providing large scale reuse, including reuse of analysis and design.

The thesis describes a process for the development of object-oriented frameworks. A comprehensive introduction to object-oriented frameworks is given and an extensive set of guidelines supporting framework development is provided.

The authors have studied several object-oriented development methods to investigate how an ordinary development process may be adapted to suit the development of object-oriented frameworks. A small case study has been performed to validate and exemplify the process.

The main focus is on capturing requirements, analysis and design of object-oriented frameworks. The process of reusing frameworks for application development is also described briefly.

CONTENTS

1 INTRODUCTION

1.1 Background and Problems

During the past years, the need for software reuse has become evident. Object-orientation has provided a means to increase the reusability of code, by introducing standard interfaces and inheritance. Class libraries have provided well defined and tested reusable components, but using class libraries mainly implies reuse of code and little reuse of analysis and design. To increase the potential of reuse, *object-oriented frameworks* have been suggested. An object-oriented framework is intended to capture the functionality common to several similar applications. By developing and reusing frameworks, the reuse will also encompass design and analysis.

Since developing object-oriented frameworks implies an analysis and design of all the applications in the domain, existing and future, the feasibility is highly dependent on the availability of experienced software designers and people with extensive knowledge in the domain. To lessen the dependence on key persons, there is a need for a methodology and a development process to follow.

Ericsson Software Technology Frameworks in Ronneby have a template methodology, which they apply to their customers' development processes and organisations. The methodology was sporadically documented, and much of the methodology was concealed in the heads of the company's consultants.

The problem is to visualize the methodology to the customers, and to new consultants. To alleviate this problem, the documentation needed to be compiled, revised and further developed.

Methodology

To compile an adequate documentation, we assembled information on frameworks and development processes and methodologies. Many articles and references were found on the WWW, some documentation was available at the company, and the rest have been elicited from informal interviews with the consultants of the company, both methodologists and developers.

1.2 Objectives

- To give an introduction to the object-oriented framework concepts.
- To provide guidelines that support the development of object-oriented frameworks.
- To overview the framework reuse process.

1.3 Target Groups

The target group for the guidelines given is a software organization intending to develop object-oriented frameworks in their application domain(s).

Target groups for the more general parts of the thesis are:

- students interested in object-oriented techniques, who wish to find out about object-oriented frameworks as a reuse technique.
- software organizations in need of a reuse technique.

1.4 Delimitations

We do not judge whether frameworks are appropriate or not in a financial perspective, and we do not provide a complete development process. We specify how to customize the target organization's development process for framework development, with the focus on analysis and design.

1.5 Disposition of the Thesis

This introduction is followed by an introduction to the framework concept. The following chapters describe the development processes, framework development and framework reuse. The thesis is closed with a summary and our conclusions. The appendices include a list of the guidelines provided, a small case study, a brief glossary and a list of our references.

Each chapter starts with an introduction and a short description of the chapter's disposition. The chapters are closed with a short summary.

The graphical notation used in illustrations of object-oriented designs is described in Appendix B.

1.6 Acknowledgements

Our tutors Grace Bosson and Björn Regnell have provided invaluable help. We would also like to thank the Framework and Cafka employees of Ericsson Software Technology, especially the boys on the 4:th floor. Particular thanks go to Michael Mattsson, whose extensive framework bibliography saved us a lot of work.

2 Object-Oriented Frameworks

The reuse of software components is recognized as an important way to increase productivity in software development. Experienced programmers have always reused code by using their experience and by looking into old code designs, but it has been found that the reuse of analysis and design has a significantly higher potential [Joh91].

The concept of frameworks makes it possible to reuse not just code, but also analysis and design.

This chapter is intended to promote the object-oriented framework concept and to give the reader a better understanding of it. The reader should have at least a basic knowledge of software engineering and object-orientation.

In section 2.1 we give a brief description of the features of object-orientation that make framework development feasible. Software reuse is introduced in section 2.2 and why software reuse should be achieved using frameworks is motivated in sections 2.3 and 2.4. Section 2.5 is a brief discussion on the documentation of frameworks and section 2.6 is a summary of this chapter.

2.1 Some Object-Oriented Concepts

It is the properties of object-oriented languages that make frameworks feasible, mainly in that they support the concepts of inheritance, polymorphism and dynamic binding.

2.1.1 Inheritance

When one class inherits from another the heir is called a subclass and the ancestor is called a superclass. The subclass inherits the superclass' methods and internal structures. In the subclass new methods may be added to the inherited ones and inherited methods may be redefined.

Inheritance has several advantages. Code is reused, as inheritance allows several classes to share common code. Inheritance is also suitable in the maintenance phase because it makes it possible to leave the original code in the superclass untouched and implement the changes in a subclass. Inheritance is a good way to structure the design and code. Good design practice states that a subclass should be a specialization of its superclass.

Inheritance has as mentioned a lot of advantages but it has some drawbacks as well. For instance, inheritance violates the information hiding introduced by objects - the subclasses are dependent on their superclasses and the code is spread throughout the system. This may increase the difficulties of understanding a large class hierarchy.[Joh88; Tal94a]

2.1.2 Dynamic Binding and Polymorphism.

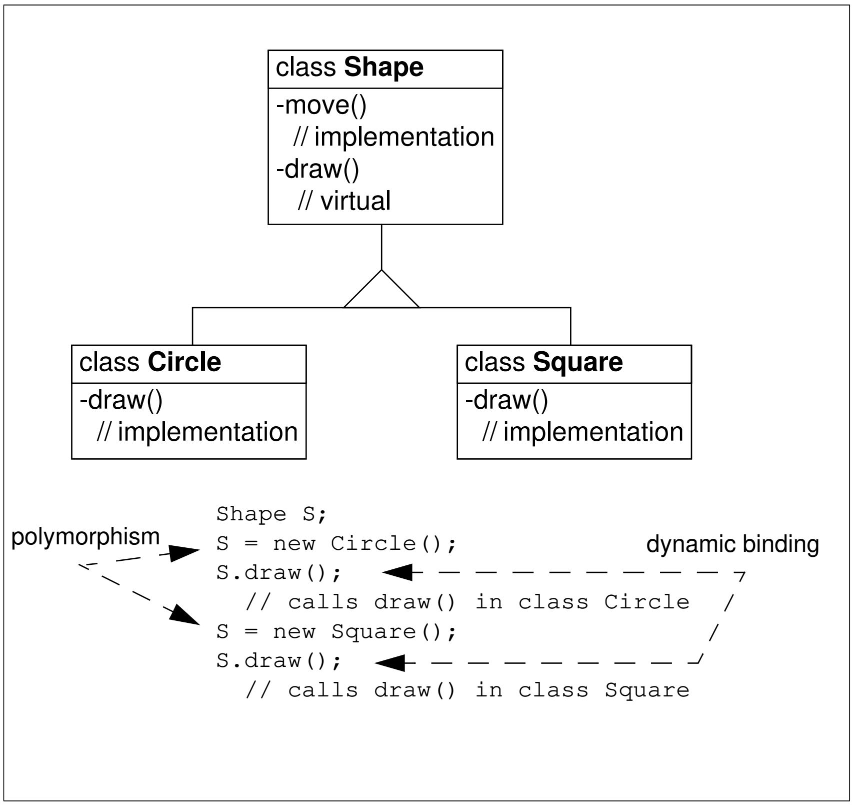


Figure 2.1 The concepts of dynamic binding and polymorphism.

Dynamic binding implies late binding of function calls, i.e. the function call is bound to an object during runtime and not during compilation [Joh91, p. 3; Ohl93; Mey94]. Checks on object types like:

```
if (S is of type Circle) {drawCircle();}
elseif (S is of type Square) {drawSquare();}
```

are replaced by a call to the common function `S.draw()`, see Figure 2.1.

In such a call the runtime system makes the decision whether to call the function `draw()` implemented in `class Circle` or the function `draw()` implemented in `class Square`.

A result of dynamic binding is that *polymorphism* is achieved. Polymorphism implies that variables and parameters can take on values of different types, see Figure 2.1 [Joh91, p. 3; Ohl93; Mey94].

2.2 Software Reuse

The gap between software demand and software productivity is steadily increasing. One way to increase the productivity is by introducing software reuse.

The idea behind reuse is to not develop anything that already exists, but to reuse it. This will lead to shortened development time and a decrease in time-to-market.

Designing software for reuse aims to produce general, extensible software components. Designers are asked to predict possible future applications and incorporate their requirements into the current design. To accomplish this, the huge number of design decisions the designers have to take will have to be limited. This has traditionally been done by providing domain-specific procedural libraries of functions or libraries of reusable classes.

The problem with this approach is that it is hard to provide default behaviour and embody domain expertise in a library. A component library contains rather small components of which the reuser has to build the required application, and when building an application with small components the communication between the different components still has do be defined.

These constraints are alleviated by using a framework; the application designer does not have to know how or when to call each function - the framework does it for him. The communication between components inside a framework is already defined and the reuser needs not to be concerned about it. [Tal94a; Laj92]

Class Libraries

A class library is a set of related classes designed to provide reusable, general-purpose functionality. An example of a class library is a set of collection classes for lists and stacks. Class libraries do not impose a particular design on an application; they just provide functionality that can help the application do its job. Class libraries provide functionality at a low level and the developer must provide the interconnections between the libraries.

As a framework consists of a collection of classes it could be regarded as a class library. This is not entirely true, since in a class library every class by itself is unique and most of the classes in a framework are dependent of each other and will be of no use outside the framework.

The great difficulty with reuse libraries is that they must contain domain-specific architectures and components large enough to be worth reusing [Grif95]. To quote Mili, Mili and Mili: “For instance, objects seldom offer any interesting behaviour on their own and it is often in combination (interaction) with other objects that any useful functionality is achieved.” [Mil95].

Frameworks provide reuse at a larger level. The technique of reusing frameworks leads to reuse of analysis and design, but they also enable a higher level of code reuse than possible with toolkits [EST95a]. In this way applications can be developed by using the framework as a starting point and writing smaller amounts of code to modify or extend the framework’s behaviour[Tal94b].

2.3 What is a Framework?

Object-oriented frameworks have been defined in two similar ways by Johnsson [Joh88; Joh91]: “A framework is a set of classes that embodies an abstract design for solutions to a family of related problems.” and “A framework is a set of objects that collaborate to carry out a set of responsibilities for an application subsystem domain.”.

An object-oriented framework is a set of cooperating classes, both abstract and concrete, that make up a reusable design for a specific class of software [EST95a, Gam94]. The framework determines the architecture of the applications built using it by partitioning the design into abstract classes and defining their responsibilities and collaborations and the thread of control. The design decisions that are common to its application domain are captured, so that the application designer can concentrate on the specifics of his/her application. Framework development thus emphasizes design reuse over code reuse [Gam94].

“Frameworks also reuse implementation, but that is less important than reuse of the internal interfaces of a system and the way that its functions are divided among its components.”

R E Johnson [Joh88]

Dynamic binding lets the framework treat an object without regard to its implementation. An application developer who derives a new class to customize the framework writes code following the template provided by the abstract superclass. The framework will invoke the methods, and because of this flip-flop in control, frameworks are sometimes referred to as upside-down libraries, see Figure 2.2 [EST95a].

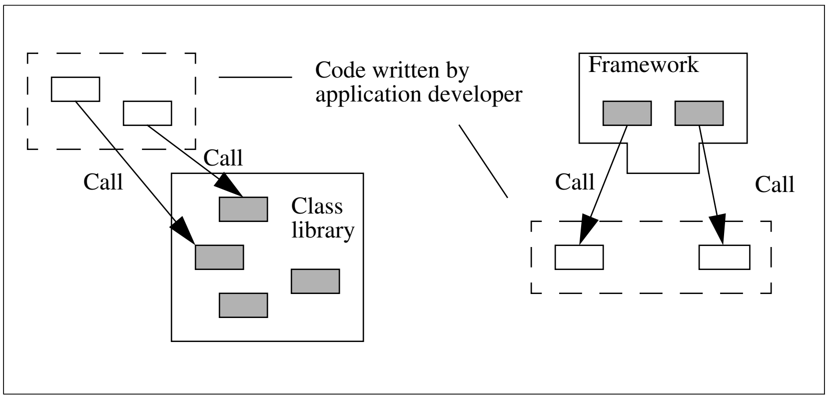


Figure 2.2 The difference in control between frameworks and class libraries.

The common framework provides an architecture-driven[1] base with a data-driven[2] layer. Application developers use the framework's built-in functionality by instantiating classes and calling their member functions (data-driven) and extend and modify the functionality by deriving new classes and overriding member functions (architecture-driven) [Tal94a].

When some applications have been developed using the framework there will be libraries of subclasses to choose from and the customization can be made increasingly by composition [EST95a; Tal94a]. If the applications are developed entirely by composition, there is no need for implementation or testing and the lead time is at a minimum.

As well as classifying frameworks by their internal structure, Taligent has classified frameworks by the problem domain they address [Tal94a]:

- **Support frameworks** provide system-level services, such as file access, distributed computing support, or device drivers. Application developers typically use support frameworks directly or use modifications produced by system providers. However, even support frameworks can be customized - for example when developing a new file system or device driver.

- **Application frameworks** encapsulate expertise applicable to a wide variety of programs. Current commercial graphical user interface (GUI) application frameworks, which support the standard function-

(1) Also referred to as white-box[Joh91] or inheritance-focused[Tal94a].
(2) Also referred to as black-box[Joh91] or composition-focused[Tal94a].

ality required by all GUI applications, are one type of application framework.

- **Domain frameworks** encapsulate expertise in a particular problem domain, e.g. a securities trading framework or a multimedia framework.

2.4 Why use Frameworks?

When using a well designed, well documented framework, both *analysis, design* and *code* are reused. A framework makes it possible to reuse analysis by describing the objects of importance, the relationships between the objects and how large problems are broken down into smaller ones. The design is reused in that the framework design contains abstract algorithms and defines the interfaces, as well as the constraints an implementation must satisfy. The code is reused since a concrete class implemented by the user can inherit most of its implementation from its superclass [Joh91].

The benefits from frameworks and reuse are gained over time, since the productivity gains do not come just from the first time you are using the framework, but from multiple use of the technology.

2.4.1 Advantages

- **Reduced time to market** When writing applications with a framework as a foundation only the code that differs from earlier applications has to be written. Less code writing is needed, which means shortened development time and hence reduced time to market.

- **Maintenance** Maintaining systems is very expensive. As much as 60 - 85% of the total life-cycle cost of a large system is spent on maintenance, and there are great potential savings in reducing the need for maintenance [Mey88]**.** When maintaining several applications of one framework only the framework and the code that is different between the applications have to be maintained. This means that changes only has to be implemented in one place, ensuring consistency. Compared to maintaining several different systems, the potential savings with framework designs are significant.

- **Testing** When reusing a framework, the tests are also reused. The only tests that have to be provided are the tests of the new modules and the interaction between the new modules and the framework, the system tests. Thus the amount of testing and debugging to be done is reduced. This assumes that the framework is correct and that the system tests checks that the framework is used correctly.

- **Reliability** A framework may, as well as all other software, contain errors and bugs, but as the framework is reused it tends to get stabilized and new errors and bugs will be reported more seldom. Reusing a stable framework will increase the reliability compared to writing a complete new code.

- **Standards** A well designed framework that follows company standards captures best practice. When developing applications from a framework, the framework sets constrains on the application code written. This leads to conformation to company standards as well as to best practice.

- **Frameworks embody expertise** Good software design in a particular area requires domain knowledge that is typically acquired only by experience. Because frameworks embody expertise, problems are solved once and the business rules and design are used consistently. This allows an organization to build from a base that has been proven to work in the past. Frameworks also enable software developers to concentrate on application solutions and rely on the framework to provide consistent services. This frees developers who are not necessarily experts in a certain area from the complexity of the underlying details. This is possible because the framework is in control. The framework provides the flow of control, while the application programmer's code waits for the framework to call. This means that the developers do not have to be concerned with details and they may focus solely on the problem domain [Tal94b].

- **Improved consistency and compatibility** There is a greater ability to work together for those applications that share a framework. They are also better integrated from a user point of view, having the same or similar user interface [Tal94b].

2.4.2 Difficulties

Components and architectures do not become reusable by it self. They must be designed with reuse in mind or redesigned for reuse. Designing for reuse takes longer time than designing systems or components without any thoughts of reuse. This extra time must be seen as an investment [Joh88; Tal94a].

It is also more difficult to design a framework than to design a component library, but the potential profit from reusing a framework is much greater than the profit from reusing a component from a component library. A framework is more difficult to design because the framework's architecture has to be designed as well as the communication between the internal components of the framework [Joh88; Tal94a]. When designing a component for a component library no such decisions has to be made.

For framework development to be successful, it must be supported by your team's processes and organization. It must be realized that the benefits from frameworks and reuse are gained over time, with multiple use of the framework[Joh88; Tal94a].

2.5 Framework Documentation

"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 them from scratch."

Grady Booch [Boo94]

The documentation of a framework is essential to its reuse potential. The documentation must describe:

- the purpose of the framework,
- how to use the framework,
- the detailed design of the framework.

Process and product documentation are necessary in all software projects, but especially important in the development of frameworks. The detailed design has to be available to the application developers, as well as a description of how to use the framework. As some aspects of a framework design are not well expressed as code, e g invariants maintained by the cooperating objects, there is a need for some other means of documentation [Joh91].

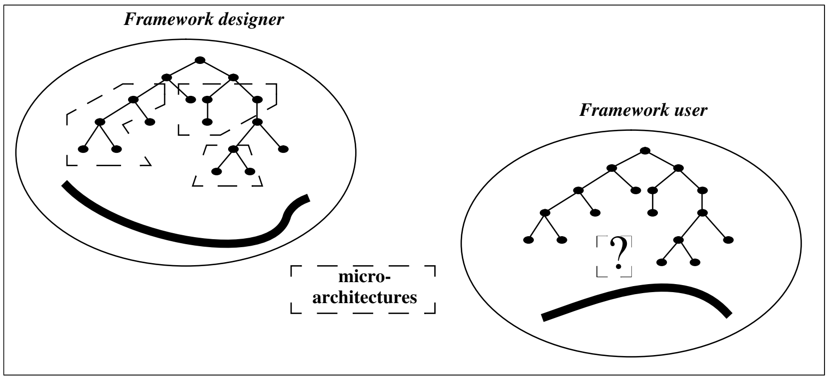


Figure 2.3 The informed framework designer has a clear picture of the framework and its micro-architectures, whereas the novice framework user is overwhelmed with the many, seemingly unrelated, classes in a poorly documented framework [Laj92].

Design Patterns

Many common framework design problems have been solved many times by different designers. Gamma et al. have documented some of their design experience in form of *design patterns* [Gam94]. Design patterns are generic designs to problems that often occur during object-oriented design. In framework design a problem might be how to keep the instantiation of the application specific classes outside the framework. The design pattern “Abstract Factory” provides a generic solution to such a problem, a solution that has been applied in several frameworks, and thereby is well-proven, see Figure 2.4 [Gam94, p. 87].

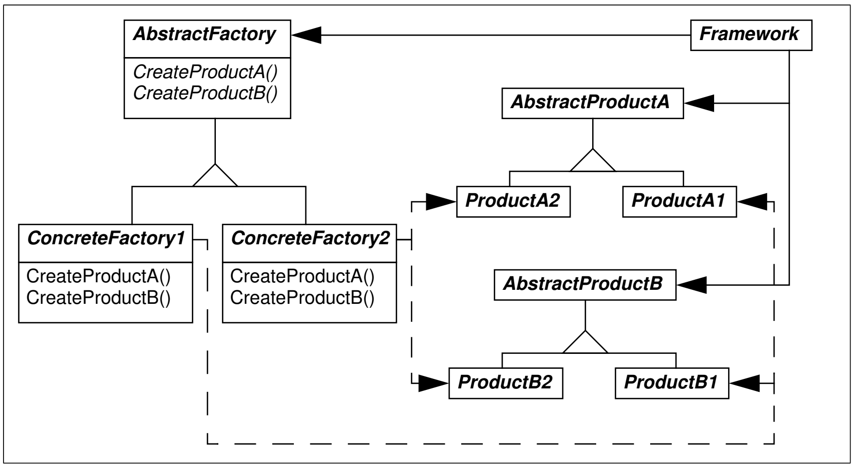


Figure 2.4 The “Abstract Factory” design pattern [Gam94]. For details on how the “Abstract Factory” can be applied, see Appendix B.

The design pattern concept originates from the architectural pattern concept introduced by Christopher Alexander in 1977 [Alex77]. Alexander’s patterns concern buildings and towns, but Gamma et al. adapts the concept to software design [Gam94].

Design patterns aim to capture design experience in a form that can be used effectively and to make it easier to reuse successful designs and architectures [Gam94].

A design pattern essentially consists of four elements:

- The **pattern name** increases design vocabulary and makes possible design at a higher level of abstraction.

- The **problem** describes when to apply the pattern.

- The **solution** describes the elements that make up the design, their relationships, responsibilities and collaborations.

- The **consequences** are the results and trade-offs of applying the pattern.

Reusing common patterns opens up an additional level of design reuse, where the implementations vary, but the micro-architectures represented by the patterns still apply [Gam94].

The term "design pattern" often refers to a pattern described by Gamma, [Gam94] or some other pattern catalogue, but any generic design can be expressed as a design pattern. The term is not absolute, something that is a design pattern to one designer might be a basic building block to another [Gam94]. What is considered a design patterns also depend on the target implementation language, e.g. if the implementation is conducted in a procedural language, design patterns might be "Inheritance" or "Polymorphism" [Gam94].

Framework design is hard, and all potential problems should be foreseen. Communication between designers and design teams, and the understanding of design decisions, and thereby the motivation of the decisions, are evident trouble sources, and a way to alleviate these problems are design patterns. Design solutions that follow proven design patterns are motivated, and communicating designs in the form of patterns add a level of abstraction, and therefore alleviate communication.

Using design patterns

- provides a common vocabulary for design,

- reduces the system complexity, since abstractions are named and defined consequently, and reduces the frameworks learning time,

- provides building blocks, from which more complex designs can be built, for example a framework,

- provides targets for restructuring class hierarchies [Laj94; Gam94].

Design patterns are a natural part of a framework documentation, since they motivate the design decisions [Gam94; Joh92; Laj94].

Without using design patterns or some other means of describing the common micro-architectures that emerge in the development of frameworks the situation illustrated in Figure 2.3 might occur [Laj94].

2.6 Summary

An object-oriented framework is the implementation of the general parts of several applications. A framework defines a set of related classes and the collaborations needed to provide the general functionality of the applications in the domain the framework covers. The framework defines how the classes will interact by defining the protocols and the algorithms.

Developing a framework will make it possible to reuse not only code, but also analysis and design. It is more time consuming to develop a framework than to develop an application with an ordinary architecture. The gains from framework development occur when the framework is reused and new applications are developed with short lead times, and a limited need for testing and maintenance.

The documentation of a framework is essential for its reuse potential. The documentation must describe:

- the purpose of the framework,
- how to use the framework,
- the detailed design of the framework.

Design patterns are suggested as a natural way to document frameworks, as they should be used in framework design.

3 FRAMEWORK DEVELOPMENT

This chapter covers the development of object-oriented frameworks. The development process is outlined in Figure 3.1.

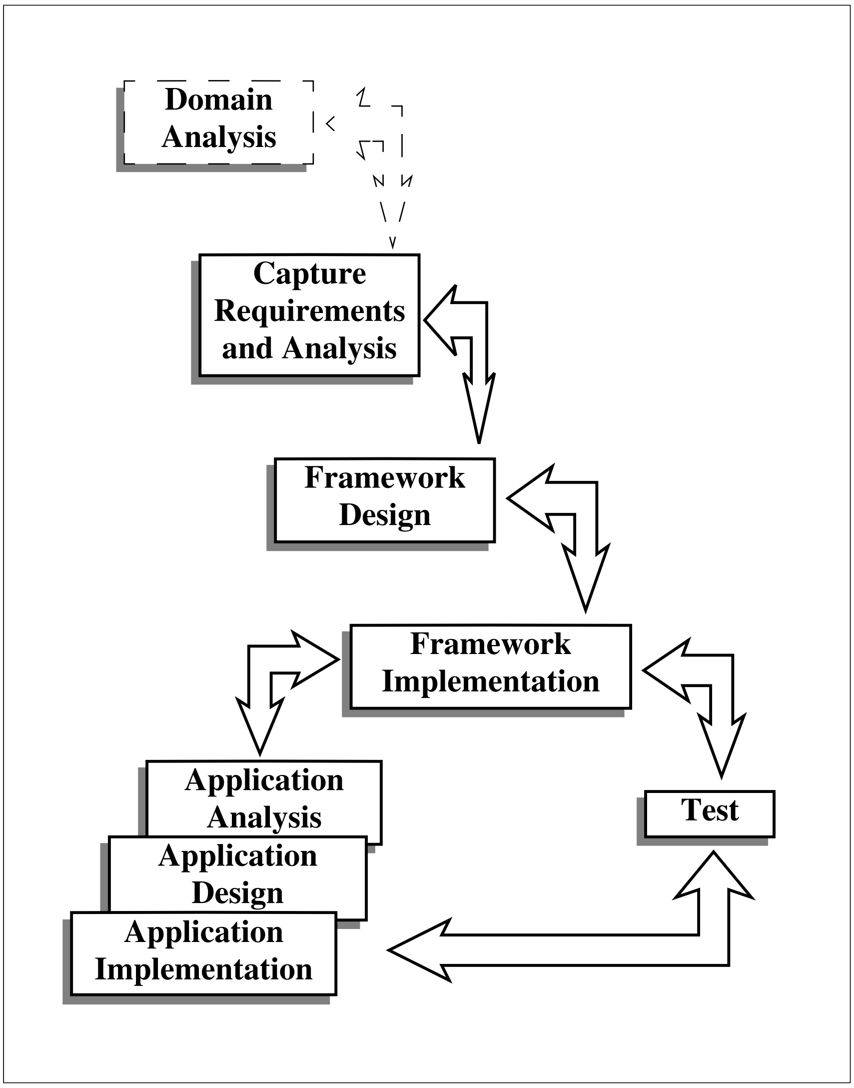


Figure 3.1 Process roadmap

3.1 Introduction

During framework development the developers should try to migrate as much common behaviour as possible from the applications into the framework. The process presented in this chapter is intended to support the construction of an adequate framework. The project organization during framework development is covered in section 3.2 and a presentation of the actions that should be taken before initiating a framework development process are described in section 3.3. The focus of this chapter is on the activities in analysis and design, as described in sections 3.4 and 3.5. These two sections provide guidelines for how to accomplish a good framework design. Guidelines for the transformation of the design into a firm framework implementation are discussed in section 3.6, and test and maintenance are briefly reviewed in sections 3.7 and 3.8.

3.2 Project Organization

This chapter is intended to give a brief introduction to the organizational matters when developing frameworks. In section 3.3.1 it is argued how a framework development product should be treated according to time constraints. Section 3.3.2 points out the difficulties of dividing the work between several parallel working groups. In section 3.3.3 some of the difficulties in adapting the organisation to development for and with reuse are described. The chapter ends with a summary in section 3.3.4.

3.2.1 A Strategical Investment

The responsibility for the development of a framework should not be in the ordinary project organizations. The reusability of a framework is strongly dependent on well defined interfaces and a good architecture, since later changes of the architecture or the interfaces will affect all applications dependent of the framework. The development of a framework should not be on the critical path of a project because the team responsible for the development should not have to make any compromises on the framework leading to bad architectures and under-defined interfaces.

The development of a framework should be viewed as a strategical investment more than an operational investment. A well developed framework will be an asset to the company, which when reused decreases development effort and lead time of future projects. It is therefore suitable to create a department or a team responsible for strategical development.

3.2.2 Work Organization

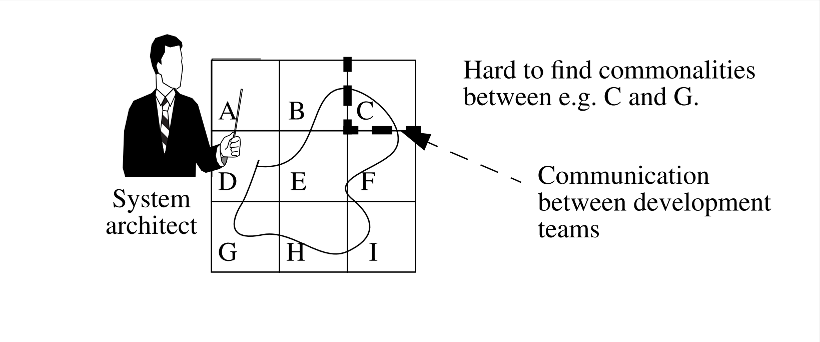


Figure 3.2 Traditional software development.

In traditional software development the work is as much as possible done in parallel with small development teams and well defined interfaces, see

Figure 3.2. Each team works with a well defined part of the system where the subsystems interfaces to the other parts of the system were defined in an earlier phase. A system architect maintains the overall picture of the system.

A framework development team should not be larger than an ordinary software development team. When the size of a software development team increases the communication overhead increases and more effort is needed to keep the team members informed. It becomes harder to get the overall picture of the team's progress.

A team suitable for framework development should roughly consist of no more than eight persons. It is suitable to vary the members according to the current phase of the development process. It is, for example, important that the team performing the domain analysis should include a couple of domain experts. When the development process continues the need for domain experts decreases and the need for system experts increases and it should be reflected by the composition of the development team.

Developing frameworks introduces new aspects when dividing the work. The main idea of a framework is to capture generalities of a domain or a set of applications within a domain. Finding generalities requires a good overview of the domain and the system respectively. This makes it less suitable to divide the work in several teams in an early phase.

There will be a trade off between shorter lead time, when dividing the work early, and a framework with a good and stable architecture, when not working in parallel.

The work should be divided into several parallel working teams as late in the development cycle as possible. The structure of the framework should as well as the public interfaces of the classes become stable before dividing the work, but it is not necessary to have defined the objects and classes in detail.

3.2.3 Development Teams For and With Reuse

There are basically two possible ways to organize the staff when applying reuse of frameworks. One is to let the same people both develop the framework and reuse it. The other way is to have separate development and reuse organizations.

If the intention is to sell the framework outside the organization the choice of reuse organization is limited. However we think that most associations will use their frameworks internally and the choice of reuse organization is heavily dependent on the company policies.

The framework must be treated as a product even if it is intended to be used internally. It must be well documented and the support of the framework must be planned.

If the framework developers will use the framework they will have the insight of the problems and the limitations of the framework. Also, they will have few problems of understanding the intentions behind the architecture and the solutions. The classical resistance of reusing other peoples solutions is also avoided and the well needed feedback from the users to the developers is easily achieved.

Development teams should, as much as possible, consist of experienced engineers, but this is seldom possible due to limited personnel and economical resources. By having separate teams of framework developers and application developers the knowledge of the experienced engineers in the development team will be reused by the, perhaps, more inexperienced engineers in the reuse teams. This is the main argument for separate organizations.

3.2.4 Summary

There are some differences between ordinary software development and development of frameworks which have an impact on the project organization.

Frameworks should be seen more like a strategical investment than an operational investment. The development of a software product will give an income after a relatively short period. A framework is more like a tool for the development of software products.

When developing frameworks it is necessary to focus on what functionality is general and what is specific, and this will have impact on the work organization. The ability of finding generalisations is dependent on an overview of the system or the domain which limits the possibility to divide work among several teams.

3.3 Preparing Framework Development

This chapter is intended to give a brief introduction to the activities performed before the framework development process is started and what input is needed to begin the development process.

The development team needs to possess extensive knowledge of the domain the framework is intended to capture. Therefore a domain analysis of some sort should be performed before or as an initialization of the process.

The domain analysis will form an input to the framework development process. However, the development process should also provide feedback to the domain analysis to make the analysis more complete.

3.3.1 Domain Analysis

Domain analysis is the identification of classes and objects that are common to all applications within a given domain [Karl95, p. 297]. The domain model should only focus on key domain artifacts and not deal with details [Karl92, p. 298]. A domain model is a good tool when starting to develop a logical view of the system. It should describe the concepts people use within the domain, and make the domain analysis an instrument for communication between the people involved in the system development by providing a common terminology. The domain model should *not* describe the domain from the developers point of view, since this will hinder communication and risk that details of design are emphasized too early in the development process.

A domain analysis also provides good support when specifying use cases [Jaco92, p. 162].

There are at least two documents that should be a result from the domain analysis: The scope of the domain and a static model containing the important objects and classes from the world of the domain.

It is important to formulate a distinct scope of the domain, because it is not possible for a framework to cover the whole world.

The scope is of much use in the capturing requirements activity of the development process. The scope makes it clear if a requirement is in the domain and valid, or outside the domain and invalid. The scope of the domain will also work as a tool in the reuse of a framework, when deciding if a framework is suitable to reuse for a required application or not.

It is often difficult, when formulating the scope of the domain, to decide what should be outside the domain and what the domain should include.

It is easier to develop a framework for a narrow domain than for a very large domain. Enough time must be devoted to this very important activity.

The static model should contain the most important objects and classes of the domain. These should be real world objects, objects from the world of the application. The objects and classes should be named from the users perspective because the model will be an instrument for communication between the developers and the users of the future application.

3.3.2 Summary

The domain analysis provides input to the framework development process with the documents: The scope of the domain and the static model.

The scope of the domain is a good tool when validating requirements in the capture requirements activity of the framework development process. The scope is also useful as a search-criteria in the activity of finding a framework suitable for reuse when developing with reuse.

3.4 Capture Requirements and Analysis Phase

This section covers the Capture Requirements and Analysis phase of framework development.

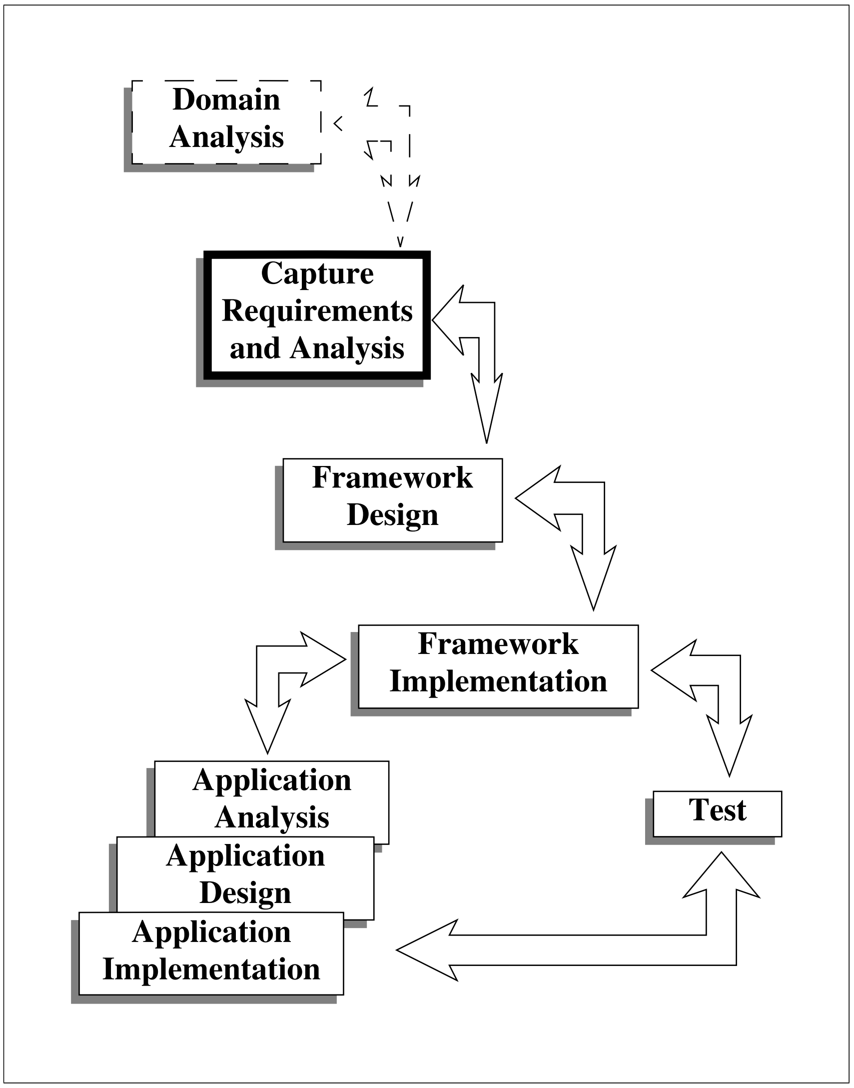


Figure 3.3 Process roadmap

The section begins with an introduction to the *capture requirement and analysis phase* pointing out the required input to the phase and the goals of the phase. The guidelines provided summarise the text above the guideline and are provided to promote a good framework analysis. Section 3.4.1 describes the requirements specification activity and the section consists of three sub-sections:

- 3.4.1.1 Requirements Process, which describes the process of finding and validating requirements and the identification of generalizations.

- 3.4.1.2 Requirements Specification, which describes the document with the same name.

- 3.4.1.3 Use Case Model, which describes the use case model together with a brief introduction to the concepts of use cases

Section 3.4.2 describes the analysis activity and the section consists of two sub-sections:

- 3.4.2.1 Performing the Analysis, which describes the process of identifying the static structure of the framework.

- 3.4.2.2 Static Object Model, which describes the product of the analysis activity.

These activity sections are followed by section 3.5.3 which points out the necessity of having the right, easy-to-understand models. The chapter is concluded with a summary.

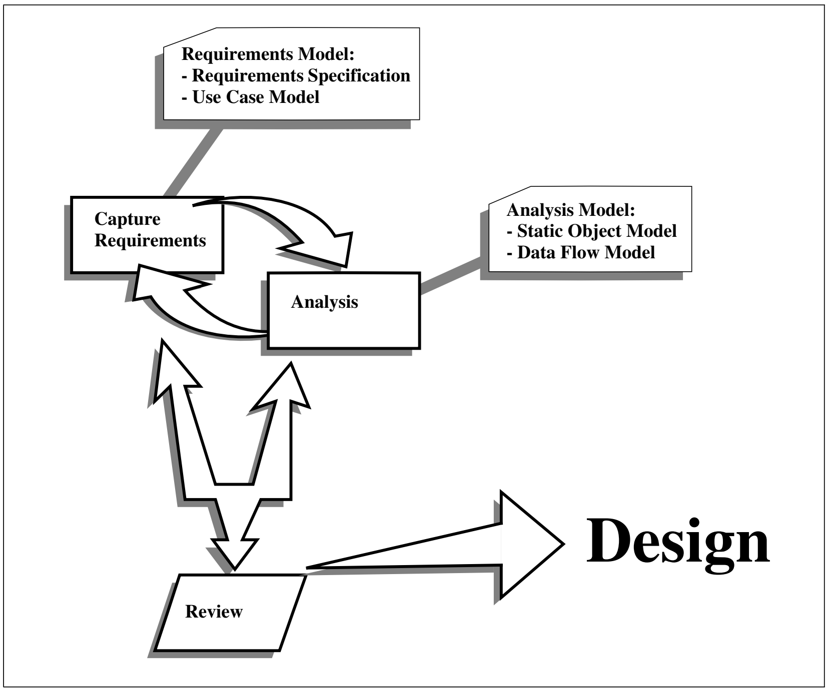


Figure 3.4 Capture Requirements and Analysis phase with its subprocesses and products.

The goal of the Capture Requirements and Analysis Phase is to capture all valid requirements and outline an ideal system that will fulfil these requirements. The phase consists of two main activities: the *Capture Requirements* activity and the *Analysis* activity. The two tasks are illustrated as they are sequential which is only partly true, the activities are much done in parallel.

The products of the analysis phase are the *Requirements Model* and the *Analysis Model*, see Figure 3.4. The requirements model will specify the requirements imposed on the system and the analysis model will outline the main concepts of the system.

A requirement specifies a constraint on the system or a service the system should provide. The requirements are tools in the process of making the correct analysis models. In the process of producing the analysis model new requirements will be identified and inconsistencies in the requirements model will be found. It is not possible to first find all requirements and succeed in making them consistent and then, with the requirements models as inputs, construct a correct and complete analysis model, therefore the two activities need to be done in parallel.

A domain analysis together with a list of requirements should be provided as an input to the analysis phase and they should concern at least two applications together with the future requirements of the framework. Providing requirements on a couple of applications would make it easier to find generalisations.

- **Guideline 1:** A list of requirements on at least two applications should be provided together with a list of requirements on the framework.

If the future applications of the framework are well defined it will be easier to develop a good and well adapted framework. If the future applications of the framework is very vague it should be considered if a framework is feasible, because vague future requirements implies that it is very uncertain whether there will be a demand for reuse of a framework in that particular domain or not. If there will not be any need for a framework to reuse, it should be considered if the extra effort that the development of the framework will require is economically justified.

Vague future requirements will also make the development of a framework very difficult. Developing a framework without any knowledge of its future applications will almost certainly lead to a framework that will be hard to reuse.

- **Guideline 2:** A list of future requirements on the framework should be provided.

There should be a team working with requirements and the analysis, not one single person. A single person will have difficulties in capturing all requirements and aspects of a framework and its future applications. The development team should include members with extensive knowledge of each application area and a member with knowledge of framework design.

3.4.1 Capture Requirements

The goal of the activity *capturing requirements* is to find all requirements on the system which is intended to be developed. Inconsistencies between requirements, requirements which are contradictory or ambiguous should be found and be resolved.

The *list of requirements* is the base from which the process of capturing requirements is started. The *domain model* is an instrument for communication and it provides a common terminology reducing the errors due to misunderstandings in the discussions with the interested parties.For further details on the domain model, see [Karl95] and [Mark94, p. 429].

The documents which will be the output from the capture requirements activity are the *Detailed requirements specification* and the *use case model.* These two models together form the *requirements model*. The requirements models are intended to be an instrument for communication between developers, procurers and users. Therefore it should be understandable to all of the interested parties. The requirements should though be formulated from the users or clients points of view and not from the view of the developers [Karl95, p. 282].

The requirements model is also intended to form a base to the testing and verification phase. The two documents proposed in this chapter are only a suggestion, other documents may be included in the requirements model. However, according to Heninger [Heni1980] the requirements documents should satisfy the following six requirements:

1. They should only specify external system behaviour
2. They should specify constraints on the implementation
3. They should be easy to change
4. They should serve as reference tools for system maintainers
5. They should record forethought about the lifecycle of the system
6. They should characterize acceptable responses to undesired events

3.4.1.1 Requirements Process

The goal of this process is to find all valid requirements on the system. The requirements process may be viewed as a cycle of three sub activities, elicitation, specification and validation.

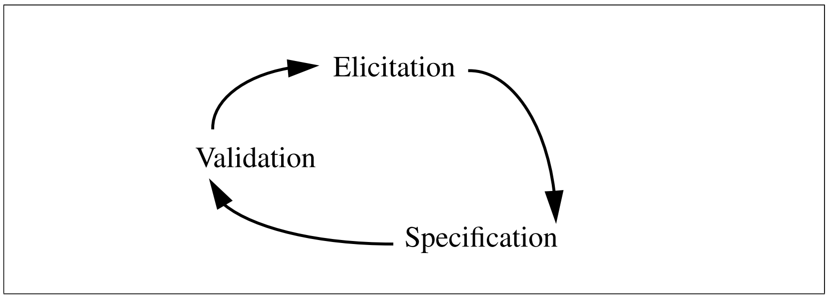


Figure 3.5

Loucopoulos and Karakostas [Louc95, p. 38] define these three sub activities as follows:

- *Requirements elicitation* is the process of acquiring all the necessary knowledge which is used in the production of the formal requirements specification.

- *Requirements specification* is the process which receives as input the deliverables of the requirements elicitation in order to create a formal model of the requirements.

- *Requirements validation* is the process which attempts to certify that the produced formal requirements model satisfies the user's needs.

Information should be gathered from all people concerned because different users will have different requirements on the system. A system based on the view of one person is not likely to fulfil all requirements imposed on the system [Karl95].

Usable information can also be found in old products such as specifications, analysis, designs, code, test cases and so on [Karl95, p. 298].

The requirements process is a very important phase in the development process because failure in finding all requirements, and finding the correct ones leads to later changes in the following phases. The cost of repairing errors due to changes in the requirements is very high since much of the design and code has to be rewritten. The cost increases for every phase the error passes undiscovered [Som92, p. 86]. Therefore much effort should be put in the analysis phase ensuring a correct, complete and consistent requirements specification.

Analysis Team

A good approach to cover as many information sources as possible is to let the team cover different roles of the stakeholders. One way is to include people from these different areas, like market people, users and developers, into the project team. Another approach is to let the project members take on different roles, perhaps more than one role per member. The latter approach leads to smaller project groups which improves the communication between the project members but the drawback is that the project member not always has the knowledge to succeed in capturing all aspects of his or hers roles. A suggestion of the roles to cover are people from the product management, the developers and the market people. There are, of course, sometimes a need to cover other roles, it depends on the nature of the system.

- **Guideline 3:** Include members with knowledge of each application and a member with knowledge of framework design into the analysis team.

Elicitation

Even if the project team include people from all interested parties is it seldom enough with the knowledge covered by the project team. Knowledge of other people concerned has to be captured. Conducting interviews is the most traditional way of working. An alternative of making interviews intended to capture the requirements and knowledge of all people concerned is to perform a Group Dynamic Modelling session [Will91]. Making interviews is very time consuming. A series of interviews has to be made during a rather long period of time. In a GDM-session all people concerned are gathered to one place during a day or two to make things out. The total cost in man time is roughly the same, but synergy effects are gained when many people meet and the time period during which the collection of requirements is done is shortened.

- **Guideline 4:** Gather information from as many different sources as possible to acquire knowledge of which requirements are of importance.

Validation

All requirements should, as mentioned above, be found. However, the requirements should be correct as well. To ensure the correctness of the requirements they have to be validated. Sommerville claims that the validation process include four steps [Som92, p. 97]:

- The needs of the user should be shown to be valid

- The requirements should be consistent, a requirement should not be in conflict with another requirement
- The requirements should be shown to be complete, the requirements should cover all functionality the system is intended to provide and all constraints imposed on the system
- The requirements should be shown to be realistic and realizable

The validation should be carried out *during* the requirements process not at the end of the process [Som92, p. 98].

Formulating the functional requirements with use cases makes the requirements easier to verify if the requirements are accomplished or not.

Find Generalizations

The main goal for the whole development process is, as mentioned earlier, to produce a framework. The main strategy to accomplish this goal is to focus on what is general between the applications, and what is specific for each application. The framework is formed by the architectural constructs, algorithms and data that are common to all applications that are intended to be covered by the framework.

The process of capturing requirements follows this strategy. Find all unique requirements and isolate them, then collect all common requirements as framework requirements. The main strategy to accomplish this is to identify all requirements of each application and make separate lists of requirements for each application of the framework. The next step is to identify all general requirements of the applications and then move these invariants into the framework requirements. The separation of requirements makes identification of common behaviour and commonalities in the requirements of the applications easier.

There is, as mentioned above, no sharp border between the process of capturing requirements and the process of analysis. Preliminary analysis models are constructed for the purpose of finding new requirements and validating existing ones.

3.4.1.2 Requirements Specification

The requirements specification is one of two documents of the requirements models.

Separation of Requirements

It is our opinion that the requirements should, in the requirements specification, be separated into two categories; framework requirements and application specific requirements and then into functional requirements and non-functional requirements, see Figure 3.6. It is suitable to make a distinction between these different types of requirements since the separation of requirements makes identification of common behaviour and commonalities in the requirements of the applications easier.

- **Guideline 5:** Separate the requirements into framework specific and application specific requirements

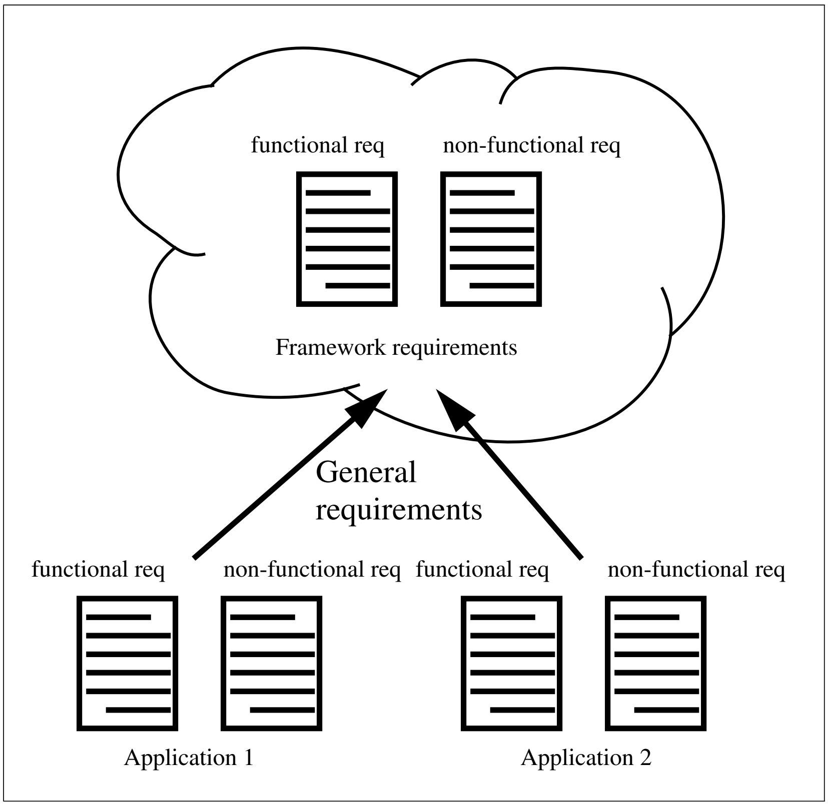


Figure 3.6 The division of requirements in functional and non-functional requirements and the separation of application and framework requirements.

Application specific requirements include all functional and non-functional requirements that are specific for each application. The framework requirements include all requirements, functional and non-functional, that are general between the applications.

If there are commonalities between some of but not all of the applications these commonalities may be grouped into sub frameworks.

Functional and Non-functional Requirements

Functional requirements are the requirements that specify the functionality the system will provide. The non-functional requirements specifies other constraints placed on the system. Such constraints may arise because of company policies, standards, constraints imposed by other systems and so on [Som92, ch. 5.2].

Sommerville points out three different classes of non-functional requirements:

- Product requirements, such as performance, size and portability
- Process requirements, like standards, naming conventions and so on
- External requirements, which cover all other non-functional requirements like cost requirements, requirements imposed by other systems, requirements which can not be categorized by the two classes above

The non-functional requirements on a framework are in general different to the non-functional requirements on the applications. The non-functional requirements of the frameworks are more design oriented than non-functional requirements imposed to applications. The reason is that the frameworks has different users than ordinary applications. The users of a framework are application developers. A framework is used to develop an application, which is developed to fulfil a user's need. In most cases it is the application developers who constitute the requirements on the framework. For example the developers may require the framework to be implemented with a certain language, or the framework implementation should follow certain standards, naming conventions etc.

The functional requirements are often easy to test and verify, especially when the functional requirements are formulated with use cases the testability is ensured.

Non-functional requirements are hard to formulate in a testable way. To give an example of a non functional requirement which shall formulate the adaptability of the framework: "New applications shall be easy to develop by modifying existing concrete classes or writing new concrete classes." This statement is not possible to test. The requirement should instead be formulated like:
"New applications should be able to be developed in a man week by modifying or writing new concrete classes." This requirement is testable [Som92, ch. 5.2].

Non-functional requirements may be hard to specify by using other than natural language because they tend to be very complex [Som92, p. 9]. Using natural languages introduces difficulties when finding inconsistencies between requirements, because related requirements may be expressed differently hiding the relation between them [Som92, p. 87]. Natural language may also cause misunderstandings between people involved in the development process because different people use different words for the same concept [Som92, p. 88].

- **Guideline 6:** The application and framework requirements should be divided into functional and non-functional requirements due to the different properties of the requirements.

3.4.1.3 Use Case Model

A use case model consists of actors and use cases.

The Use Case

A use case defines how the system will be used and what the system will perform in response to a certain input. Every use case is a specific way to use the system. Jacobson defines the use case as “Each use case constitutes a complete course of events initiated by an actor and it specifies the interaction that take place between an actor and the system.”.

We believe that the use cases should be separated, as well as the requirements, into specific and general behaviour. The separation makes it easier to identify what is general between the different applications and what behaviour that is specific to each application. The separation of requirements follows the main strategy of the framework development process, which is to focus on what is generic and what is specific between the given applications.

- **Guideline 7:** Separate the use cases into framework specific and application specific use cases. This enables to focus on what is general and what is specific between the given applications.

Actors and Users

Jacobsson et. al. defines in OOSE [Jaco92, ch. 6.4.1] the concepts of actors and users.

The *actor* is a modelling concept for human users or other systems and it is an aid to define what exists outside the system. Actors have instances, called *users*, which perform sets of operations on the system.

There are a correspondence between classes and actors as well as objects and users. The object is an instance of a class and a user is an instance of an actor.

An actor is non-deterministic, the *actor* may give several responses to a certain stimulus when in a specific state. The user do however perform behaviourally related sequences of actions in dialogue with the system. The sequences is behaviourally related to the role the actor or user is intended to perform [Jaco92, ch. 6.4.1]. An example of different actors: There are mainly two different user categories in a time reporting system, the ordinary employee who reports how much time he or she spends on different projects or activities, and an employee on the financial department who compile the time reports. These actors will perform sequences that are behaviourally related to their tasks, like a potential user will be using the future system when it is developed.

The Model

The functional requirements should, when possible, be formulated by use cases. Use cases make it useful to find general behaviour between applications, general behaviour which should be moved into the framework.

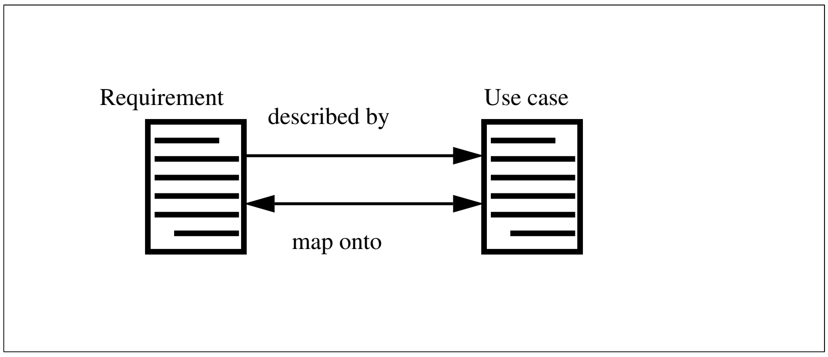


Figure 3.7 Requirements formulated in use cases.

The use case model is a good communication medium between users and developers because a use case is expressed in terms familiar to the users. A use case is also a good instrument in the activities of finding inconsistencies between different requirements since use cases are more formal than normal language. The increased formality forces similar requirements to be expressed similarly which makes it easier to identify relations between requirements.

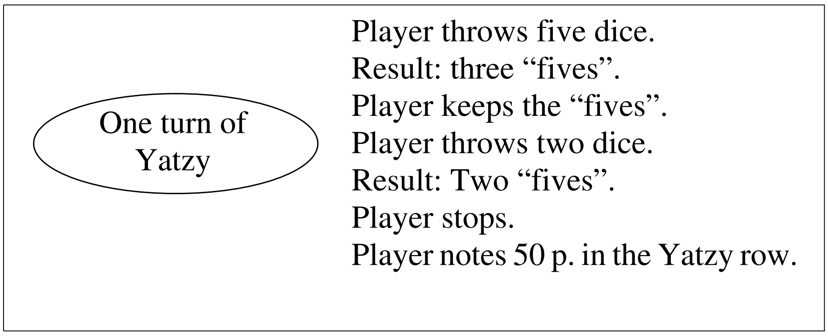


Figure 3.1 A use case of “One turn of Yatzy” together with its symbolic representation.

The use case model will also form a base for the testing process. If the requirements are formulated by use cases and the tests are designed according to these use cases then there will be a direct relationship between passing the tests and fulfilment of the requirements. The relationship between the use case model and the other models of the development process is visualised by figure 3.6 [Jaco92, p. 132].

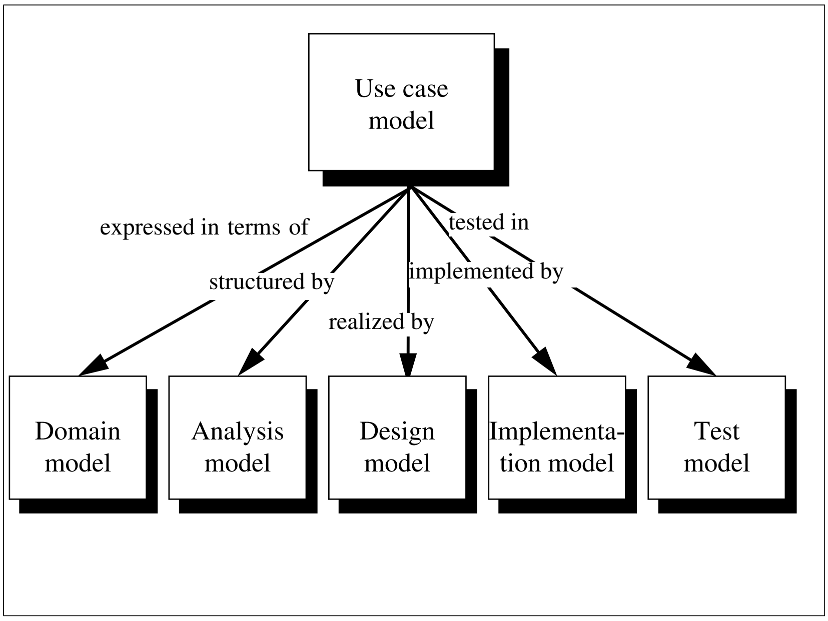


Figure 3.8 The relations between the use case model and the other models of the system development process.

3.4.1.4 Designing Use Cases

The process of finding use cases is iterative as most other construction processes in software development. However, the first step to perform is to find the actors who interacts with the system. A good way to find actors is to focus on the purpose of the system and on how the system will be used. All those actors are not found at once, there will almost certainly be new actors identified during the requirements and analysis process. More and more actors will be found as the system becomes clearer [Jaco92].

Jacobson categorizes actors into primary and secondary actors. The secondary actors exist only to support the primary actors use of the system. The primary actors are the actors the system is intended for and, thus, the most important users of the system. [Jaco92, p. 154].

The next step is to identify the use cases by viewing the requirements from the users perspective and perhaps continue with interviews with the real-world user the actor is intended to model. In Jacobson et. al. [Jaco92, p. 155] a number of good questions are presented, which answers will lead to the identification of use cases:

- What are the main tasks of each actor?
- Will the actor have to read/write/change any system information?
- Will the actor have to inform the system about outside changes?
- Does the actor wish to be informed about unexpected changes?

It is hard to tell how detailed the use cases should be. It is often not obvious when to stop and there is no limit on how detailed a use case can be. Generally it is better to have a few detailed, more extensive use cases, than many short ones.

Extends

Most of the use cases are, in at least a small part, variants of other use cases. There will be many use cases that only differs in just small parts. This leads to a unnecessary large use case model and it may be hard to relate these use cases to each other. By only modelling the differences between different use cases the use case model becomes more perceptible. In OOSE this is done with a modelling concept called *extends* [Jaco92, p. 158].

The extend concept may be a good tool to isolate differences, when designing use cases for framework development.

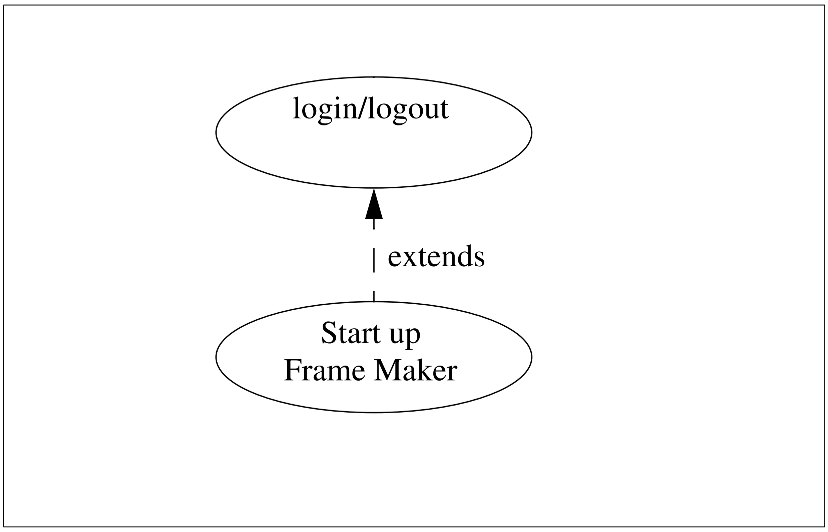


Figure 3.9 The extends concept.

In the example above, Figure 3.9, there is a use case which only actions are *login*, followed by *logout*. This use case is then extended with a start-up command to Frame Maker, which results in a new use case containing the action sequence: *login, start-up framemaker, logout*. Now there are two use cases and the information which is common to both the use cases is only modelled once and the model contains no redundant information.

3.4.1.5 Use Cases and Object-Oriented Frameworks

It is, as mentioned above, suitable to divide the use cases into use cases specific to each application of the framework and into use cases general to these applications. Use cases that are general should be moved into the use case model of the framework and use cases which are application specific into the use case model of the application to which they belong.

A concept that we think support this activity is to accomplish *abstract use cases*.

Abstract Use Cases

Abstract use cases is another concept that origins from the OOSE methodology [Jaco92]. The use cases are divided into abstract and concrete use cases. Concrete use cases are the use cases that will be initiated by an actor to produce a result. An abstract use case contains a sequence of actions that are shared by several concrete or abstract use cases. The abstract use cases will not be used directly by an actor, they will be used only by concrete use cases, or other abstract use cases.

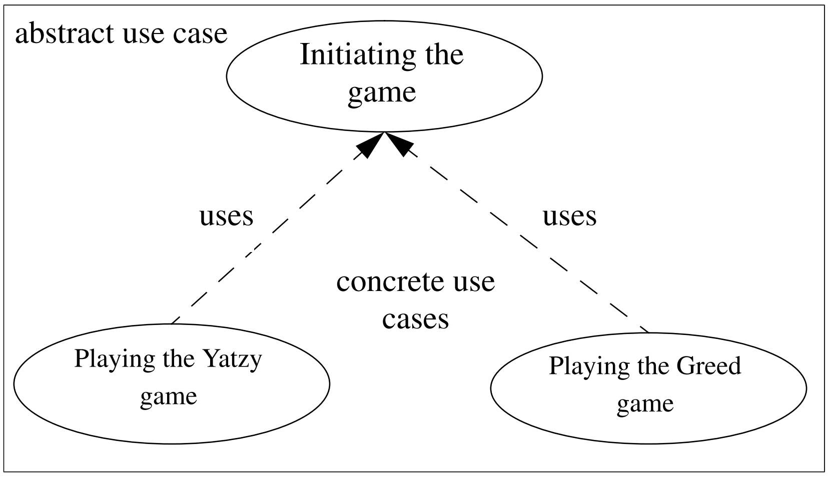


Figure 3.10 The concept of *abstract use case*. Both when playing a game of Yatzy and when playing a game of Greed some common initializations needs to be made.

A concrete or abstract use case may use several different abstract use cases to complete it's sequence of action. As many abstract use cases as possible, which are common to several use cases, should be found. The next step is to identify sequences which are shared by several of these abstract use cases, which were identified earlier. These sequences will form new abstract use cases. This last step iterates until no more common sequences are found. The result of this activity will be a hierarchy of abstract use cases.

We believe that such a hierarchy of use cases will map onto a typical class hierarchy, a class hierarchy that for example forms a framework. Use cases shared by several applications will form a hierarchy of abstract use cases. The abstract use cases are sequences of action that much likely will be performed by the future framework.

The process to identify a use case hierarchy mapping onto a framework should include the following steps:

1. Identify commonalities between the use cases of an application.

2. Repeat step one for each application of the intended framework.

3. Identify commonalities between the concrete and abstract use cases of all applications of the intended framework.

4. Repeat step three until no more general abstract use cases are found.

This hierarchy should then form the use case model of the framework. The use case model is the foundation from which the common parts are identified and isolated into the framework.

3.4.2 Analysis

The goal of the analysis is to outline a model of a system which fulfils the requirements. The analysis should focus entirely on the problem and be done without consideration to the implementation environment. The reason for this approach is that the analysis model should remain relevant even if the implementation environment will change. Another, even more important reason is that the implementation details would risk to put the developers focus on implementation problems and put the problem that the system is intended to solve out of focus [Jaco92, ch. 7].

Once you have identified the problem domain and the requirements have defined which part of the problem domain the framework (or system) is intended to capture, the system has to be outlined and the frameworks within this system should be identified.

The analysis models includes a *static object model*. The analysis models are built of real world objects just like the domain analysis. Objects present in both the domain and the analysis model should be named the same in both the models to ensure traceability and to decrease the amount of errors due to misunderstandings.

The idea with a model is to capture the concepts of importance and filter out those of no importance. All abstractions are subsets of reality selected for a special purpose. This makes it easier for the developers to focus on the problem without irrelevant details hiding the problem. Every model should have it’s special purpose. Models supporting framework development should have the ability to focus the developers attention on what is similar between the applications to be developed and what is not.

3.4.2.1 Performing the Analysis

The process of producing the analysis model is iterative in nature and a model suitable as a base for the design phase is achieved by successive refinement and an increasing degree of formalization [Karl95]. There is no sharp edge between analysis and design. Some activities that normally belongs in the design phase are done in advance during the analysis phase in the purpose of finding all classes and important relations in the analysis models.

Outline the situation and the problem, describe them from the user's perspective. Once the situation and the problem is outlined, it should be possible to identify necessary abstractions and begin the construction of the analysis models[Tal94a].

The analysis process should include the following steps according to Taligent [Tal94a]:

- Outline the situation and the problem.
- Examine existing solutions.
- Identify key abstractions.
- Identify high level abstractions.
- Identify what parts of the problem the framework will deal with.
- Ask for input from clients and refine the approach.

In the process of refinement should classes from the domain model which are not needed be removed. New necessary classes should be introduced as well as, when possible, higher levels of abstraction. Introducing higher levels of abstraction leads to increased generalization of the system [Karl95].

- **Guideline 8:** Remove redundant classes to refine the model from unimportant information.

- **Guideline 9:** Identify high level abstractions preparing for the identification of the framework.

By introducing high level abstractions more commonalities between the applications are found, commonalities which should be moved into the framework.

High level abstractions makes the component more stable to changes in the requirements. A component with an architecture containing high level abstractions may be changed without restructuring of it's architecture. Needed changes are introduced by creating specialization of the high level abstraction [Karl95, p. 302].

Requirements will always change because the world surrounding the software system is always changing. The changes of the surrounding world will reflect in changes in the requirements imposed on the system [Som92, p. 534]. Thus finding generalizations is of great importance not only in framework analysis.

Finding these abstract classes is the first step in the analysis activity of identifying the frameworks in the system.

The easiest way to identify the abstractions is with a bottom-up approach. Start by examining existing solutions. Examine existing solutions or systems may generate useful knowledge and provide important information about the possible frameworks.

- **Guideline 10:** Examine existing solutions to gain knowledge of possible frameworks.

Analyse the data structures and algorithms and then organize the abstractions. Always identify the objects before you map out the class hierarchy and dependencies. Identify what the solutions have in common and what is unique to each program. Taligent [Tal94a] suggests that potential frameworks could be found in:

- Real-world models.
- Activities performed by end users.
- Source code for current software solutions.

Some of the generalizations identified may be introduced in the framework to increase the framework's generality according to future requirements These abstractions may not exists in the applications under development. However, it is often difficult to decide if a generalization is necessary or not. A generalization increases the complexity and may increase the development and reuse costs. Therefore there must be a trade off between the generality and the complexity of the framework. As a rule the introduced high level abstraction should be within the domain of the framework.

- **Guideline 11:** Introduce only abstractions which are within the domain of the framework.

Two examples in the domain of dice games: It is a proper generalization to let a *die* have any number of sides and the make a specialization of this general die to achieve an ordinary die with six sides. It is probably not a proper generalization to make a high level abstraction of a *dice player* to achieve an high level abstraction *player.* A player which may be specialized into a *hockey player* is outside the domain.

Decisions about generalizations should, in not obvious cases, be documented and motivated.

The frameworks should not be to big. Big frameworks should instead be decomposed into smaller more focused frameworks. Smaller frameworks is easier to reuse [Tal94a].

- **Guideline 12:** Structure large frameworks into sub frameworks. Small frameworks are in general more focused than large ones.

3.4.2.2 Static Object Model

A goal of the *static object model* is to capture the objects, the relations between objects and other concepts of the real world that are of importance to the application we have the intention to build [Rumb91 p.17]. The static object model should provide a graphical model easy to understand, suitable for communication both between developers and between developers and customers.

The static object model should not contain any computer constructs unless the problem to be solved is a computer problem. The naming of the objects and concepts in the model should be done from the users perspective [Rumb91, p. 17].

As mentioned above, objects present in the domain model should be named the same in the static object model. The static object model should be a reference document, not only throughout the development process, but also in the maintenance phase. Therefore, naming of the objects and concepts in the model should be done with great care.

- **Guideline 13:** Abstractions present in the domain model should be named the same in the static object model ensuring traceability.

According to Rumbaugh [Rumb91] the static object model should include analysis objects and the associations between these objects. Aggregational relations are not yet of importance but those found should be introduced in the model. Suitable inheritance structures should be found and attention should be paid to find structures and objects common to more than one of the applications intended to be captured by the framework.

It is suitable to develop a static object model for each application. The model development should be done for all applications in parallel. When common abstractions occur they should be introduced in the static object model of the framework.

- **Guideline 14:** Develop a static object model for each application.

- **Guideline 15:** Introduce abstractions common to several applications in the static object model of the framework.

3.4.3 Complementary Results and Models

The models presented in the chapter presenting the analysis phase are not the only useful models and they may not be suitable for every organization. However, these models presented are common in most development methodologies but every analysis requires a special set of models. Sometimes not every model presented in this chapter is needed and sometimes they are not enough. Use models that cover the needs and if the models used not highlights the properties of importance, use an additional model.

The models should support identification of general concepts, be as easy to understand as possible. The notation used should be kept simple with no room for misunderstandings. Graphically models are good. One example could be the use of different colours in the object model to express concepts, relationships and so on. An model easy to understand reduces errors due to misunderstandings and it is necessary for effective reuse that the reuser will understand the model quick and easy [Karl95].

- **Guideline 16:** Use graphical notations. Graphical notations make the models easier to understand.

The models should be easy accessible to all members of the project because it should be easy to discuss and refer to the models. It can be achieved by presenting the models on large sheets on the wall.

- **Guideline 17:** Present the models clearly visible to all project members making the models easy to discuss.

3.4.4 Summary

The goal of the capture requirements and analysis phase is to identify all valid requirements and then outline an ideal system which fulfils these requirements.

3.4.4.1 Capture Requirements

In most software development processes there exists an activity of capturing requirements, the framework development process is no exception. All requirements should be found during this activity and they should be validated to ensure correctness and consistency.

The analysis team is somewhat different from an ordinary analysis team. The analysis team should have knowledge of the domain and of each application that is intended to be developed using the framework. The analysis team should also have knowledge of framework development.

It is suitable to apply use cases to describe the requirements. Use cases may be directly tested during the tests in the verification and validation phase, thus it may be directly verified if the requirements imposed on the system are accomplished or not.

The biggest difference between the capture requirement activity for framework development and for ordinary software development is the focus on which requirements that are general for a set of applications and which requirements that are specific for each application.

One goal of the capture requirement activity is to isolate all requirements general between the applications and to let these requirements be the requirements imposed on the framework.

The above presented use case concepts *abstract use cases* and *extends* are good tools which support this activity of isolation of general requirements.

The product of the capture requirements activity is the requirements model which consists of the requirements specification and the use case model.

3.4.4.2 Analysis

The presence of the analysis model is not unique to the framework development process. There probably exist one analysis activity in every development process.

The goal of an analysis activity is to outline a model of the system to be developed. The analysis should entirely focus on the problem without consideration of the implementation.

The big difference between the framework analysis activity and an ordinary analysis is again the focus on what concepts are general between the applications and what is specific to each application.

All abstractions that are common between the applications are moved into the framework. The method used to identify the more obscure common abstractions is to introduce high level abstractions. It is suitable to develop a static object model for each application. The model development should be done for all applications in parallel and when common abstractions occur they should be introduced in the static object model of the framework.

When high level abstractions are used to build an architecture the architecture will be more stable with respect to changes in the requirements.

Frameworks should not become too big. It is better to divide a large framework into several small frameworks. Small frameworks are easier to reuse.

The product of the analysis activity is a static object model of the framework and one for each application. The static object model consists of real world abstractions, high level abstractions and the relations between these abstractions.

3.5 The Design Phase

This section covers the Design phase of framework development.

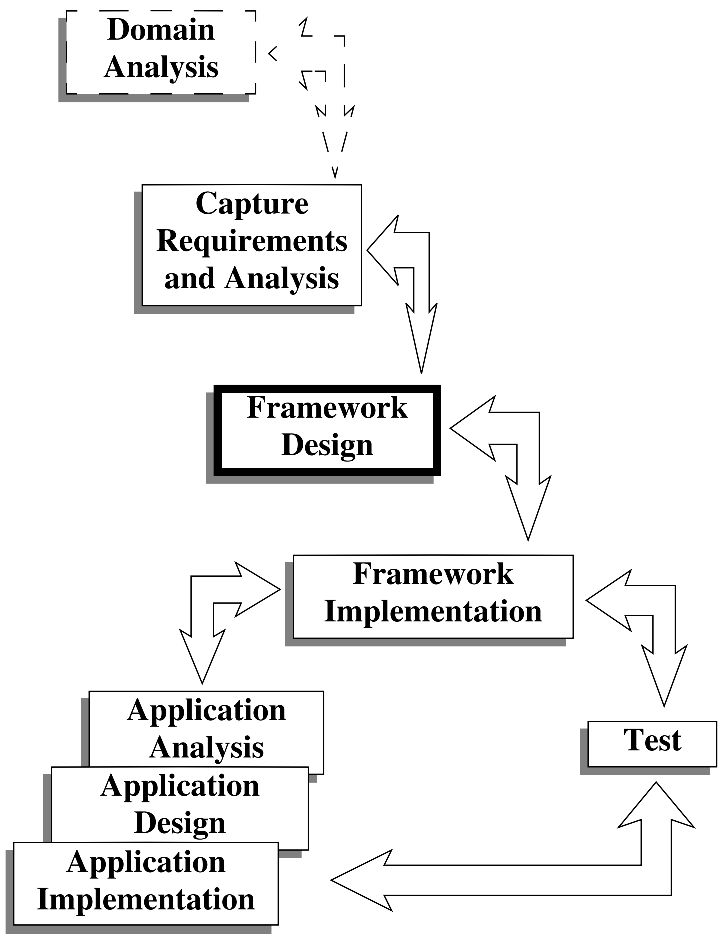


Figure 3.11 Process roadmap

The section will give a motivation for and a description of the design phase, as well as a description of the special concerns in framework design. The section is opened with an introduction to object-oriented design and the certain concerns of framework design, followed by a description of the framework design process.

The guidelines provided in this section are intended to promote a good framework design.

The design phase encompasses an architectural design phase, where the objects and their collaborations are defined, and a detailed design phase, where the classes and their methods are described in more detail.

The output from the design phase is a static object model and dynamic models that describe the collaborations. These models should constitute an adequate base for the implementation of the system.

3.5.1 Object-Oriented Design

The reason for having a design phase, and not to start writing code directly after analysis, is that the analysis models are inappropriate as a basis for source code writing. The analysis models view the system from a conceptual point of view, without regard to the implementation environment. To provide a firm ground for the implementation, the objects has to be refined, and the models have to be extended. Among else it has to be determined what operations shall be offered and exactly what the communication between the objects looks like. The design also serves to validate the analysis, and unclarities that are discovered might result in a return to the analysis process. [Jaco92, p. 196]

The analysis models may have to be changed in various ways to adapt to the implementation environment. These changes should be conducted with care. Changes should add or change functionality concerning the implementation environment, and changes to other functionality belong in the conceptual, logical object modelling, conducted during the analysis phase.[Jaco92, p. 206]

Subsystems

Subsystems are used to manage large software systems. The subsystems group objects to a larger unit, and thereby reduce the complexity of the system [Jaco92, p. 190]. A subsystem should be a part of the system under development that can be designed and implemented independently, and is likely to be affected by the same minor change in requirements.

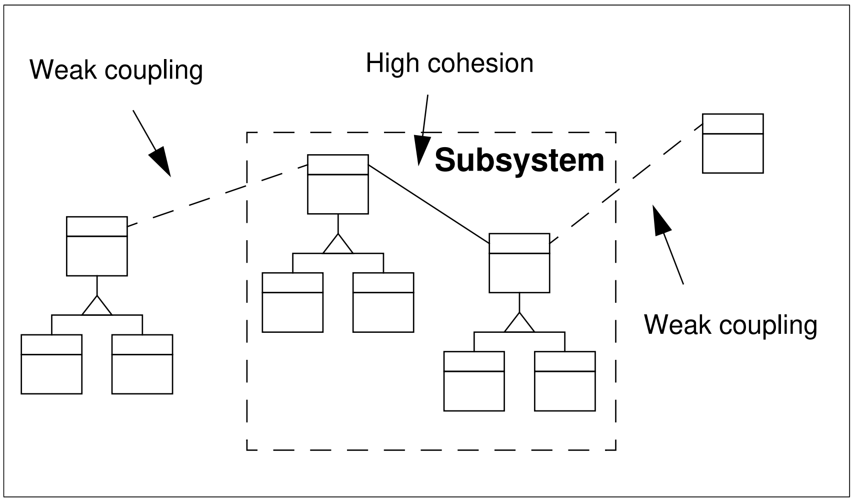


Figure 3.12 The classes in a subsystem should have high cohesion, and the coupling to the classes outside the subsystem should be weak.

The classes in the subsystem should have high cohesion, i. e. fit well together, and implement a single logical entity, to which all classes shall contribute [Som92, p. 183]. The coupling, the number of collaborations a subsystem has with other classes or subsystems, should be minimized [Karl95, p. 307]. Coupling is a measure of dependency between classes. Strong coupling arise when many different messages are passed, or one message is passed frequently. A subsystem with high cohesion and weak coupling is illustrated in Figure 3.12.

- **Guideline 18:** Subsystems shall have high cohesion and weak coupling.

Late introduction of subsystems inhibits early division of work among teams, but a detailed object and class design allows better division into subsystems [Karl95, p. 307].

3.5.2 The Framework Design Process

A framework design is a software design that, when implemented, provides the general and abstract functionality identified in analysis. The framework will be an implementation of the general parts of the applications in the domain [Joh95].

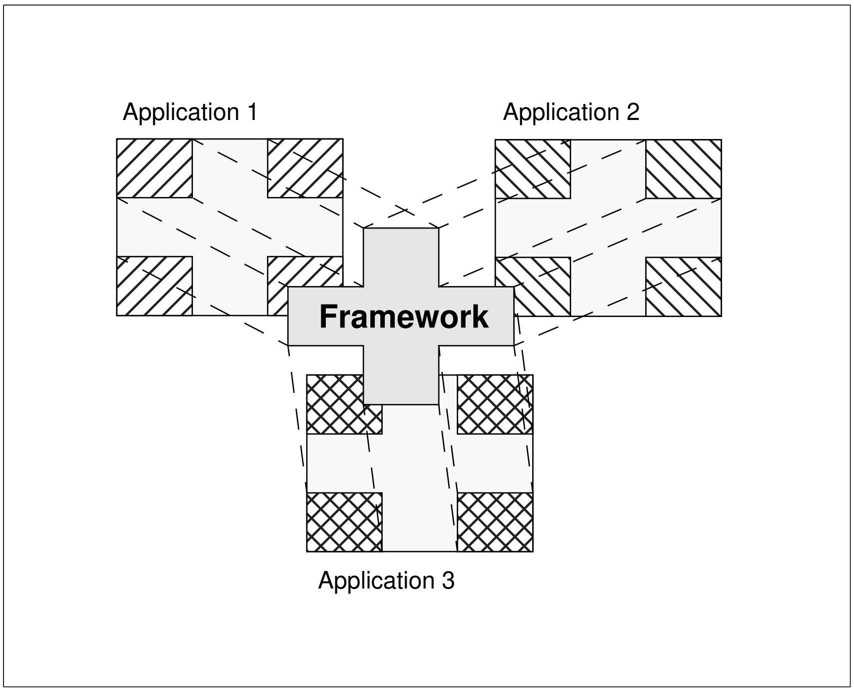


Figure 3.13 A framework captures the general parts of the applications in the domain.

The framework design subprocess of the framework development process consists of architectural design and detailed design. During the architectural design, the object and their collaborations will be changed, as consideration is taken to the implementation environment. During the detailed design phase, the objects identified in architectural design are described in the target implementation language and if necessary, the objects are refined [Karl95, p. 281-282].

The chronological order of the activities in the design phase is shown in Figure 3.14.The design is continuously reviewed, and suggested design solutions might be proven by prototyping.

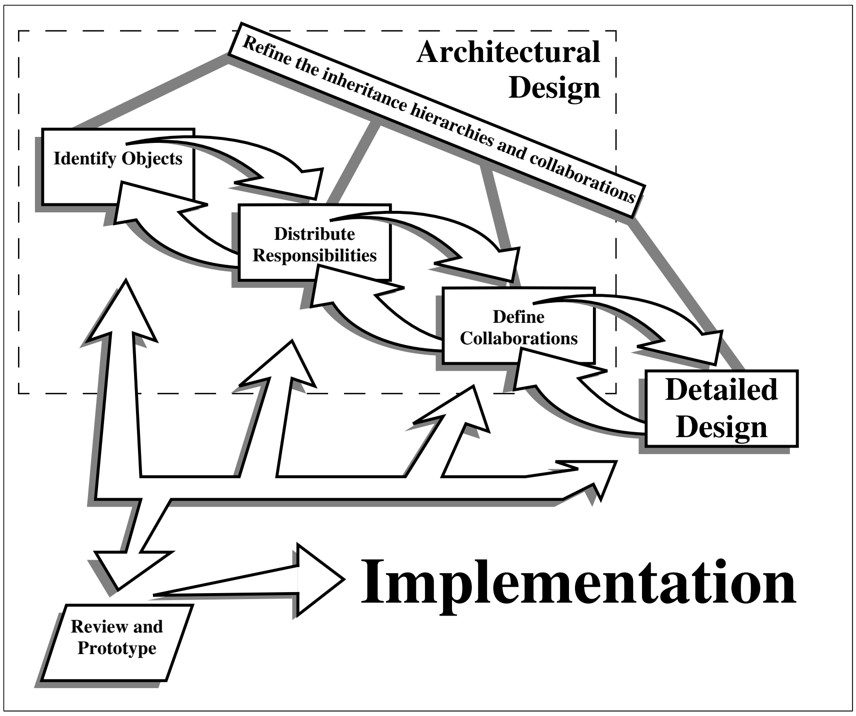


Figure 3.14 The process roadmap. The design is continuously reviewed, and design solutions might be validated by prototyping.

The main issue during framework design is to provide a base for a generic implementation, that applies to several similar applications. During the design process, many abstractions will be identified, and therefore the design has to be easy to change.

Identify abstractions

Most of the concepts common to the applications will have been identified in domain analysis and analysis, and the abstractions found during the design phase will probably be at a lower level. The identification of a high level abstraction during design might result in a return to analysis.

Search for key mechanisms of the applications that can be captured in the framework. Try to abstract as much as possible of these mechanisms into the framework

The fact that abstractions are found "bottom-up", by studying concrete examples, implies there has to be a design to find an abstraction [Joh95]. The design could be an overview in the designer's mind, a prototype design or an old application in which the same design problem is solved.

Identify Generic Design Solutions

There are no benefits in designing the same thing twice, or in doing two complete designs to solve two similar problems. A generic design solution solves the current problem, but also takes similar problems that may occur into consideration.

Seeking to reuse previous design solutions will limit the need for complex design decisions. If a design problem is similar, or identical, to a design problem already solved, the previous design solution should be reused.

The design knowledge available in the organization should be (re)used to the maximum extent.

- **Guideline 19:** Study existing frameworks and generic designs, and try to reuse all available design knowledge.

Design patterns are generic solutions to problems that often occur in framework design [Gam94]. The design patterns have been applied to many designs and the solutions they suggest are well-proven. Design patterns also ease communication between design teams and make the framework easier to understand [Laj94; Gam94].

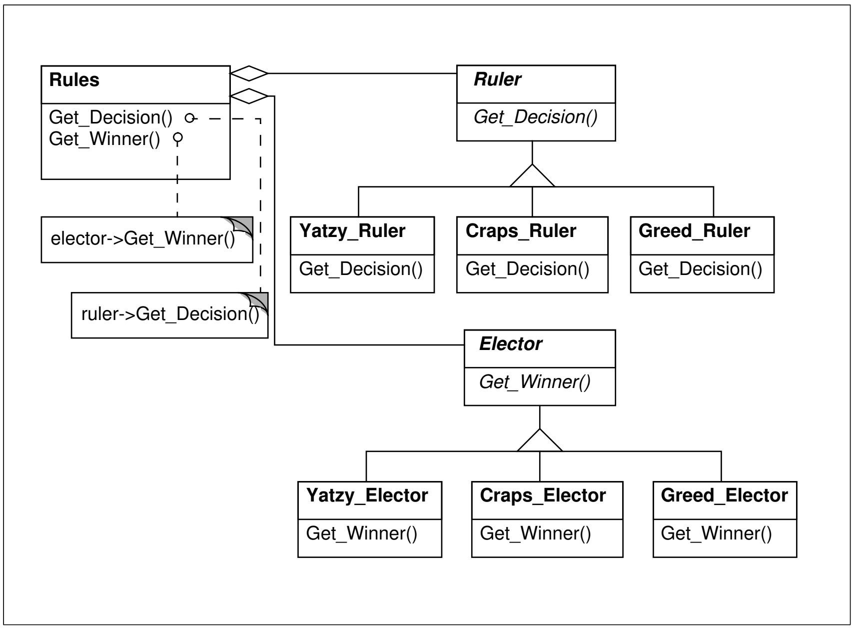


Figure 3.15 The design pattern “Strategy” applied in the Dice Game framework.

Design patterns which might be applicable in the framework design should be investigated, and if a design pattern applies to a problem, the problem should be solved according to the design pattern. In the our case study, we applied the design pattern “Strategy” when designing the representation of the rules of the games. The rules only differed in the algorithms for decision making, and these algorithms were factored out into separate objects. The “strategy objects” are then used to compose instances of the “ruling class”. This approach makes the ruling class independent of the algorithms used to “calculate” the decisions. See Figure 3.15.

- **Guideline 20:** Each design problem to which a design pattern apply shall be solved according to that pattern.

If necessary, go as far as to implementing parts of the applications to validate the design solutions, and see that they really are general and useful.

- **Guideline 21:** Approve the design solutions by prototyping. If necessary, go as far as to implementation to validate the design solutions.

3.5.3 Architectural Design

During the architectural design phase, a high-level description of the framework and the applications is made based on the models provided by analysis [Karl95, p. 281].

The activities shown in Figure 3.16 should be common to architectural design in most object-oriented methods [Karl95, p. 305].

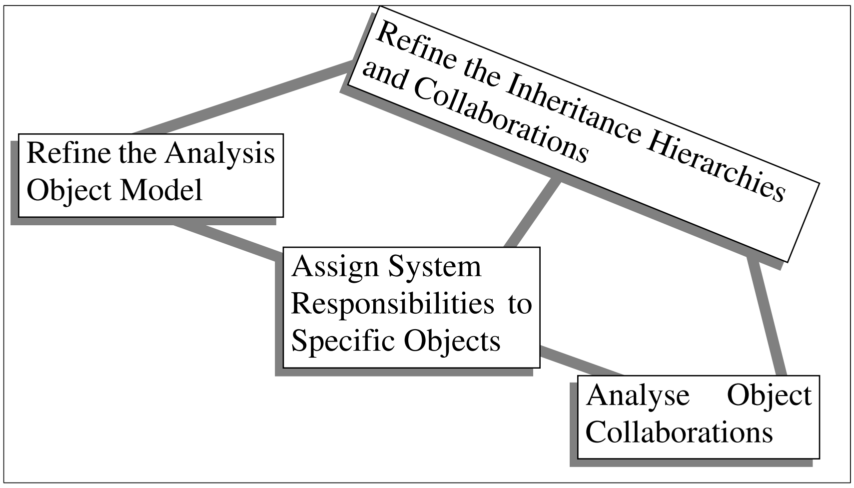


Figure 3.16 The activities of architectural design.

The objective of architectural design is to identify the objects needed to implement the system, and the way the objects collaborate. The system is also, if necessary, divided into subsystems during this phase.

The input to the architectural design phase are the requirements and analysis models, as described in the analysis section.

The architectural design phase produces output in the form of a static object model and dynamic models (interaction diagrams, state transition graphs and data flow models). These will form the basis for the identification of the implementation classes.

3.5.3.1 Refine the Analysis Object Model

In this activity, new objects, not present in the analysis models, may be introduced to adapt the system under development to the implementation environment. An analysis of the implementation environment should been done in parallel with analysis, or at least before the design phase is entered [Jaco92, p.196]. Other changes may be deleting, splitting or joining objects from analysis. Such changes should be conducted with great care, as they often tend to decrease the robustness of the system [Jaco92, p. 206].

It is important for the understanding of a framework to maintain the traceability between the analysis and design models [Jaco92, p. 117]. Many of the design objects may have been identified in the analysis phase, and these objects' names should be the same in both models.

- **Guideline 22:** Objects directly transferred from analysis should keep their names. To understand the framework from a conceptual point of view, the reuser should be able to trace the objects back to the analysis models.

A class represents an abstraction of the objects instantiated from it. If a class has many methods, it probably consists of several different abstractions. The amount of methods that indicate a large class varies, but more than 25 qualifies the class for examination [Joh88].

Classes with a large number of methods are not likely to be shared by several parts of the design. Parts of a class that is examined, and turns out to represent several abstractions, might be shared by parts of the design that did not share the original large class.

- **Guideline 23:** Keep classes appropriately small. Classes with more than 25 methods should be considered candidates for restructuring.

3.5.3.2 Assign System Responsibilities to Specific Objects

The responsibility of an object or a system has been defined as the "knowledge to maintain and actions that can be performed" [Wirfs90]. In this activity, the system responsibilities shall be distributed among the objects identified in the earlier phases.

During the identification of the operations an object is responsible for performing, and what knowledge it shall maintain, a common way to express similar responsibilities should be used, since this may help identifying abstractions.

- **Guideline 24:** State responsibilities as generally as possible. A common way to express responsibilities may help finding abstractions.

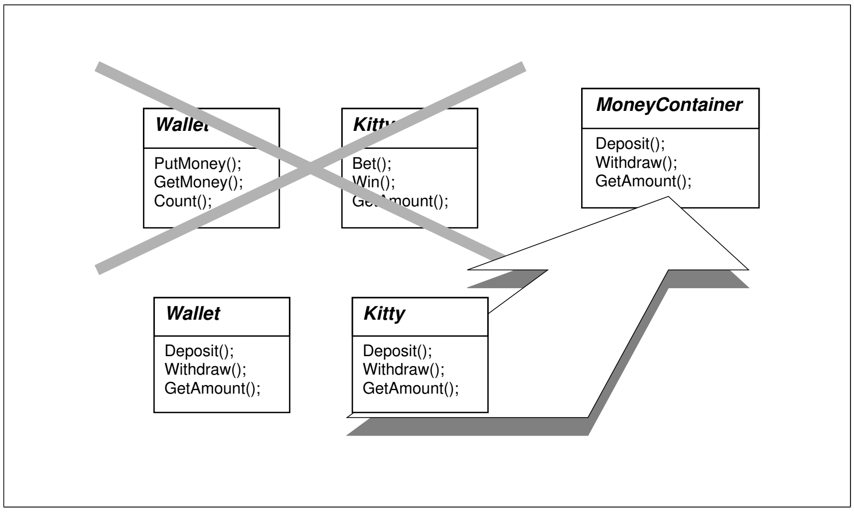


Figure 3.17 Generally stated responsibilities promote abstraction identification

Responsibilities should be placed where they logically belong, but in some cases it may not be clear to which class a responsibility logically belongs. The designer should then aim to distribute the intelligence to achieve the highest level of abstraction. If a responsibility can belong in several classes from a logical point of view, the responsibility should be placed in the class where it allows the designer to identify the largest abstraction.

- **Guideline 25:** The first concern when distributing the responsibilities should be to create methods which perform logical operations on instances of the class.

- **Guideline 26:** Distribute system intelligence so that abstractions can be identified. When in doubt, the responsibility should be placed where it allows for the most abstractions.

Identify abstractions, i.e. extract common behaviour into abstract superclasses. Defining as many abstract classes as possible implies factoring out as much common behaviour as possible [Karl95, p. 310].

Moving common responsibilities as high up in the inheritance hierarchy as possible helps finding the most suitable abstractions [Wirfs90].

- **Guideline 27:** Create as many abstract classes as possible. Look for duplicated responsibilities and factor them into abstract superclasses.

- **Guideline 28:** Factor common responsibilities as high in the inheritance hierarchy as possible.

3.5.3.3 Analyse Collaborations

An object collaborates with an other object if it has to invoke one or more of the other objects methods to fulfil its responsibilities [Karl95, p.306]. During this activity, the collaborations between the objects in the system should be identified. For each object and each responsibility, it should be found out if the responsibility can be fulfilled by the object itself and, if not, which objects it has to collaborate with.

Interaction diagrams, as shown in Figure 3.18, are helpful tools in defining how the objects should collaborate in the system [Jaco92, p. 142].

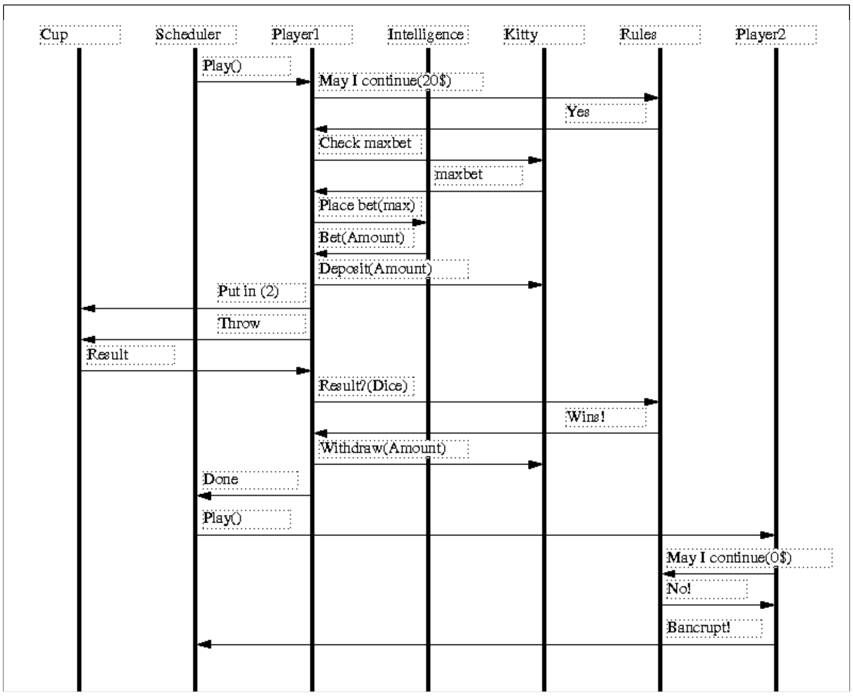


Figure 3.18 Interaction diagrams are an aid in analysing the collaborations

The associations between the objects may have to be changed from analysis. This is probably the most common change to the analysis model. The actual implementation of associations and synchronization between processes are examples where the associations may be changed. [Jaco92, p. 206]

Since the intention is to make the design extensible, no references to concrete classes should be made. Make sure to define collaborations between abstract classes, as in Figure 3.19. Though the collaborations become somewhat abstract, this paves the way for using dynamically bound methods in the concrete classes.

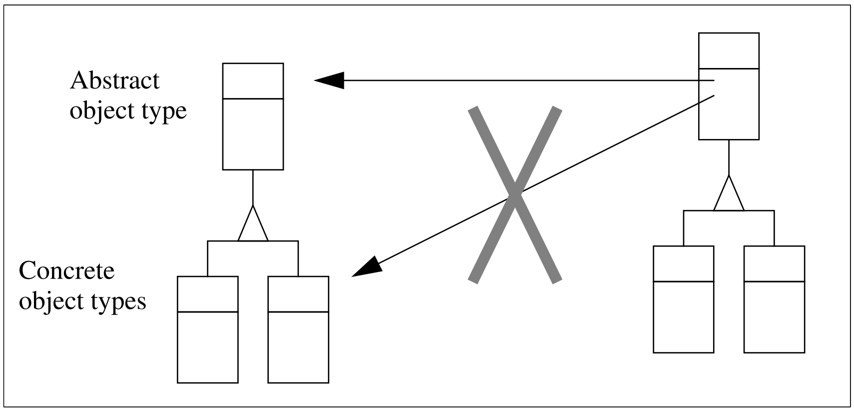


Figure 3.19 If the concrete leaves of the framework are referenced, they are no longer easily interchangeable.

- **Guideline 29:** Define collaborations between abstract classes. Use polymorphism to access the methods in the concrete leaves of the framework.

3.5.3.4 Refine the Inheritance Hierarchies and Collaborations

This activity is going on continuously throughout the process. Since all abstractions are not likely to be identified at once, the designers probably will have to iterate through the previous activities. The guidelines provided here are intended to promote the identification of abstractions during the process.

Following the guidelines in the previously described activities, and actively seeking to identify abstractions, will provide deep and narrow inheritance hierarchies, since the behaviour shared by classes will have been abstracted into superclasses. Wide and shallow inheritance hierarchies indicate that abstractions still are to be found in the hierarchy.

- **Guideline 30:** Class hierarchies should be fairly deep and narrow. Shallow and wide inheritance hierarchies indicate that abstractions still are to be found in the hierarchy.

A major concern when refining the hierarchies and collaborations should be to preserve the general and abstract functionality identified in analysis. Further refinement should not violate the conceptual abstractions.

- **Guideline 31:** Preserve the abstractions identified in domain analysis and analysis. Further refinement should not violate the conceptual abstractions.

One way to start the refinement is to look for subclasses that implement the same method and try to migrate the method to a new common superclass [Joh88]. This approach might result in a deep inheritance hierarchy that might be hard to comprehend, since its methods will be spread in the hierarchy. Whenever possible, the inheritance should be replaced by composition. Try not to extend the inheritance hierarchy too far, but to extract behaviour into a new class hierarchy and use instances of the new class hierarchy as components in instances of the first class hierarchy.

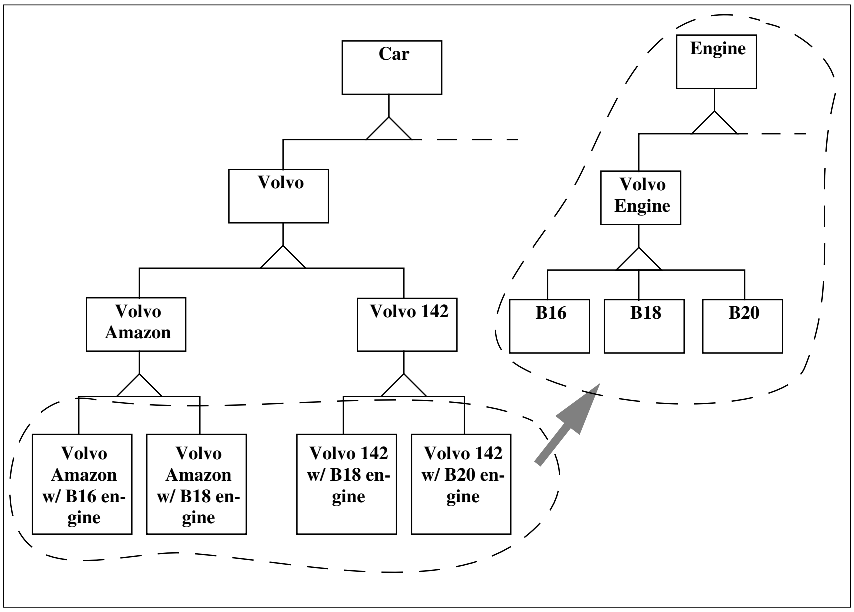


Figure 3.20 A transfer from inheritance to composition. An engine is used as a component in an instance of a car.

- **Guideline 32:** Try not to extend the inheritance hierarchies too far. Class hierarchies with more than 5 levels of abstraction should be considered candidates for restructuring. Use composition to flatten the hierarchies.

Designers should look for classes or methods that have different names, but provide the same functionality. Renaming these is conceptually simple, and will make it easier to see commonalities, but requires quite a lot of text editor work. [Joh95].

- **Guideline 33:** Make sure things that are the same are named the same.

If there are methods or classes that provide approximately the same functionality, the possibility of parameterizing shall be investigated. If the differences can be eliminated by passing parameters, similar classes in different applications can be replaced by one general class in the framework. The class is used with different parameters passed, depending on the application using the class.

- **Guideline 34:** Eliminate differences by parameterizing. If some classes or methods provide approximately the same behaviour, the possibility of parameterizing should be investigated.

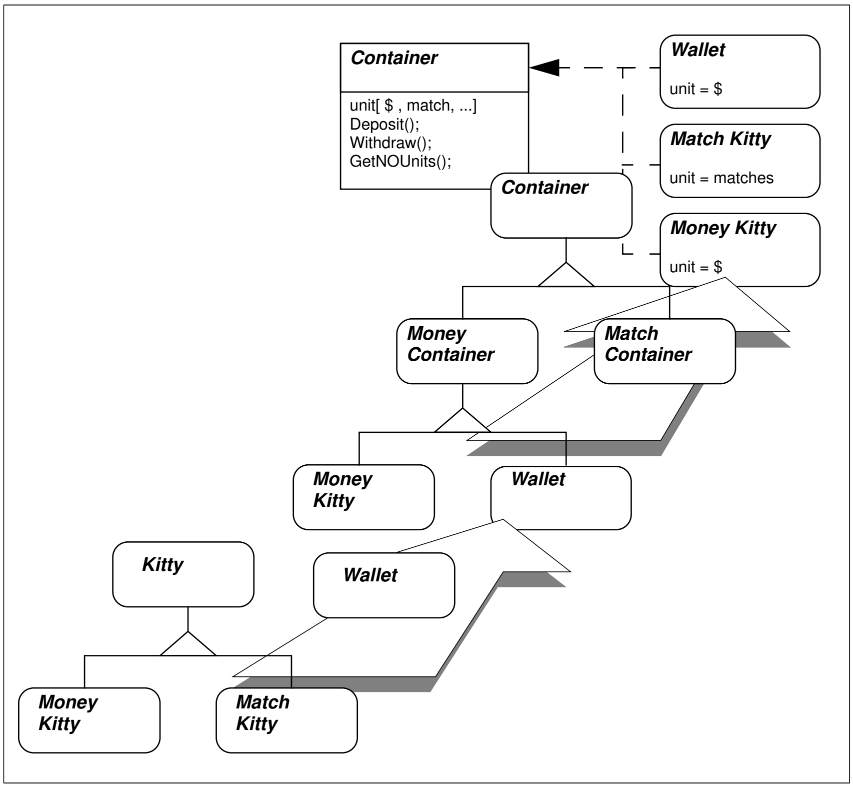


Figure 3.21 General properties is identified by renaming and parameterizing.

Johnsson says that iteration seems inevitable, as all abstractions are not likely to be found in the first try [Joh95]. This implies that the class hierarchies will be restructured during the design, and a prerequisite for restructuring the hierarchies is to understand them. Understandability can be achieved either by proper documentation or by simplicity, or by a combination of both.

• **Guideline 35:** Maintain the documentation and models, to ease the understanding of the class hierarchies.

Multiple inheritance is a feature in some object-oriented languages, e.g. C++, that makes it possible for a subclass to inherit from more than one superclass. Taligent recommends that a distinction is made between base classes, that represent logical objects, and mixin classes, that represent optional functionality [Tal94c]. A class may inherit from zero or one base classes, plus zero or more mixin classes, and a class that inherits from a base class is itself a base class. Mixin classes only inherit from other mixin classes. This approach provides a conventional inheritance hierarchy of base classes, with add-in mixin classes for optional functionality [Tal94c; Gam94].

If a relationship is realized through multiple inheritance that violates these guidelines, the motivation should be thoroughly documented. Multiple inheritance and especially ambiguous multiple inheritance makes the inheritance hierarchy hard to understand.

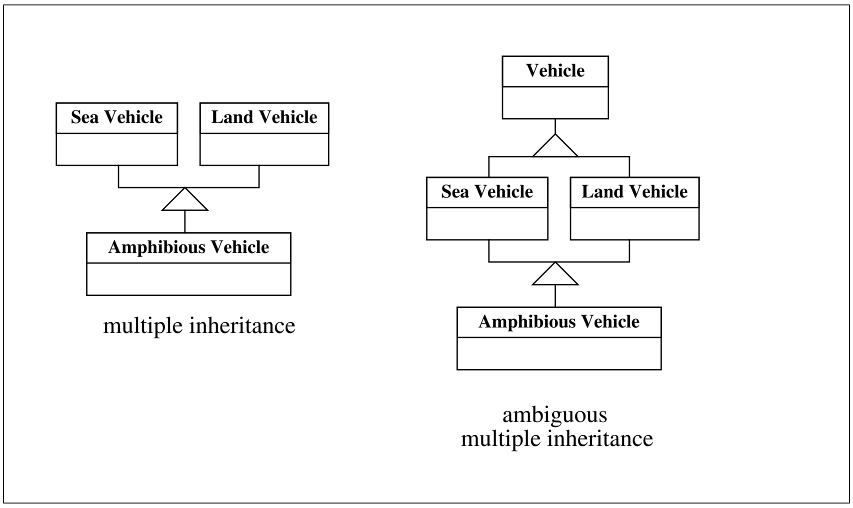


Figure 3.22 Multiple inheritance complicates the inheritance structures. Especially ambiguous multiple inheritance should be handled with care.

• **Guideline 36:** Multiple inheritance should be handled with care. Multiple inheritance complicates the inheritance structure and might make the framework design hard to understand.

Future classes derived from the framework should be able to use any data representation without fear of conflicting with the one inherited. In a concrete superclass it is easy to make to restrictive assumptions about specializations. Subclassing a concrete class indicates a faulty design and should be avoided.

- **Guideline 37:** Only the leaves of an inheritance hierarchy in a framework should be concrete. Restructure the hierarchy instead of inheriting from a concrete class.

No new methods should be introduced in the concrete leaves of the inheritance hierarchy, as these methods cannot be called through the abstract superclass' interface. Cancelling inherited methods imply that the superclass' interface is not valid for the subclass. The concept of polymorphism is fundamental to framework design and shall not be violated.

- **Guideline 38:** Use type preserving inheritance when the concrete leaves of the framework are derived from its superclasses. Both adding and cancelling inherited methods will violate the polymorphism.

3.5.4 Detailed Design

During the detailed design phase, all classes with attributes and methods are identified and described using the target implementation language [Karl95, p.283]. The inputs are the objects and the collaborations identified in architectural design, as represented in the static object model and the dynamic models, e.g. interaction diagrams and state transition graphs [Jaco92, p. 215].

A method with few parameters is more likely to be common to more than one class than a method with a lot of parameters. A method common to more than one class may be abstracted into a common superclass. Methods with many parameters should be redefined and possibly divided into several methods. An exceptions to this are object constructors. [Jaco92, p. 215; Joh88]

- **Guideline 39:** Methods should have few parameters. Methods with more than five parameters should be considered candidates for restructuring.

A method should perform only one task. A method which performs many different tasks should be divided into several methods, since parts of the method may be shared by several classes while other parts are unique to one class.

- **Guideline 40:** Let one method perform only one task. Parts of a methods performing several tasks might be common to several classes.

Classes are abstractions of the objects instantiated from it. Classes with many methods (more than 25) represent complicated abstractions, and probably consist of several different abstractions. These abstractions should have their own classes. Complicated public interfaces are also hard to understand.

- **Guideline 41:** Keep a small public interface for a class. Classes with more than 25 methods should be considered candidates for restructuring.

New abstractions may be found during detailed design. The abstractions should be introduced in the models where they belong, to maintain the structure of the documentation. Conceptual abstractions belong in analysis.

- **Guideline 42:** If new abstractions are identified, introduce them in the appropriate model. Conceptual abstractions in the analysis models, and lower-level abstractions in the design model.

To allow the identification of further abstractions, method signatures should be consistent, uniformity should be favoured over specificity.

- **Guideline 43:** Keep method signatures consistent. Things that are the same should be named the same.

3.5.5 Summary

The objective of the design phase is to provide a design of an implementation that easily can be adapted to provide the specific functionality of the applications in the domain.

The design shall provide a firm base for the implementation of the framework, and preserve the abstractions from the earlier phases.

Further abstractions should always be sought, but the hierarchies should not be extended too far. Composition should replace inheritance whenever possible.

“Obvious” abstractions, and well-known abstractions are quite easy to find, but steps should be taken to alleviate the finding of new abstractions. Such steps are having a strategy for expressing responsibilities in a general way, or strict naming conventions. Design patterns might be targets when identifying abstractions.

It is of importance to keep up a good framework structure, e.g. with only the leaf classes of the inheritance hierarchy concrete, since the structure will change continuously as new abstractions are found. The decisions and abstraction should also be possible to trace back to the origin in the analysis models.

Communications and the decision-making during design can be alleviated by using design patterns, since they provide a level of abstraction above objects and classes, and represent proven design solutions to common problems in framework design.

The detailed design shall provide uniform classes with methods that are as probable as possible to be common to several classes. Means to achieve this may be keeping the number of arguments small.

3.6 Implementation

“A framework is a generalisation of the implementation of several applications.”

Ralph E Johnson [John95]

This section covers the Implementation phase of framework development.

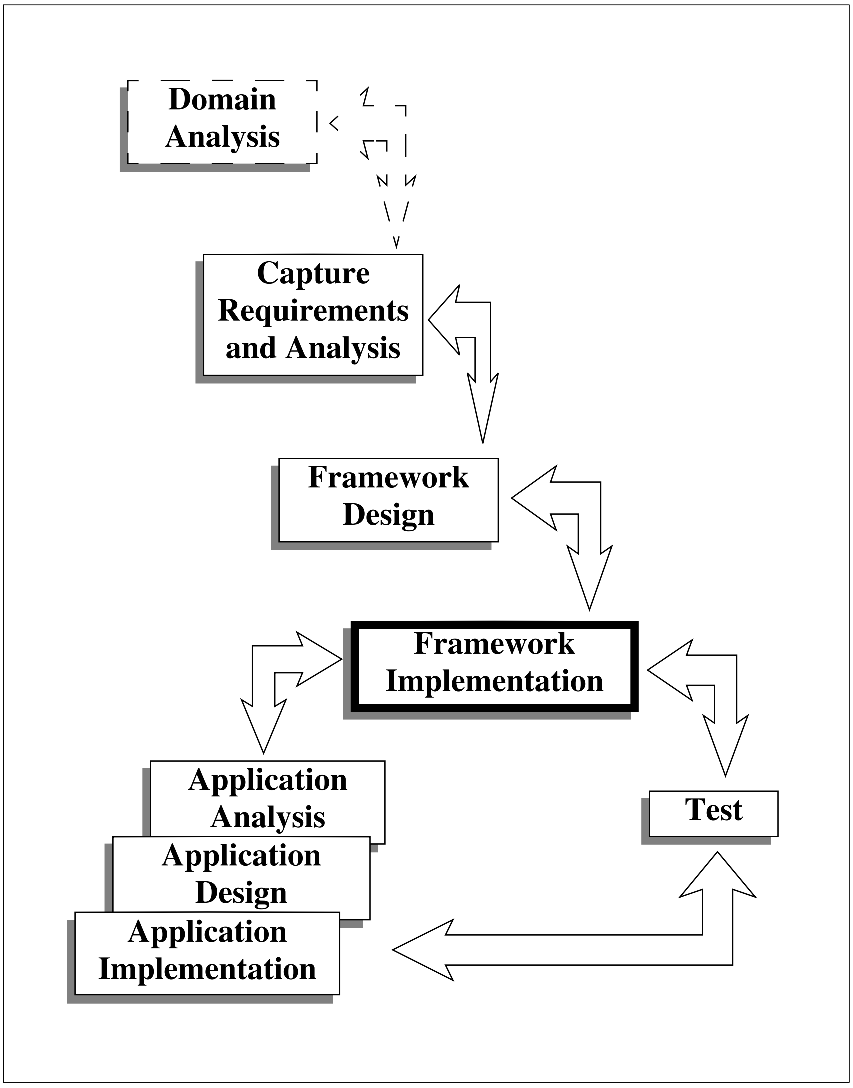


Figure 3.23 Process roadmap

The section is intended to provide some guidelines to follow when implementing an object-oriented framework. The guidelines are compiled from the REBOOT project, as documented in [Karl95, p. 315-334], and revised for the implementation of framework designs. Some of these guidelines are C++ specific and some are general for all object-oriented languages, and the reader is supposed to have a fair knowledge of C++ and its object-oriented constructs.

Implementation Strategy

In the implementation of a framework, a top-down approach should be the most suitable, with development of the high-level objects first. These implement the general functionality of the applications, and subcontract to low-level objects. [Karl95, p. 285]

All low-level objects are not available at the time of the testing of the framework, so a means of replacing them has to be found. Either code stubs can be provided, or the calls to the low-level objects can be simulated. [Karl95, p. 285; Som92, p. 381]

The top-down approach favours prototyping, since the main functionality is defined at an early stage [Karl95, p. 285].

Implementation Standards

The implementation standard conventions should either be defined or reused. These conventions include the definition of file structures, naming conventions and rules for references and inline functions and so on [Karl95, p. 285].

By having a uniform source code, the reading of the code is facilitated [Jaco92, p. 239]. To have a code that is easy to read will facilitate the understanding of the framework, and shorten the retention time for the user.

Many companies have defined their own standards, and some are widely spread, e.g. the Taligent and Ellemtel style guides for C++ programming [Tal94c, Henr92].

3.6.1 Process

The implementation phase follows the detailed design phase, where all classes with attributes and methods are identified and described using the target implementation language [Karl95, p. 283]. The objective of the implementation phase is to implement the objects, the relationships and the collaborations identified in the design phase.

There is no strict boundary between detailed design, implementation and testing, since inconsistencies discovered during implementation require a return to detailed design. Components are also often tested during implementation.[Karl95, p. 283]

The input is a detailed description of the classes, their interfaces and external definitions specified with the formalism of the implementation language. The output is a set of implemented classes, ready to be tested [Karl95, p. 283].

For each class there are two steps:

- Implementation of the class' external interface. The interface, defined during detailed design, is completed to include the internal definition of the class, i.e. protected and private attributes and methods.

- Implementation of the methods, starting with an empty method body with correct return type. The internal behaviour is identified by examining the dynamic models, i.e. the interaction diagrams and the state transition graphs. The interaction diagrams may also contain pseudocode, on which the implementation can be based. The methods' entire behaviour is implemented in this step.

These steps are usually followed by unit testing. [Karl95, p. 286]

3.6.2 Guidelines

The purpose of these guidelines is to help preserve the benefits from making a good framework design, to make the code easy to understand and to ease the work of the framework user.

3.6.2.1 Relationships

The relationships between classes identified in the design should be preserved or transformed in a standardized manner in a framework implementation. Some transformations and concepts that might be encountered in a C++ implementation are described in Table 3.1.

Object oriented concept	C++

B “is-a” A, inheritance	Public inheritance.
	`class A: public B {`
	`  :`
	`}`

A “has-a” or “consists of” B, aggregation	Declare the contained objects as private or protected attributes.
	`class A {`
	`protected:`
	`  B myB;`
	`    :`
	`}`

A “knows” or “uses-a” B, association	Take a reference, or a pointer, to another class as a parameter.
	`class A {`
	`  void Operation(B* aB);`
	`}`

Polymorphism	Declare methods as `virtual` in the base classes, while their implementation is declared in the subclasses.
	`class A {`
	`  virtual void Operation();`
	`}`
	`class B: public A {`
	`  void Operation();`
	`}`
	`class C: public A {`
	`  void Operation();`
	`}`
	`B::Operation() {`
	`  //implementation B`
	`}`
	`C::Operation() {`
	`  //implementation C`
	`}`

Table 3.1 Object-oriented concepts in C++

Object oriented concept	C++

Abstract class	By declaring the constructor as `protected` and/or by having at least one pure `virtual` method, the class cannot be instantiated
	`class A {`
	`protected:`
	` A();`
	`public:`
	` virtual void Operation() = 0;`
	`  // pure virtual`
	`}`

Encapsulation, information hiding	Specify attributes as `private`

Table 3.1 Object-oriented concepts in C++

Multiple inheritance will complicate the inheritance structure. This is particularly true for ambiguous inheritance structures. Complicated inheritance structures are hard to understand without proper documentation. The ideas behind the framework design has to be as evident as possible, in the code as well.

- **Guideline 44:** Comment all multiple inheritance thoroughly. Thorough documentation might make up for the complications multiple inheritance implies.

Explicitly calling a method lower in a class hierarchy is called casting down, and introduces dependencies beyond the inheritance structure. A method lower in the class hierarchy should be implicitly called through the abstract superclass' interface, and the run-time system will bind the call to the appropriate implementation dynamically.

- **Guideline 45:** Avoid casting down the inheritance hierarchy. The methods in a subclass should be accessed through the superclass' interface.

The `friend` relation in C++ introduces relations in a component that are difficult to follow and handle by the framework user. It allows other classes to get access to a class' private parts and thereby violates encapsulation. If possible, it is better to make some member functions `friends` than to make a whole class a `friend`.

- **Guideline 46:** Avoid using `friend` if possible, as the `friend` concept violates information hiding. It is better to make some member functions `friends` than to make a whole class a `friend`.

A subclass should not decide which methods to inherit, as this contradicts the specialization relationships in the inheritance structures. Restructure the inheritance hierarchies instead of using type-restrictive inheritance, i.e. cancelling one or more of the inherited methods, as the interface of a subclass using type-restrictive inheritance will be unclear.

- **Guideline 47:** Restructure the inheritance hierarchies instead of using type-restrictive inheritance.

```
class List{
public:
    :
  int Count();
    :
};
class Set: private List{
public:
  List::Count(); //Makes the count method visible
    :
}
```

Figure 3.24 Inheritance with cancellation, realized by private inheritance in C++.

Type-restrictive inheritance is realized by private inheritance in C++, see Figure 3.24. Private inheritance is not an object-oriented concept and it is not used in any object-oriented design methodology. Private inheritance may confuse the framework user, since it is difficult to examine what is inherited and therefore what is reused.

- **Guideline 48:** Do not use private inheritance. Private inheritance is not an object-oriented concept.

3.6.2.2 Classes and Methods

The the guidelines from the design phase should still apply, and it is important that the structures from detailed design is transferred into code with care.

Abstract classes are not intended to be instantiated, and shall be prevented from being instantiated. Classes can not be declared as abstract in C++, but the class can not be instantiated if at least one method is declared as pure `virtual`, or the constructor is specified as protected. See Table 3.1.

- **Guideline 49:** Inhibit abstract classes from being instantiated

The methods of a framework class that are intended to be redefined in the subclasses should be declared as `virtual`. Defining methods as `virtual` provides hooks for changing or extending the behaviour of the framework. There is no mechanism in C++ to make it difficult or impossible to overload a method, and if the decision is made to redefine a non-`virtual` method, the purpose of the framework is violated and either the problem should be solved in another way or the framework should be redesigned.

- **Guideline 50:** All methods intended to be overloaded or redefined in subclasses must be declared as `virtual`.

A method with more than twenty lines of code is a potential candidate for modification, as methods should be quite small. Small methods are easier to understand and modify, and the behaviour can be changed by redefining a few small methods, instead of modifying one large method. A small method is also more likely to be common to several classes, and can then be migrated into a common superclass.

- **Guideline 51:** Keep methods small, methods with more than 20 lines should be regarded candidates for modification.

By declaring a member method as `const`, it is impossible to change its attributes in the implementation of the class; that is, member methods declared as `const` do not change the state of a class, see Figure 3.25. Declaring member methods as `const` ensures a framework designer that the called method will not change the class' internal attributes. There is therefore no risk of changing the state of an object by calling these methods. This guideline forces the framework user to implement the redefined methods as the framework developer intended.

- **Guideline 52:** Declare member methods `const` when possible. Declaring a method `const` ensures that invoking it will not affect the state of the object.

```
class ABCD{
public:
  Set(const int newInt);
    // newInt can not be changed inside the method
  int Get() const;
    // this method will not change the state of the
       class
}
```

Figure 3.25 Declaring parameters and methods const may help force the intended use of the framework.

By declaring a parameter or a pointer to a parameter as `const`, it is impossible to change its value in the implementation of the method, see Figure 3.25. Declaring parameters as `const` ensures the framework designer that the parameters, or pointers to parameters, are not changed in the implementation of the method. There is therefore no risk in passing a parameter that it is not intended to change to such a method. This guideline forces the framework user to implement the redefined methods as the framework developer intended

- **Guideline 53:** Declare parameters `const` when possible. Declaring a parameter `const` ensures that its value will not be changed in the method.

Use polymorphism instead of explicitly checking the object type. When checking object types, unnecessary dependencies on other object types are created. When a new object types is introduced in the framework, it has to be modified by inserting another check. By using polymorphism instead, the framework can be extended by simply deriving a new subclass.

- **Guideline 54:** Eliminate explicit type checking on object types

Specify attributes as private and provide access to attributes only through public methods for external objects, and through protected methods for subclasses. This hides data representation and keeps the interface stable even if the type of the attribute is changed. The framework developer has the responsibility of making a suitable interface for the framework user. By using this interface, the framework classes are derived correctly.

- **Guideline 55:** Specify attributes as `private`. Specifying attributes as private hides the data representation and makes the class' interface stable.

Implementation and code in header files violates encapsulation, use explicit `inline` instead. See Figure 3.26.

```
class Kitty: public Money_Container {public:
  void Deposit(int const nAmount){
    nSum = nSum+nAmount;
  }
    // Implicit inline
}

// File: file.h (header file)
class Kitty: public Money_Container {
public:
  void Deposit(int const nAmount);
}

// File: file.cc (source file):
inline void Kitty::Deposit(int const nAmount){
  nSum = nSum+nAmount;
}
  //Explicit inline
```

Figure 3.26 Use explicit inline to avoid code in header files.

A framework developer should not lay any restrictions upon the implementation of a class derived from a framework class. The framework user should be free to use the implementation and data representation of his/her choice.

- **Guideline 56:** Avoid implicit `inline`, as implementation in header files violates encapsulation. Use explicit `inline` instead.

3.6.2.3 Constructors and Destructors

To keep the control and to avoid unexpected behaviour when an object is created, copied and destroyed, the framework developer should implement implicitly-generated methods, i.e. *constructor*, *destructor*, *copy constructor* and *assignment*, himself, and thereby prohibit the compiler from implementing them. This is especially important for classes with dynamically allocated memory and classes that have *uses-a* relations with other classes.

- **Guideline 57:** Implement implicitly generated class methods. The compiler's implementations may result in unexpected behaviour.

To inhibit the framework user from performing unintentional operations with the class, hide the copy constructor and the assignment operator in the private part of the class specification, when copying or assignment makes no sense.

- **Guideline 58:** When copying or assignment makes no sense, hide the copy constructor and the assignment operator in the private part of the class specification.

To stress correct deallocation of memory resources, the destructors should always be made `virtual` in the base classes. If the destructors are not `virtual`, only the destructor for the declared object is called. The classes in the framework should deallocate their own memory resources, and leave only the deallocation of the derived classes memory resources to the framework user. By declaring the destructors `virtual` in the base classes the framework users work will be decreased.

- **Guideline 59:** Always make destructors `virtual` in the base classes. Classes should deallocate the memory resources they use themselves.

3.6.3 Summary

The main concerns in the implementation phase are to provide code that is easy to understand and to preserve the benefits from a good framework analysis and design.

A well defined implementation standard makes the framework code easy to read and understand. Well defined standards have evolved in several companies, and some are available for use outside the company of origin, e.g. the Ellemtel style guides for C++ programming [Henr92].

To maintain a good object-oriented design, no non-object-oriented special features of the implementation language should be used, unless they are very well motivated. E.g. the `friend` concept in C++ should be avoided, but `friend` is used in some well-proven design patterns, where it is thoroughly motivated and documented [Gam94].

A top-down implementation strategy is suitable in framework development, since the over all behaviour is defined first. A top-down approach also favours prototyping.

3.7 Verification and Validation

This section covers the Verification and Validation phase of framework development. The subject Verification and Validation is very wide and this section is intended to provide a brief overview of the topic.

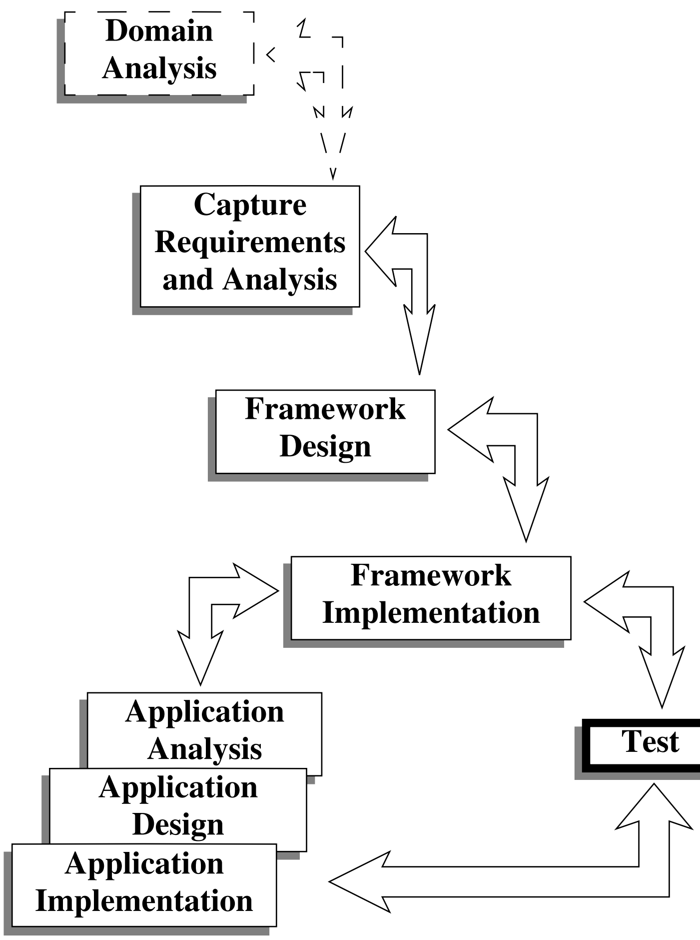


Figure 3.27 Process roadmap

The section starts with presenting the concepts of verification and validation followed by sections 3.8.1 to 3.8.4 which presents the main concepts of the test activities. Section 3.8.5 present the more important aspects when testing frameworks and the applications developed with frameworks. The chapter is closed with a short summary.

The verification aims to verify that the system being under construction will fulfil the requirements stated in the Domain Analysis or in the Analysis phase [Jaco92, p. 307]. The validation aims to check if the product under construction is the product the procurer really wants.

Two questions which are very common in the literature summarize the verification and validation activities [Jaco92, p. 307; Karl95, p. 335; Som92, p. 374]:

- Validation - Are we building the correct product?

- Verification - Are we building the product correctly?

The verification and validation activities take up a substantial part (30% - 50%) of the development costs [Jaco92, p. 308; Marc94, p. 1331]. Decreasing the need for test will have quite an economical impact on the development process.

One activity of the verification and validation phase is the test activity, This activity is often categorized into unit testing, integration testing and regression testing [Jaco92, p. 310; Karl95, p. 339; Marc94, p. 1334; Som92, p. 376]. Other activities in the verification and validation phase are code inspections and reviews and statistical testing [Jaco92, p. 313; Marc94, p. 1334].

This chapter will only briefly present statistical testing. Code inspections and reviews are not different for framework development and reuse than for other software development and will not be further presented in this chapter.

3.7.1 Unit Testing

When performing a unit test, only one unit is tested at a time. A unit may be an operation, a class, or a module consisting of several classes, maybe a framework. The idea is that a well defined unit with well defined responsibilities is tested so it is verified that the unit will fulfil the requirements imposed on the unit.

According to Marciniak, Jacobsson and Karlsson there are two methods for making unit tests [Marc94; Jaco92, p. 316; Karl95, p. 340]:

- S*tructural testing,* which requires knowledge of the unit's internal structure, provides knowledge of the test's code and branch coverage, which indicates the unit's reliability.

- *Specification testing,* or the functional testing, which only is based on the requirements imposed on the unit. No attention is paid to the internal structure of the module, it is only of interest how the unit responds to certain inputs.

3.7.2 Integration Testing

Testing how software units work together is called integration testing [Jaco92; Karl95, p. 340]. An integration test may, at the same time, be a unit test, a unit may consist of several sub units. In Ivar Jacobson's book *Object Oriented Software Engineering,* use cases are suggested as good tools for performing integration tests [Jaco92, p.310].

The final integration test, when all modules are combined to form the final system, is sometimes called the *system test*. The system test verifies if the system satisfies the requirements or not. If the requirements are formulated as use cases, the results of a certain use case will directly map onto the satisfying of a specific requirement. The system test will be the first possibility to test if the system will satisfy the requirements imposed on the system. The earlier integration test will only satisfy requirements which are derived from the procurer's requirements, thus the importance of the system test.

3.7.3 Test Model

It is suitable to use a test model both in the development of the framework and during the reuse of the framework. The test model consists of test cases and test beds [Karl95, p. 337].

A test bed [Jaco92, p. 316] is used to simulate the tested unit's surrounding world. The test performs calls to the operations of the tested object.

A test case consists of an action sequence or a sequence of stimulus, together with a specification of the desired event. If use cases are used in requirements model they will form a good base from which test cases are easy to derive. The requirements which are formulated as use cases are directly verifiable by the execution of the test cases which are derived from use cases.

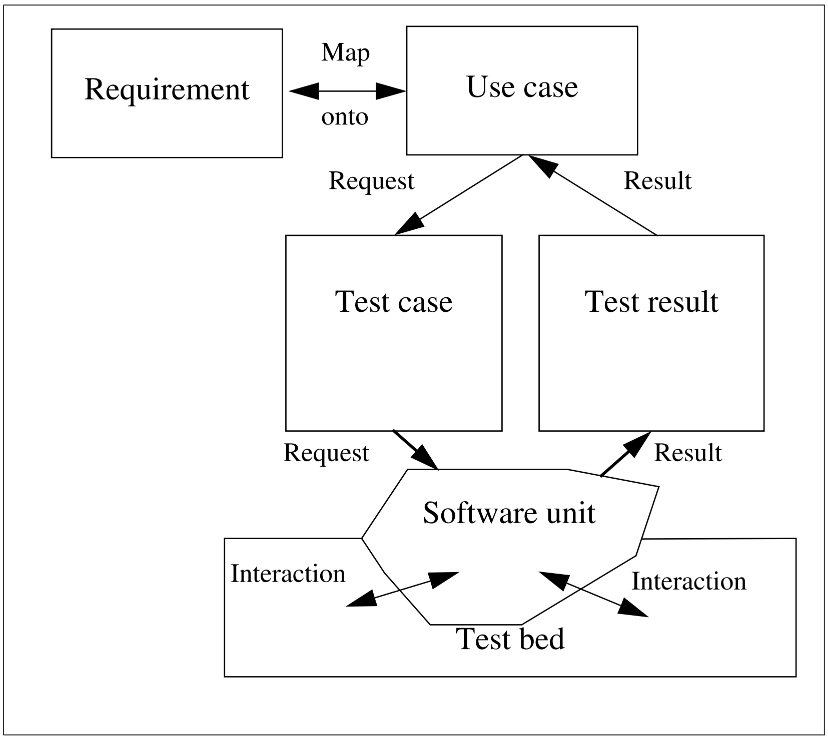


Figure 3.28 The test relations.

3.7.4 Frameworks Specifics

There is no substantial difference between testing units of a framework compared to testing ordinary software systems. However there are some differences depending on the use of object oriented techniques and the fact that frameworks are developed to be reused.

There are mainly two questions that should be answered by the testing of a framework. First, if the framework really covers the domain as intended [Joh91, p. 33]. This is verified by continuous reuse of the framework. If the framework is unsuitable for developing an application within the framework's domain, it should be considered if the framework should be redesigned or, perhaps, if the scope of the domain should be modified. Existing applications may be used to model scenarios of events that can be used in a later stage, to test the reusability of the framework [Karl95, p. 302].

Second, the framework must fulfil the parts of the application's requirements which is within the framework's area of responsibility. This is verified primarily within the activity of system testing.

Jacobson [Jaco92, p. 319] claims that polymorphism is a good tool in the testing process. If changes are made in a subclass - the server class - there will be no need to test the client class (the framework). It will be sufficient to test the subclass, verifying that the subclass will respond appropriate to the calls from the framework.

The statement above implies that if new concrete classes are designed to configure the framework to a specific application. then each of these new classes have to be unit tested, but the framework itself do not have to be tested as a single component again. However, it has to be tested in an integration test, a test of the complete application, how these new concrete classes will interact with the framework.

If all of the concrete classes are taken from the framework's library (the classes have been used before) two cases may occur. The same combination of these classes and the framework have been used and tested before, and it will not be necessary to test this combination again. The second case occurs when the concrete classes have been used before together with the framework but not in this particular combination. The second case requires no new separate tests of the concrete classes, but a test of the complete application is necessary.

The fact that it is not necessary to unit test a reused framework should save a great deal of testing effort. A mature framework with all of the needed concrete classes in the framework's library will of course save even more testing effort.

Jacobsson [Jaco92, p. 320] also points out some difficulties with polymorphism compared to traditional procedure oriented languages. The difficulties with polymorphism occurs in a structural test when different paths should be covered. When using a CASE statement as in ordinary techniques the possible paths are shown but when using polymorphism only the invoked unit is displayed.

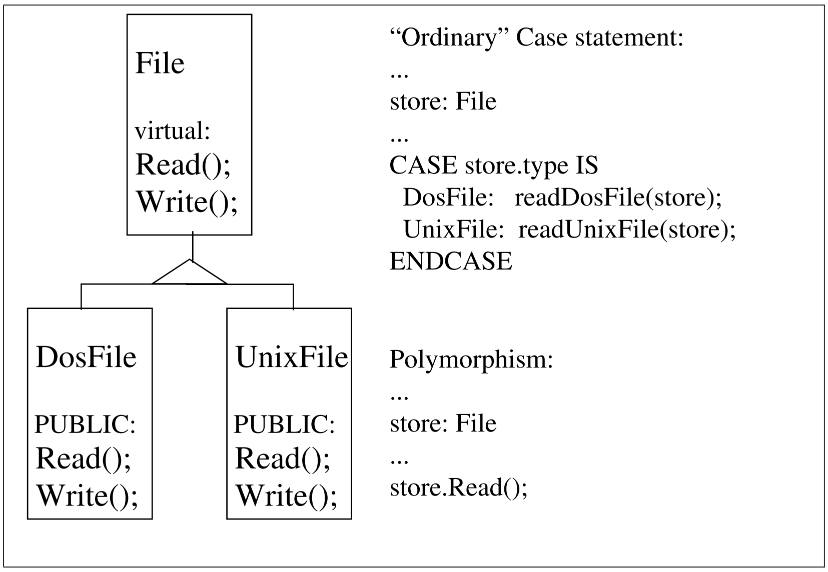


Figure 3.29 The differences between polymorphism and an “ordinary” case statement.

Inheritance will decrease the amount of code, but not the amount of testing. Methods inherited from a super class may work fine in the super class, but they must not necessarily work well in the inheriting class [Jaco92, p. 321]. Inherited methods should be tested as well as methods implemented in the class.

3.7.4.1 Reliability certification

Before reusing a component it is, in most cases, important that the reuser can make a judgement of the component's reliability. Reliability is often critical because a failure of the system will often cost the reuser a large sum of money [Som92, p. 403].

The requirements should specify the required reliability level of the system [Som92, p. 396].

If it is impossible or difficult for the reuser to gain confidence of the candidate component then the reuser probably will choose another component or develop his own.

There exist a number of reliability metrics possible to provide with the component. However, different users will use the component in different ways. The reliability metrics of the component are strongly dependent on how the component is used.

Sommerville [Som92, p. 394] presents some reliability metrics:

- The first metric is the *probability failure on demand* which is a measure of the probability that the system will behave in a unexpected way when it is called.

- The *rate of failure occurrence* is the second metric which is a measure of how many times the system will behave unexpected per time unit. According to Sommerville this metric is the most general one.

- The third metric is *mean time to failure*, which is a measure of the mean time between two failures occurs on the system.

Availability shows the probability that the system is available for use. According to these metrics the rate of error per lines of code is not as important as the actual rate of failure under use.

3.7.4.2 Statistical Testing

Statistical testing is a test method which provides information of the probability of error or the probability that the system is available for use. Sommerville [Som92, p. 398] identify four steps in the process of statistical testing:

1. Determine the pattern of how the unit is going to be used.

2. Collect and select the test data identifying test cases according to this pattern.

3. Execute the test cases according to the usage pattern. Recording the execution time until a failure occurs.

4. Compute the software reliability according to the test results.

A framework is reused with the purpose of developing new applications. Different applications may have very different usage patterns, which may limit the use of statistical testing. However, work which argues for usage testing as a method for reliability certification of software components has been done. Usage testing may also be a suitable method to certify frameworks as well.

3.7.5 Summary

Reducing the need for test will probably have a rather significant economical impact on the development costs of a software product.

The tests performed during the framework development process are not different than the tests in an ordinary object-oriented development process.

The validation of a framework is done when it is reused.

A reused framework does not have to be unit tested again the framework has already been tested once.

A mature framework with many reused concrete classes reduces the need for testing even more because reused concrete classes are not needed to be unit tested again.

However, when applications are developed using the framework are integration tests are always necessary.

3.8 Maintenance

3.8.1 Introduction

Every software system of a significant size that will be used during a long period of time needs maintenance. Object-oriented frameworks are not an exception to this rule. However, an object-oriented framework is a special type of a software system and will have some special characteristics compared to more common software architectures.

The first two sections discuss computer systems in general. The first section, 3.9.2 considers the evolution dynamics of software systems. 3.9.3 argues about how to limit the need for maintenance of software systems. These general sections are followed by two more framework specific. 3.9.4 discuss the aspects of evolution dynamics of object-oriented frameworks. The following section argues about how to limit the need for maintenance of object-oriented frameworks. The chapter is closed with a short summary.

3.8.2 Computer System Evolution Dynamics

This chapter intends to discuss the evolution dynamics of computer systems. All large systems are exposed to changes in the surrounding environment leading to modification of the requirements imposed on the systems. To meet with the changing requirements the system has to evolve and thus, the system requires adaptive maintenance. There is also a need for corrective maintenance because of undiscovered system errors and from coding, design and analysis errors. Most of the studies of program evolution dynamics has been performed by Lehman and Belady (1985) resulting in a set of five laws, the Lehman's laws, which in fact are hypotheses not laws:

1. The law of continuing change
2. The law of increasing complexity
3. The law of program evolution
4. The law of organizational stability
5. The law of conservation of familiarity

Sommerville clarifies these laws [Som92, p. 534-538].The first law, the law of continuing change states that a program that is being used in a real-world environment has to change with the changing environment or become less suitable for its purpose. A real-world environment will always change. For example a computer program for taxation will have to be modified if the government will change the taxation rules and a fire-alarm system will have to be modified if a new detector device is required by a new customer.

The law of increasing complexity states that a system exposed to changes will become more complex. The structure of a system that is being changed is degraded and the structure will become more unclear. Most often changes are done with a very local perspective and the structural consequences are not understood. To preserve the quality and understandability of the system's architecture extra resources must be provided, enabling restructuring activities.

The third law is the law of the evolution of large programs and it states that program evolution is a self-regulating process. System attributes as system size, the number of found errors and the time between releases is approximately the same between each release of the system. Lehman suggests that the dynamics of the system is established in an early stage of the development process. If a system is of low quality, inhabiting lots of errors, there will be a need of corrective maintenance. However, changes to the system will introduce new faults to the system, which then requires new actions to be taken and so on. Therefore the error rate will be fairly constant. The constant time between releases may be explained by the bureaucracies of the development organizations, which slows down the change process.

The law of organizational stability is the fourth law and it states that the rate of change over the systems life time is approximately constant and independent to the resources devoted to system development. This law is valid in most large programming projects. The law implies that large software development teams are unproductive because of the rapid increase of communication when the size of the development team grows. A large team needs to devote much effort to the communication.

The fifth law is the law of conservation of familiarity, which states that the incremental system change in each release is approximately constant.

3.8.3 Limiting the Need for Software Maintenance

Maintenance of software systems is hard, mainly because little attention is devoted to maintenance requirements during the development process. This results in a loss of traceability which makes it hard to trace back to user requirements or design specifications when making any modification or correction to the system [Marc94, p. 621].

Actions should be taken to ensure the traceability from the requirements through the processes of analysis, design and test down to the maintenance phase. Any modification made to a system shall be traced from a requirement, new or old. The test plans from the test phase should be the foundation from which the maintenance tests are done. It is necessary to identify if the modifications are of analysis, design or coding nature to be able to take the correct actions.

To enable traceability the right documents have to be produced. The analysis phase should produce documents which states the analysis results. The results of the design phase should be derived from the results of the analysis phase. The design results as well as the relationship between the results from the analysis and the design should be documented. Every phase should produce documents stating the results and it should always be possible to trace back the results to the earlier phase. For an example, to be sure of making the correct tests the test plans should be traced back to the requirements of the system.

The third law of the Lehman's laws, the law of program evolution, states as mentioned above that the rate of faults remains fairly constant. The dynamics of the system is established in an early stage of the development process and this implies that it is very important to do things correct the first time in software development. The earlier in the development process the fault is introduced the more damage it will do and the more expensive it will be to correct it.

The software should be developed with the knowledge of that the system will be maintained and necessary precautions should be taken to make maintenance of the system easier. This is done by following the design and coding guidelines of best practice.

3.8.4 Framework Evolution Dynamics

The aspects of software maintenance discussed above are also valid to object-oriented frameworks and some of these are even more important. This chapter will emphasize on these, for frameworks, especially important questions.

Lientz and Swanson [Lien80] found that maintenance made to improve the system but not changing the system's functionality took up a major part, 65%, of the maintenance effort, maintenance of corrective nature about 17% and adaptive maintenance about 18%.

A framework's internal architecture consists, as mentioned before, of a structure of abstract and concrete classes. A good framework consists of a fairly deep but narrow inheritance structure. However, inheritance violates the principle of information hiding. A change implemented in a super class will affect all of the depending subclasses.

A safe way to implement changes in a class hierarchy, in the short perspective, is to implement them in a subclass and inherit from the class which characteristics should be modified. The effects of the changes will then be controlled and the possibilities of unwanted behaviour depending of unknown dependencies is reduced.

Such modifications will, in the long run, lead to a degeneration of the class and inheritance structure. Therefore there is a need to restructure a framework regularly [Gam94, p. 353].

A framework is constructed with the intention that it will be used to develop many applications. The usefulness of a framework is reflected by how many times it is reused and all these applications are dependent on the framework. Modifications of interfaces and names are crucial and will most likely affect the dependent applications.

Much effort should be placed on the interface design to be able to produce stable interfaces. Naming should be done with great care. Good names are essential when designing for reuse and improves the understandability of the code and the functionality of the framework. Renaming will as well as changes to the interfaces affect the dependent applications and should as much as possible be done correctly the first time.

Yet modifications of names and interfaces will, probably, be necessary. When any change request is found valid it is better to implement these changes as fast as possible because more dependent applications may be developed meantime, increasing the number of dependent systems.

Internal modifications of the framework will not, in most cases, affect the interaction between the framework and the applications. Therefore these changes are not so crucial.

Each reuser should be notified when a component is modified, and especially if faults have been corrected and new versions issued [Karl95, p. 302]. Recording a reuse history is important, to enable modifications or validation of quality models and reusability models, as well as for validation of development guidelines.

Reusers should give feedback on components. Quality problems must be identified and dealt with both when developing new components and maintaining existing components [Karl95, p. 302].

3.8.5 Summary

Frameworks may reduce the need for maintenance. A framework is, as mentioned above, common to several applications and the framework is a big part of each application. Thus, the amount of code to maintain is reduced compared to the situation of maintaining a set of independent applications.

However, frameworks also introduces some difficulties. Applications developed from the framework will be dependent on the framework's interfaces. Modifications such as naming of the methods and parameters, the number of parameters in a method and possible changes in the methods services will have a great impact on the dependent applications.

4 FRAMEWORK REUSE

This chapter is intended to present an overview of the framework reuse process. The chapter is started with a discussion on how to organize the organization for reuse and the necessity of collecting reuse experiences in an organized way, section 4.1. Section 4.2 describes framework reuse and the necessity of understanding the framework before reusing it. Section 4.3 discusses how to introduce reuse in the analysis phase and section 4.4 present a brief overview on how to reuse architectural designs using frameworks. The chapter is concluded with a summary.

4.1 Reuse Organization

There are basically two ways to organize the staff when applying reuse of frameworks. One is to let the same people both develop the framework and reuse it. The other way to organize is to have separate development and reuse organizations.

If the intention is to sell the framework outside the association, the choice of reuse organization is limited. However, we believe that most associations will use their frameworks internally and the choice of reuse organization is heavily dependent on the company policies.

If the framework developers themselves will use the framework, they will know the problems and the limitations of the framework. Also, they will not have any problems of understanding the intentions behind the architecture and the solutions. The classical resistance to reusing other peoples solutions is also avoided and the well needed feedback from the users to the developers is easily achieved.

Development teams should, as much as possible, consist of engineers with knowledge of the domain, but this is seldom possible due to limited personnel and economical resources. By having separate teams of framework developers and application developers, the domain knowledge of the experienced engineers (the framework developers) may be reused by engineers with less domain knowledge (the application developers). This is the main argument for separate organizations.

Reporting

A framework will probably not be completely stable after the first application has been developed from it. The framework will probably need modification the first times it is reused and each time it is reused it will grow more and more stable [Joh91].

The reporting activity in the development with reuse process will contribute to the increasing maturity of the framework.

Depending on how the responsibilities for the reuse and development of the framework is organized the reports looks a little different, but in general a reuse report should include [Karl95, p. 355]:

- Information about the reuse environment.
- Information about difficulties in understanding the framework or the framework's library.
- Information about difficulties when adapting the framework.
- The costs of reusing the framework.
- If any adaptions were needed to modify the framework.

4.2 Overview

We have limited our thesis to cover object-oriented frameworks. However, there are other reusable components, mainly small components. Most literature covering the *development with reuse phase* only address these small components, and do not cover frameworks.

Reuse with frameworks requires a life cycle different from a life cycle supporting reuse with smaller components. For example, searching for suitable components requires a great effort in the case of reusing small components, but when reusing a framework the effort devoted to searching for frameworks is much less, due to the differences in size between components and frameworks.

Another difference between small components and frameworks is that understanding a framework requires greater effort than it takes to understand a smaller component, due to the framework's greater complexity.

Reuse will have greater impact if it is introduced early in the development phase. Reusing a framework, which is an architectural component, leads to reuse of all associated information, the products of the architectural design, code and test [Karl95, p.344]. If the framework covers a complete domain, parts of the domain analysis of this domain and the analysis is reused as well.

Understanding the Framework

The effort required to understand the framework is a serious limitation to reuse. Without understanding of what the framework does and how the framework works it can not be decided if the framework is a suitable solution to the problem. It is very difficult to understand the concepts of a software component by just reading its specifications [Marc94, p.603]. To gain full understanding of the object, people must see the object's static context and behaviour and have some knowledge of world surrounding the component.

In the REBOOT project, two aspects of understanding the component is considered [Karl95, p. 94]:

- Understanding of the functionality offered by the component. The focus should be on the component's interface and later, if needed, effort should be put into understanding the component's internal structure.
- Understanding of the non-functional aspects of the component such as efficiency, portability, reliability and understandability.

The results from the domain analysis are a good support for understanding a framework within the actual domain.

- **Guideline 60:** Use domain analysis results, if available, to understand the characteristics of the component [Karl95, p.348].

Any model of the framework is a good support when understanding it

If domain experts, experienced users of the framework or the developers of the framework are available, consult them when needed.

- **Guideline 61:** Consult experienced people and every useful model [Karl95, p.349].

Understanding of the components that are to be reused is important. Gaining understanding of these components can be costly. It is, however, essential because there are only limited ways to adapt a framework and it is necessary that the chosen framework is the appropriate one [Karl95, p. 349].

- **Guideline 62:** Carry out a complete study of the framework so it is fully understood and the appropriate choice can be made.

4.3 Analysis with Reuse

The analysis process aims to capture the requirements imposed on the system as well as modelling the application world.

Introducing reuse at the analysis stage has a great impact on the continuing development process [Karl95, p. 357]. A good tool to accomplish reuse of analysis is to reuse domain frameworks.

- **Guideline 63:** Identify reuse opportunities during the specification phase, and identify specific reuse requirements that support them [NATO91a; NATO91b; Karl95, p. 347].

It is important to formulate the requirements as generally as possible when performing analysis *with reuse,* because too detailed requirements will not map onto existing requirements. Different people will solve the same problem differently and no solution should be proposed in the requirements, since this will obscure the similarities between the requirements on existing components and requirements on the component to be developed [Karl95, p. 347]. To increase the suitability for reuse, only necessary functionality and performance should be described, since this will increase the freedom to choose among reusable solutions [Karl95, p. 359].

- **Guideline 64:** Do not over-specify requirements [NATO91a,b, Karl95 p.357].

Several advantages exist when reusing requirements [Karl95, p. 358]:

- Consistency among related systems is provided.
- De-facto standards are established.
- Proven implementations are used, which increases the reliability.
- The overall risk is reduced if the development process of a similar component may be studied.

The results from the domain analysis or the scope definition of the domain should be used to determine if the required component is comprised within a domain already covered by a reusable framework.

- **Guideline 65:** Use the products from the domain analysis to understand the context in which the application takes place, and its dynamics [Karl95, p. 359].

- **Guideline 66:** Use the domain analysis to determine and validate new application requirements[Karl95, p. 359].

If the required application is comprised within the domain and there exists a (almost) suitable framework its generic requirements can be used in the negotiations of the application's capabilities with the procurer [Karl95, p. 359]. The procurer may accept some reduction of the application's functionality if this reduction makes it possible to reuse a framework.

4.4 Design with Reuse

The design phase consists of two sub phases, architectural design and detailed design.

Architectural Design with Reuse

The architectural design aims to define a high-level strategy for solving the problem and implementing the solution.

Karlsson points out two main difficulties of reuse in the architectural phase [Karl95, p. 361]:

- The difficulty of building up a knowledge of predefined solutions.
- Applying that knowledge to structure the actual problem so it can be solved by the predefined solutions.

The problem is about knowing what solutions already exist and to identify solutions suitable to the problem. A framework may be such an existing solution. It may be necessary to adapt the framework to fully suit the problem, or it may be necessary to negotiate the capability of the required application as mentioned in section 4.3.

An adaption of the framework is most likely needed and to make the framework easier to understand and to adapt, the framework should be well documented. If there are several framework candidates, choose one that is well documented.

- **Guideline 67:** Select well-documented frameworks [Karl95, p. 362].

The selected framework should be evaluated and the framework's behaviour should be understood. A way to achieve this understanding is to carry out a some detailed design in advance [Karl95, p. 362]. This detailed design will generate knowledge about how to get access to the most central functionality of the framework.

When knowledge of how the framework interacts with other components and how the most central functionality is accomplished the remainder of the adaption is left to the phase of detailed design.

- **Guideline 68:** Carry out some detailed design of the application in advance for evaluation of the framework.

Implementation of new components that adapt the framework should follow the architectural strategy of the framework, since these components will be added to the framework's component library. To make it easy to understand the components in the framework's library, the components should be implemented and designed following the standards and strategies of the framework.

- **Guideline 69:** Preserve the strategy implemented in the framework when adapting it [Karl95, p. 362].

Reusing a framework is partly done by inheriting from a set of objects and classes already defined by the framework. When making these adaptions new objects and classes will be identified. Object-oriented design is an iterative process but the detailed objects and classes should be kept apart from the results from the architectural design phase. These results may be introduced in a preliminary version of the detailed design results.

- **Guideline 70:** Keep results from different phases separated

The best way to adapt a framework is refining it by using inheritance. Inheritance will let the framework's internals be kept unchanged.

- **Guideline 71:** Use inheritance to customize a framework [Karl95, p. 363].

4.5 Summary

Organizing the reuse process is important, if the framework is reused internally there are mainly two possibilities to organize the organization. Separate development and reuse organizations or not separated organizations. Separate organizations makes most use of the experienced engineers by letting them develop the framework and the less experienced engineers use the framework to develop the applications. The drawback is the difficulties to reuse other peoples work.

A framework will not be stable at once, feedback from the reusers are necessary to accomplish a stable framework. Therefore a process for collecting the reuse experiences should be established.

The life cycle of reuse of a framework is different from the life cycle of reuse of smaller components. Less search for components is needed when reusing a framework and a framework is introduced earlier in the life cycle

Understanding frameworks is harder than understanding a smaller component, but reuse introduction early in the life cycle will have a greater impact.

Requirements imposed on a system should be formulated as generally as possible if the system is going to be developed by reusing frameworks, or if the system itself is developed for reuse.

The scope of a domain framework is important information when it should be established if the framework is suitable for reuse in a special case or not.

5 SUMMARY AND CONCLUSIONS

This thesis provides a process for the development of object-oriented frameworks. Object-oriented frameworks are a reuse technique claimed to have high reuse potential.

Our conclusion is that frameworks do allow for a high degree of reuse, but are not always the best alternative from an economical point of view. Developing frameworks is only recommended if the application domain and its future evolution are well known.

The development of frameworks is time consuming, but a good framework will pay back when the development time of later applications is reduced. Little public evidence of economical benefits from framework development exists, but many companies use frameworks internally.

Current object-oriented methodologies do not support the identification of abstractions shared by several applications.

Framework development

Some of the techniques used in framework development may be suitable to ordinary application development as well. The framework development results in a stable architecture, less sensitive to changes in the requirements.

A thorough analysis of the domain should precede the framework development. The intention is to investigate if developing a framework in the domain is feasible, to identify concepts common to the applications and to define the borders of the domain. It is important to define the borders of the domain. A very general framework, that can be reused by a variety of applications, can not capture much of a specific application's functionality.

Frameworks should not become to big, it is better to divide a large framework into several small frameworks. Small frameworks are easier to reuse and maintain.

Analysis

The analysis team should have knowledge of the domain, i e knowledge of each application that is intended to be developed from the framework. The analysis team should also have knowledge of framework development.

A major issue during the capturing of requirements is to isolate all requirements common to the applications and to let these requirements be the requirements imposed on the framework. Use cases are suitable for framework analysis, since the use case concepts *abstract use cases* and *extends* support the isolation of general requirements.

Design

During the design of the framework, the focus should be on reusing and designing generic design solutions.

Communications and the decision-making during design can be alleviated by using design patterns, since they provide a level of abstraction above objects and classes, and represent proven design solutions to common problems in framework design.

Test, Maintenance and Reuse

A framework's architecture is validated when the framework is reused and the framework's implementation is tested when the applications are tested.

The amount of code to maintain is reduced when using frameworks, as large parts of the applications' code is implemented in the framework.

The scope of the framework is important information when establishing if the framework is suitable for reuse in a special case or not.

Reusing a framework might be hard, due to lack of adequate documentation. There are currently no well-documented techniques for framework documentation, but current research suggests design patterns as a component.

A GUIDELINES

This appendix contains a list of the guidelines provided for framework development and framework reuse. The section headings are included for reference.

3 Framework Development

3.4 Capture Requirements and Analysis Phase

- **Guideline 1:** A list of requirements on at least two applications should be provided together with a list of requirements on the framework.
- **Guideline 2:** A list of future requirements on the framework should be provided.

3.4.1 Capture Requirements

3.4.1.1 Requirements Process

- **Guideline 3:** Include members with knowledge of each application area and a member with knowledge of framework design into the analysis team.
- **Guideline 4:** Gather information from as many different sources as possible to acquire knowledge of which requirements are of importance.

3.4.1.2 Requirements Specification

- **Guideline 5:** Separate the requirements into framework specific and application specific requirements
- **Guideline 6:** The application and framework requirements should be divided into functional and non-functional requirements due to the different properties of the requirements.

3.4.1.3 Use Case Model

- **Guideline 7:** Separate the use cases into framework specific and application specific use cases. This enables to focus on what is general and what is specific between the given applications.

3.4.2 Analysis

3.4.2.1 Performing the Analysis

- **Guideline 8:** Remove redundant classes to refine the model from unimportant information.
- **Guideline 9:** Identify high level abstractions preparing for the identification of the framework.
- **Guideline 10:** Examine existing solutions to gain knowledge of possible frameworks.
- **Guideline 11:** Introduce only abstractions which are within the domain of the framework.
- **Guideline 12:** Structure large frameworks into sub frameworks. Small frameworks are in general more focused than large ones.

3.4.2.2 Static Object Model

- **Guideline 13:** Abstractions present in the domain model should be named the same in the static object model ensuring traceability.
- **Guideline 14:** Develop a static object model for each application.
- **Guideline 15:** Introduce abstractions common to several applications in the static object model of the framework.

3.4.3 Complementary Results and Models

- **Guideline 16:** Use graphical notations. Graphical notations make the models easier to understand.
- **Guideline 17:** Present the models clearly visible to all project members making the models easy to discuss.

3.5 The Design Phase

3.5.1 Object-Oriented Design

- **Guideline 18:** Subsystems shall have high cohesion and weak coupling.

3.5.2 The Framework Design Process

- **Guideline 19:** Study existing frameworks and generic designs, and try to reuse all available design knowledge.
- **Guideline 20:** Each design problem to which a design pattern apply shall be solved according to that pattern.
- **Guideline 21:** Approve the design solutions by prototyping. If necessary, go as far as to implementation to validate the design solutions.

3.5.3 Architectural Design

3.5.3.1 Refine the Analysis Object Model

- **Guideline 22:** Objects directly transferred from analysis should keep their names. To understand the framework from a conceptual point of view, the reuser should be able to trace the objects back to the analysis models.
- **Guideline 23:** Keep classes appropriately small. Classes with more than 25 methods should be considered candidates for restructuring.

3.5.3.2 Assign System Responsibilities to Specific Objects

- **Guideline 24:** State responsibilities as generally as possible. A common way to express responsibilities may help finding abstractions.
- **Guideline 25:** The first concern when distributing the responsibilities should be to create methods which perform logical operations on instances of the class.
- **Guideline 26:** Distribute system intelligence so that abstractions can be identified. When in doubt, the responsibility should be placed where it allows for the most abstractions.
- **Guideline 27:** Create as many abstract classes as possible. Look for duplicated responsibilities and factor them into abstract superclasses.
- **Guideline 28:** Factor common responsibilities as high in the inheritance hierarchy as possible.

3.5.3.3 Analyse Collaborations

- **Guideline 29:** Define collaborations between abstract classes. Use polymorphism to access the methods in the concrete leaves of the framework.

3.5.3.4 Refine the Inheritance Hierarchies and Collaborations

- **Guideline 30:** Class hierarchies should be fairly deep and narrow. Shallow and wide inheritance hierarchies indicate that abstractions still are to be found in the hierarchy.

- **Guideline 31:** Preserve the abstractions identified in domain analysis and analysis. Further refinement should not violate the conceptual abstractions.
- **Guideline 32:** Try not to extend the inheritance hierarchies too far. Class hierarchies with more than 5 levels of abstraction should be considered candidates for restructuring. Use composition to flatten the hierarchies.
- **Guideline 33:** Make sure things that are the same are named the same.
- **Guideline 34:** Eliminate differences by parameterizing. If some classes or methods provide approximately the same behaviour, the possibility of parameterizing should be investigated.
- **Guideline 35:** Maintain the documentation and models, to ease the understanding of the class hierarchies.
- **Guideline 36:** Multiple inheritance should be handled with care. Multiple inheritance complicates the inheritance structure and might make the framework design hard to understand.
- **Guideline 37:** Only the leaves of an inheritance hierarchy in a framework should be concrete. Restructure the hierarchy instead of inheriting from a concrete class.
- **Guideline 38:** Use type preserving inheritance when the concrete leaves of the framework are derived from its superclasses. Both adding and cancelling inherited methods will violate the polymorphism.

3.5.4 Detailed Design

- **Guideline 39:** Methods should have few parameters. Methods with more than five parameters should be considered candidates for restructuring.
- **Guideline 40:** Let one method perform only one task. Parts of a methods performing several tasks might be common to several classes.
- **Guideline 41:** Keep a small public interface for a class. Classes with more than 25 methods should be considered candidates for restructuring.
- **Guideline 42:** If new abstractions are identified, introduce them in the appropriate model. Conceptual abstractions in the analysis models, and lower-level abstractions in the design model.

- **Guideline 43:** Keep method signatures consistent. Things that are the same should be named the same.

3.6 Implementation

3.6.2 Guidelines

3.6.2.1 Relationships

- **Guideline 44:** Comment all multiple inheritance thoroughly. Thorough documentation might make up for the complications multiple inheritance implies.
- **Guideline 45:** Avoid casting down the inheritance hierarchy. The methods in a subclass should be accessed through the superclass' interface.
- **Guideline 46:** Avoid using *friend* if possible, as the friend concept violates information hiding. It is better to make some member functions *friends* than to make a whole class a *friend*.
- **Guideline 47:** Restructure the inheritance hierarchies instead of using type-restrictive inheritance.
- **Guideline 48:** Do not use private inheritance. Private inheritance is not an object-oriented concept.

3.6.2.2 Classes and Methods

- **Guideline 49:** Inhibit abstract classes from being instantiated
- **Guideline 50:** All methods intended to be overloaded or redefined in subclasses must be declared as virtual.
- **Guideline 51:** Keep methods small, methods with more than 20 lines should be regarded candidates for modification.
- **Guideline 52:** Declare member methods const when possible. Declaring a method const ensures that invoking it will not affect the state of the object.
- **Guideline 53:** Declare parameters const when possible. Declaring a parameter const ensures that its value will not be changed in the method.
- **Guideline 54:** Eliminate explicit type checking on object types
- **Guideline 55:** Specify attributes as private. Specifying attributes as private hides the data representation and makes the class' interface stable.
- **Guideline 56:** Avoid implicit inline, as implementation in header files violates encapsulation. Use explicit inline instead.

3.6.2.3 Constructors and Destructors

- **Guideline 57:** Implement implicitly generated class methods. The compiler's implementations may result in unexpected behaviour.
- **Guideline 58:** When copying or assignment makes no sense, hide the copy constructor and the assignment operator in the private part of the class specification.
- **Guideline 59:** Always make destructors virtual in the base classes. Classes should deallocate the memory resources they use themselves.

4 Framework Reuse

4.2 Overview

- **Guideline 60:** Use domain analysis results, if available, to understand the characteristics of the component [Karl95, p.348].
- **Guideline 61:** Consult experienced people and every useful model [Karl95, p.349].
- **Guideline 62:** Carry out a complete study of the framework so it is fully understood and the appropriate choice can be made.

4.3 Analysis with Reuse

- **Guideline 63:** Identify reuse opportunities during the specification phase, and identify specific reuse requirements that support them [NATO91a; NATO91b; Karl95, p. 347].
- **Guideline 64:** Do not over-specify requirements [NATO91a,b, Karl95 p.357].
- **Guideline 65:** Use the products from the domain analysis to understand the context in which the application takes place, and its dynamics [Karl95, p. 359].
- **Guideline 66:** Use the domain analysis to determine and validate new application requirements[Karl95, p. 359].

4.4 Design with Reuse

- **Guideline 67:** Select well-documented frameworks [Karl95, p. 362].
- **Guideline 68:** Carry out some detailed design of the application in advance for evaluation of the framework.

- **Guideline 69:** Preserve the strategy implemented in the framework when adapting it [Karl95, p. 362].
- **Guideline 70:** Keep results from different phases separated
- **Guideline 71:** Use inheritance to customize a framework [Karl95, p. 363].

B CASE STUDY

The purpose of the case study is to provide a simple working example of the high-level development of a small framework.

The domain we chose is small and simple enough according to the time constraints of the thesis. The domain should be well known to both the readers and the authors of the thesis and requires no further investigation. The chosen domain was games of dice, which is a small domain whose few concepts are well known to most people.

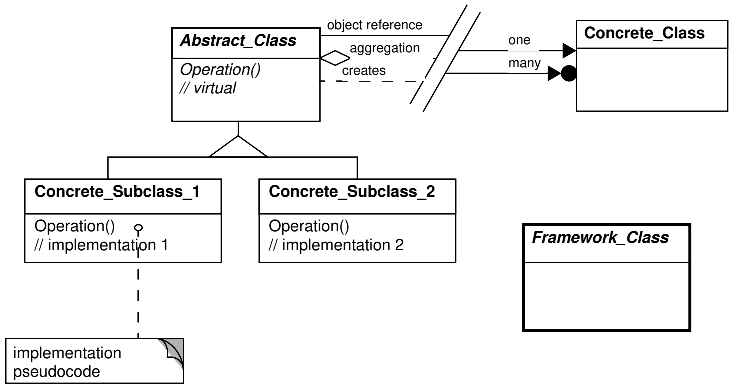
We decided to develop three applications of games of dice; The Game of Greed, Craps and Yatzy. The games were randomly chosen with no regard to similarities between the games or whether they where suitable to fit into a common framework or not.

Use cases should be used to describe then entire external behaviour of the system, but since the domain is well known, we have only provided a few use cases and interaction diagrams. The implementation of the framework is not in the scope of this case study and no consideration is taken to user interfaces.

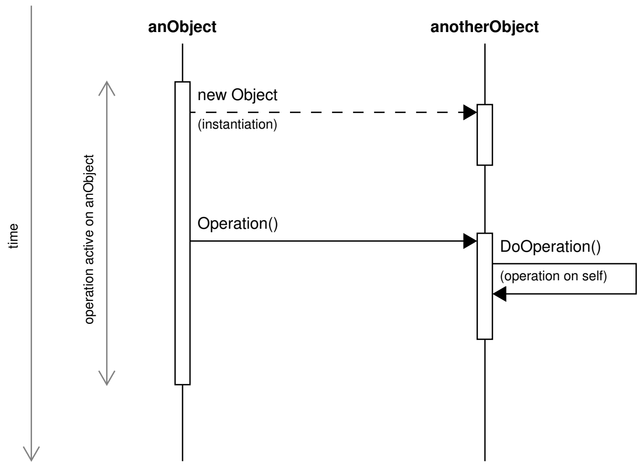
B.1 Notation

In the interaction diagrams and object models, we have used an OMT-based notation, defined by Gamma et al. in the Design Patterns book [Gam94].

B.1.1 Class Diagram Notation [Gam94]



B.1.2 Interaction Diagram Notation [Gam94]



B.2 The Analysis Phase

B.2.1 Requirements

We provided requirements on three applications. We also provided requirements on the framework and future requirements on the framework.

B.2.1.1 Dice Game Framework Requirements

Concept descriptions

Die: Cube. Six sides, numbered from one to six. When thrown it lands with one side up.

Throw: A die is thrown upon a surface. The side facing up is the result of the throw.

Player: A person participating in a game of dice.

Requirements

A player shall be able to throw any number of dice.

There shall be a dice container, dice can be added and removed and all dice can be thrown at once.

There shall be rules to a game of dice.

A player can bet, play a game according to the rules, decide if to stop or continue and throw the dice. He has somewhere to keep his money, or his current score.

There is an order of turns. A scheduler shall keep track of who is in turn.

It shall be possible to see when the game is over and who has won.

B.2.1.2 Yatzy Requirements

The Yatzy game uses five dice, of which an arbitrary number can be thrown together.

A Yatzy protocol looks like this:

Yatzy	Axel	Niklas
Ones		
Twos		
Threes		
Fours		
Fives		
Sixes		
Sum		
Bonus		
One Pair		
Two Pairs		
Threesome		
Foursome		
Small straight		
Large Straight		
Full House		
Chance		
Yatzy		
Sum		

Two or more players can participate in a game.

The calculation of Yatzy scores is supposed to be known by the reader.

Rules

When the game starts, the order of turns is decided upon, and the player to begin is chosen.

In the first throw of a turn, the player throws all five dice. He can chose to throw an optional number of dice again or to stop. When he is satisfied or he has thrown three times, he chooses in which row in the protocol to note the score. The score is calculated according to the chosen row and noted in the row.

The turn goes to the player to the left of him when he is done.

The game ends when all squares in the protocol are full (after 16 rounds) and the winner is the player with the highest score.

B.2.1.3 Greed Requirements

The Game of Greed uses five dice, of which an arbitrary number can be thrown together.

A Greed protocol looks like this:

Axel	Niklas

Two or more players can participate in a game.

The score of a throw is calculated as follows:
Three 1:s give 1000 points, three n:s give n*100 points. Single 1:s give 100 points, single 5:s give 50 points. Each die can only give points once.

Rules

When the game starts, the order of turns is decided upon, and the player to begin is chosen.

In the first throw of a turn, the player throws all five dice. If the score is >300 p., he can choose to throw the dice that did not give any points again or to stop. He continues to throw until the score of a throw is 0, or until he chooses to stop. When finished he notes the score in the current square of his column in the protocol.

The turn goes to the player to the left of him when he is done.

The game ends when a player reaches 5000 p and all players have had an equal number of turns. The player with the highest score wins.

B.2.1.4 Craps Requirements

The Craps game uses two dice, which are always thrown together.

Two or more players can participate in a game. Each player has a wallet in which he stores his money, and from which he can take money to bet.

Rules

When the game starts, the order of turns is decided upon, and the player to begin is chosen.

In a round, all players except from the player to throw bets money to the kitty, saying the thrower will fail. The thrower bets, no more than is in the kitty, saying he will make it.

The player throws the dice, and if the sum of the eyes of the two dice (“the sum”) is 7 or 11, he wins. If the sum is 2, 3 or 12 he loses. If the sum is 4, 5, 6, 8, 9 or 10 he gets a second chance. The player to the left throws the dice, and the player has to throw the same sum as the player to the left to win, and the sum can not be 7.

If he wins, he can take out twice his bet from the kitty.

The turn goes to the player to the left of him when he is done.

The game ends when one or more of the players goes bankrupt, and the winner is the player with the most money.

B.2.1.5 Future requirements

The design shall be possible to extend to a general framework for application of games of dice.

The framework should be able to extend into both human and computer players.

B.2.1.6 Requirements modifications

Craps

After renegotiations with the customer the requirement:

“The player throws the dice, and if the sum of the eyes of the two dice (“the sum”) is 7 or 11, he wins. If the sum is 2, 3 or 12 he loses. If the sum is 4, 5, 6, 8, 9 or 10 he gets a second chance. The player to the left throws the dice, and the player has to throw the same sum as the player to the left to win, and the sum can not be 7.”

is changed to the new requirement:

“The player throws the dice, and if the sum of the eyes of the two dice (“the sum”) is 7 or 11, he wins. If the sum is 2, 3 or 12 he loses. If the sum is 4, 5, 6, 8, 9 or 10 he gets a second chance, and he has to throw the same sum twice in a row to win, and the sum can not be 7.”

The change in the requirement did not have any negative impact on the functionality of the application. The requirement is changed to let the Craps application fit better into the framework promoting reuse.

B.2.2 Use Cases

B.2.2.1 Yatzy Use Cases

Use case: Start of game

Two players start playing with five dice according to Yatzy rules. They write their names at the top of a column in the Yatzy protocol.

Use case: One turn

The first player throw all five dice, and get three fives. He keeps the fives and throws the other two dice again. He get two more fives and notes fifty points at the "Yatzy" square of his column in the Yatzy protocol.

Use case: End of game

The players take turns playing until the protocol is full, i.e. 16 times. The first player wins, since he has a score of 362 and the second player has a score of 253.

B.2.2.2 Greed Use Cases

Use case: Start of game

Two players start playing with five dice according to Greed rules. They write their names at the top of a column each in the Greed protocol.

Use case: One turn 1

The first player throws all five dice, and get three ones, a two and a three. He keeps the ones and throws the other two dice again. He get two threes, and scores zero, which he notes in the current square of his column in the Greed protocol.

Use case: One turn 2

The second player throws all five dice, and get two fives, a two, a three and a one. Since the score is 200, and less than 300, he scores zero, which he notes in the current square of his column in the Greed protocol.

Use case: End of game

The first player reaches 5020 p, and the second player gets a last chance. The second player throws all five dice, and gets three ones, a two and a three. He chooses to stop, and scores 1000 p, which he notes in the current square of his column in the Greed protocol. His total score is 5100, and he wins.

B.2.2.3 Craps Use Cases

Use case: Start of game

Two players start playing with two dice according to Craps rules. They have a kitty to place the bets in.

Use case: One turn 1

The second player bets $10 to the kitty, saying the first player will fail. The first player sees there are $10 in the kitty and places the maximum $10 bet. The first player throws the two dice and the sum is 7, and he gets to take $20 from the kitty.

Use case: One turn 2

The second player bets $5 to the kitty, saying the first player will fail. The first player sees there are $5 in the kitty and places a $2 bet, saying he will make it. The first player throws the two dice and the sum is 4, and he gets a second chance. He throws the two dice and the sum is 6. He throws the two dice and again the sum is 6, and he gets to take $4 from the kitty.

Use case: End of game

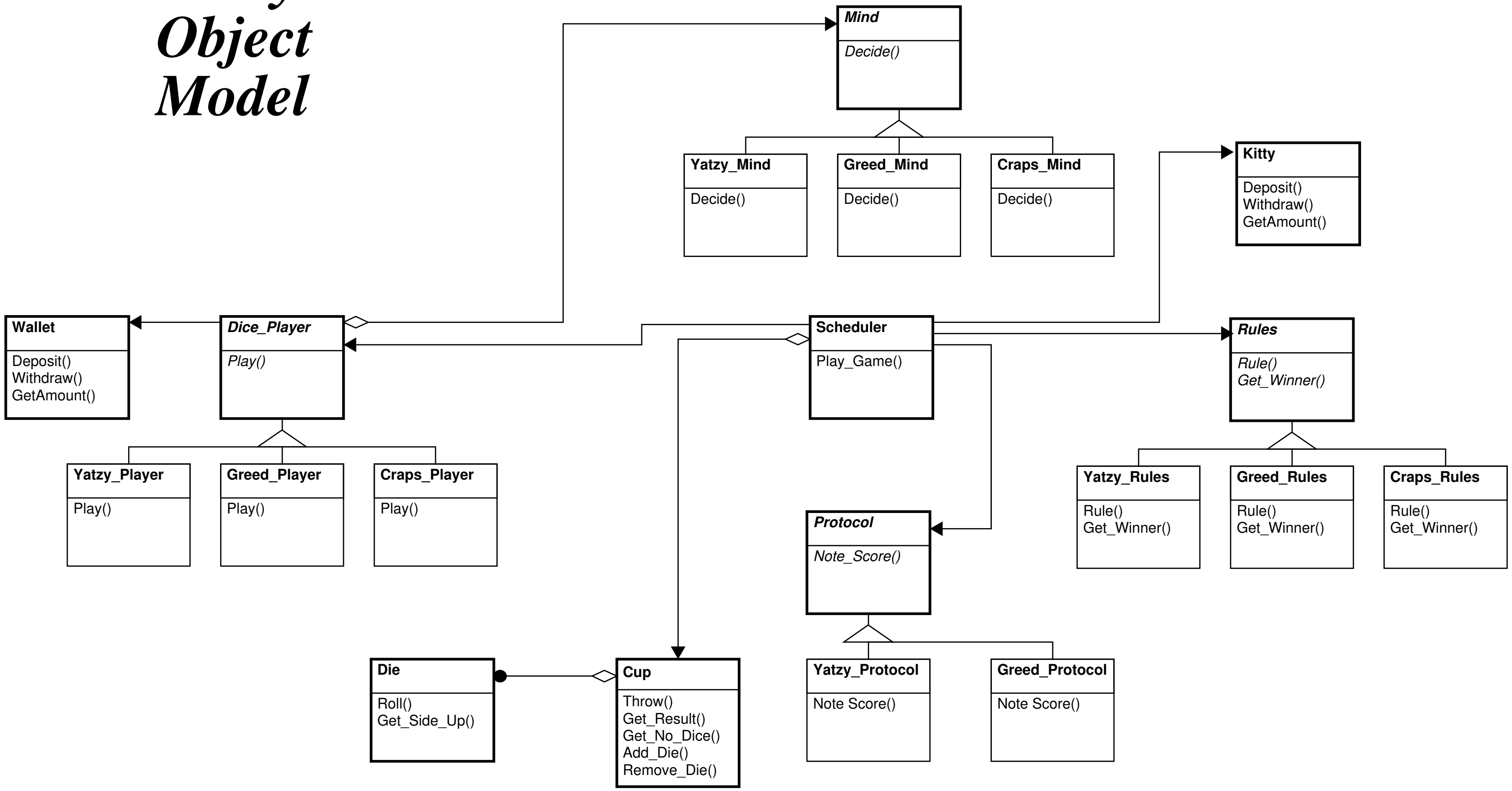
The first player goes bankrupt, and the second player wins the game.

B.2.3 The Object Model

The process of identifying the objects of a framework is started by the identification of the objects of each application. The object identification of an application should be done in parallel with the other applications to achieve consistency between the applications in naming of similar objects.

The above also applies to the design process of the analysis models. One model should be designed for each application. The analysis model of each application should be designed in parallel with the other applications' analysis models to achieve maximum similarity between the architectures of the applications. Designing the analysis models in parallel makes it easier to find similarities between the models, that can be moved into the framework.

Analysis Object Model



B.2.3.1 Yatzy Object Responsibilities

Player

- Be able to participate in a game of Yatzy.
- Be able to perform a turn of the game, according to the rules.
- Be able to read the protocol.

Scheduler

- Keep track of the players participating in a round of the game.
- Keep track of who is in turn.
- Judge if the game is over.
- Judge who has won.

Die

- Be able to produce a random integer in [1..6], when thrown.
- Remember the result until thrown again.

Cup

- Contains a number of dice.
- Throw contained dice and tell their result, respectively.

Protocol

- Keep track of the cells of a Yatzy protocol, with 18 rows and n columns.
- Be able to store and tell the number in each cell.

Rules

- Calculate the result of a throw, depending on the dice and which row in the protocol to use.

- Judge if the player may continue, depending on the number of throws he has done in this round.
- Judge who is the final winner.

Mind

- Make a decision to throw the dice.
- Make a decision which dice to throw again.
- Make a stop/continue decision.
- Make a decision about where to put the result in the protocol.

B.2.3.2 Greed Object Responsibilities

Player

- Be able to participate in a game of Greed.
- Be able to perform a turn of the game, according to the rules.
- Be able to read the protocol.

Scheduler

See Yatzy.

Die

See Yatzy.

Cup

See Yatzy.

Protocol

- Keep track of the cells of a greed protocol, with n rows and m columns.
- Be able to store and tell a number in each cell.

Rules

- Be able to calculate the score for a throw with n dice (according to an algorithm), and which dice can be thrown again.
- Judge if the player may continue, depending on the score and the number of throws he has done in this round.
- Judge who is the final winner.

Mind

- Make a decision to throw the dice.
- Make a decision to stop or continue.

B.2.3.3 Craps Object Responsibilities

Player

- Be able to participate in a game of Craps.
- Be able to perform a turn of the game, according to the rules.
- Be able to bet money from his wallet to a kitty.
- Be able to receive money from kitty to wallet.
- Be able to tell how much money he has got.
- Be able to check the amount in the kitty.

Scheduler

See Yatzy.

Die

See Yatzy.

Cup

See Yatzy.

Kitty

- Be able to receive money.
- Be able to pay money.

Rules

- Add up the dice.
- Judge if the player has won/has lost/gets a second chance, depending on the sum and the number of throws he has done.
- Judge who is the final winner.

Mind

- Make a decision to throw the dice.
- Bet an amount.

B.2.3.4 Identification of Similar Objects

We found that many classes were common to all three applications. These classes were moved into the framework. However, some objects were not completely similar:

Player

This object exists in all three games (and in all other games of dice to). Some behaviour and attributes is common to all rules and some is not.

Analysis decision:
Commonalities between the rules of the three games are moved into the abstract class *Player*. The differences between the applications are moved into three different specializations: *Yatzy_Player*, *Greed_Player* and *Craps_Player*. The class Mind should be moved into the framework.

Protocol

This object does not exists in Craps but it exists in both Yatzy and Greed. The object protocol is not completely similar between the games Yatzy and Greed, but the differences may be modelled through specialisation.

Analysis decision:
The behaviour and attributes that are similar between the two protocol objects is moved into the abstract class *Protocol*. The differences is moved into the specializations *Yatzy_Protocol* and *Greed_Protocol*. The abstract class is moved into the framework.

Kitty

This object only exists in Craps and it should only be modelled outside the framework. However, many games of dice use a Kitty and considering future applications the object Kitty should be moved into the framework.

Analysis decision:
The object *Kitty* should be moved into the framework.

Rules

This object exists in all three games (and in all other games of dice to). Some behaviour and attributes is common to all rules and some is not.

Analysis decision:
Commonalities between the rules of the three games are moved into the abstract class *Rules*. The differences between the applications are moved into three different specializations: *Yatzy_Rules*, *Greed_Rules* and *Craps_Rules*. The class Rules should be moved into the framework.

Mind

This object exists in all three games (and in all other games of dice to). Some behaviour and attributes is common to all rules and some is not.

Analysis decision:
Commonalities between the rules of the three games are moved into the abstract class *Mind*. The differences between the applications are moved into three different specializations: *Yatzy_Mind*, *Greed_Mind* and *Craps_Mind*. The class Mind should be moved into the framework.

B.3 The Design Phase

B.3.1 Architectural Design

During this phase it occurred that the object *Wallet* had the same functionality as the object *Kitty*, and a new object was identified and introduced into the design model, the *Unit Container*.

B.3.1.1 Framework Solutions

Mechanisms are needed to decouple the instantiation of application specific classes from the framework. The design pattern “Abstract Factory” provides a way to isolate the instantiation from the framework. For example, when a Yatzy game is initiated, the *Yatzy_Player* objects should not be instantiated inside the *Scheduler*, as this couples the framework to one of its applications. By using an “abstract factory”, the Scheduler can use instances of *Yatzy_Player* without having to know their actual type.

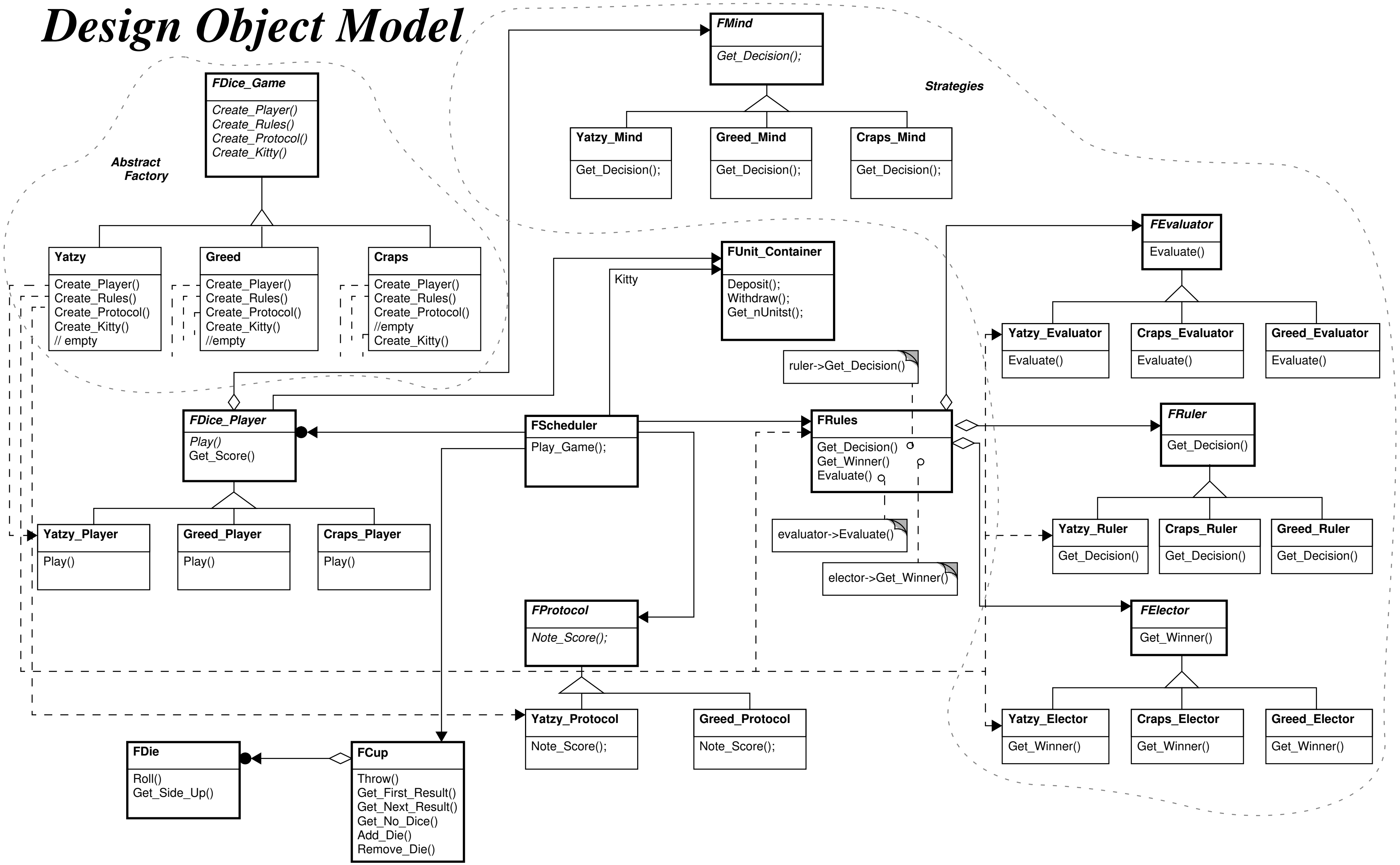
Upon discovering that the *Rules* classes for the applications only differed in the algorithms used by its methods, we saw a possibility to apply the “Strategy” design pattern. By doing this we discovered further possibilities of generalisation between the *Ruler* and *Mind* classes, but these generalisations are not included in the design models. There are other possibilities to apply the “Strategy” pattern, e.g. the *Play*() method in the *Dice_Player* class hierarchy. Using “Strategy” makes the *Rules* class independent of the algorithms used to “calculate” the decisions.

B.3.1.2 The Object Model

Interaction diagrams were used to identify the methods of the design objects.

The design objects are added to the object model, and we have marked out the design patterns used in the design. How the “Abstract Factory” instantiates the application specific classes is exemplified for the Yatzy factory, and the other applications work in a similar way.

Design Object Model



B.3.2 Interaction Diagrams

We used interaction diagrams to identify the collaborations between the objects and to identify the methods. The interaction diagrams shows how the system handles the functionality in the use cases.

B.3.2.1 Yatzy

Use case 1: Start of game

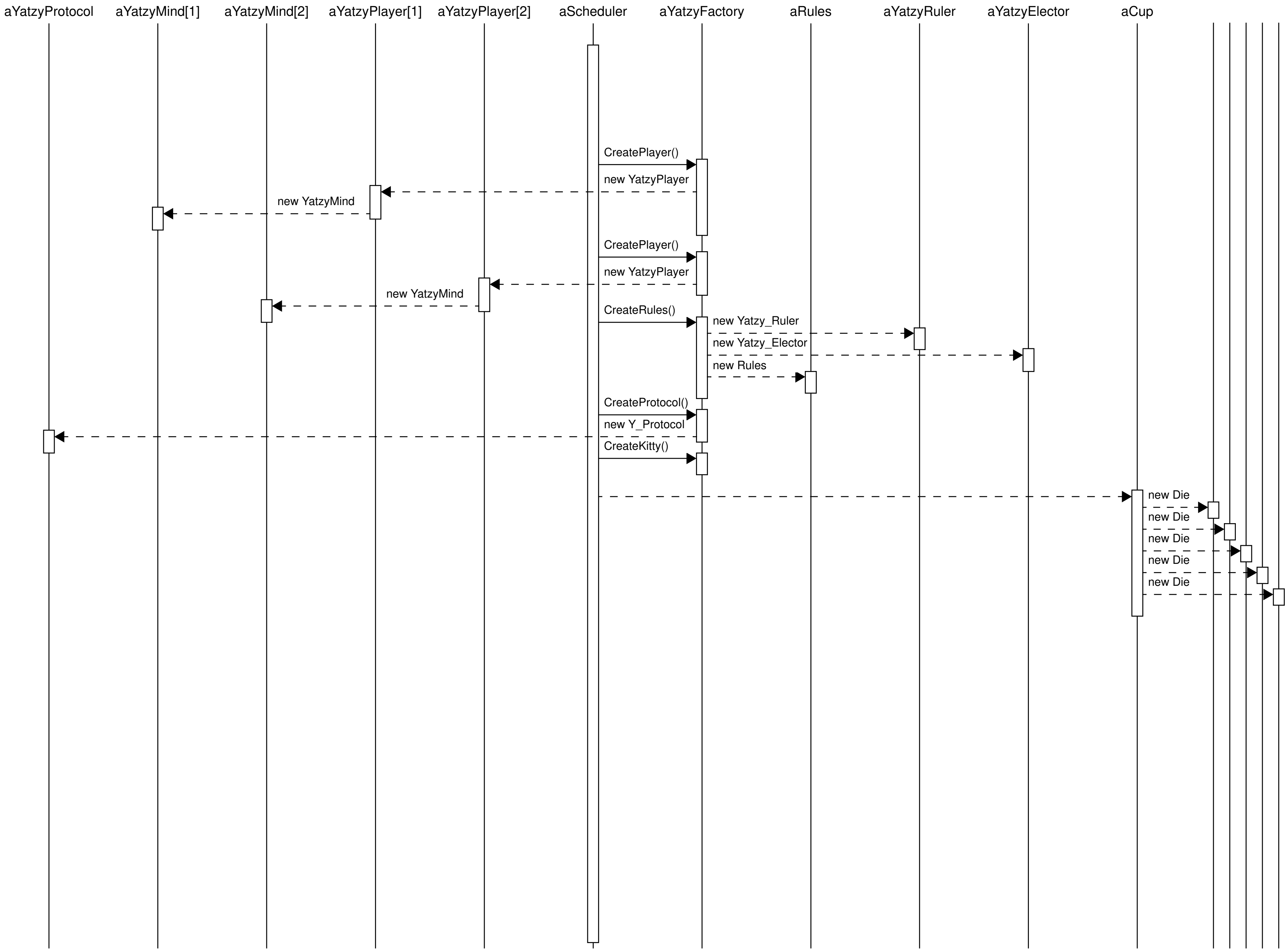
Two players start playing with five dice according to Yatzy rules. They write their names at the top of a column in the Yatzy protocol.

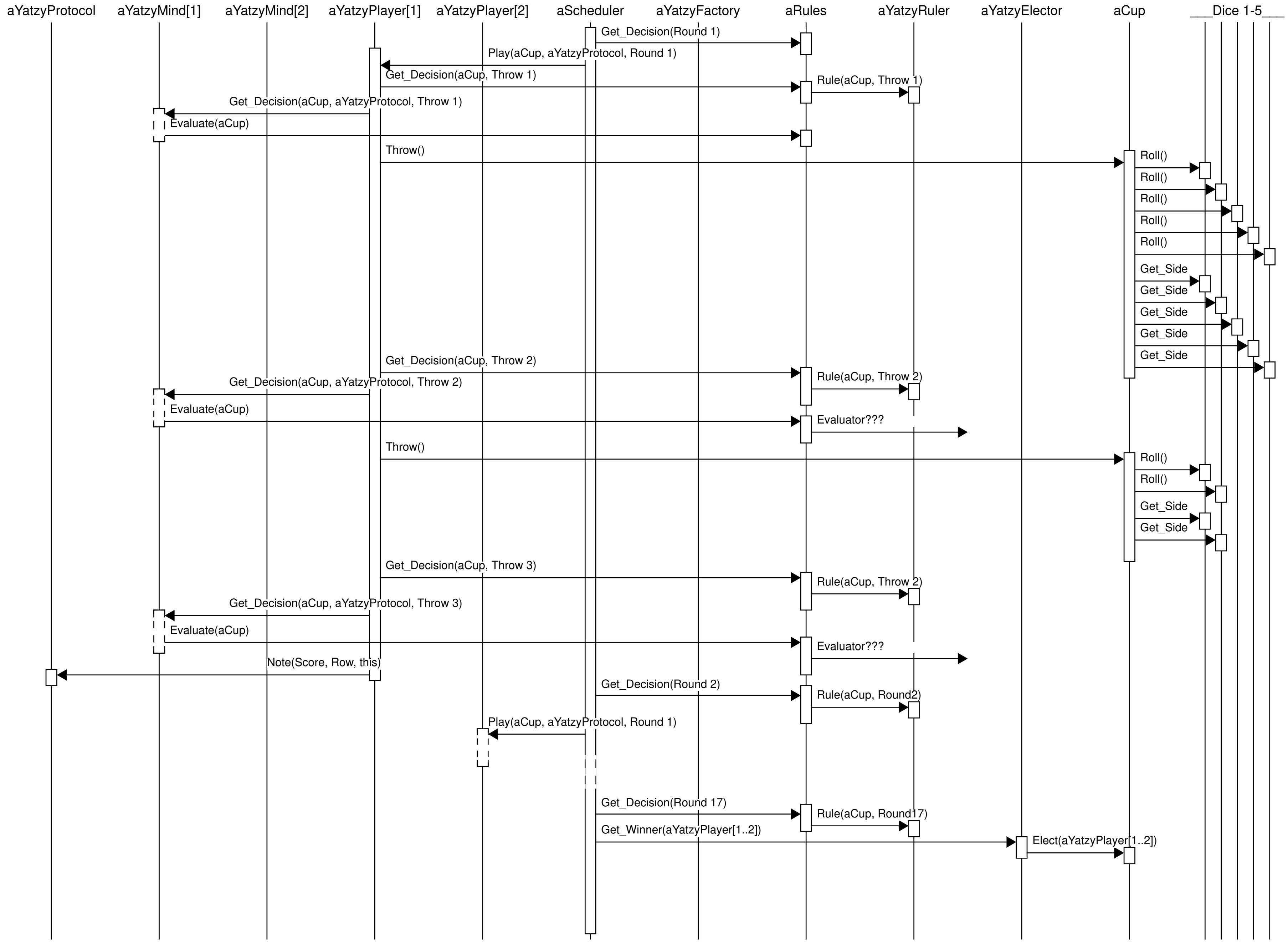
Use case 2: One turn

The first player throw all five dice, and get three fives. He keeps the fives and throws the other two dice again. He get two more fives and notes fifty points at the “Yatzy” square of his column in the Yatzy protocol.

Use case 3: End of game

The players take turns playing until the protocol is full, i.e. 16 times. The first player wins, since he has a score of 362 and the second player has a score of 253.


aYatzyProtocol
aYatzyMind[1]
aYatzyMind[2]
aYatzyPlayer[1]
aYatzyPlayer[2]
aScheduler
aYatzyFactory
aRules
aYatzyRuler
aYatzyElector
aCup
CreatePlayer()
new YatzyPlayer
new YatzyMind
CreatePlayer()
new YatzyPlayer
new YatzyMind
CreateRules()
new Yatzy_Ruler
new Yatzy_Elector
new Rules
CreateProtocol()
new Y_Protocol
CreateKitty()
new Die
new Die
new Die
new Die
new Die


aYatzyProtocol
aYatzyMind[1]
aYatzyMind[2]
aYatzyPlayer[1]
aYatzyPlayer[2]
aScheduler
aYatzyFactory
aRules
aYatzyRuler
aYatzyElector
aCup
___Dice 1-5___
Get_Decision(Round 1)
Play(aCup, aYatzyProtocol, Round 1)
Get_Decision(aCup, Throw 1)
Rule(aCup, Throw 1)
Get_Decision(aCup, aYatzyProtocol, Throw 1)
Evaluate(aCup)
Throw()
Roll()
Roll()
Roll()
Roll()
Roll()
Get_Side
Get_Side
Get_Side
Get_Side
Get_Side
Get_Decision(aCup, Throw 2)
Rule(aCup, Throw 2)
Get_Decision(aCup, aYatzyProtocol, Throw 2)
Evaluate(aCup)
Evaluator???
Throw()
Roll()
Roll()
Get_Side
Get_Side
Get_Decision(aCup, Throw 3)
Rule(aCup, Throw 2)
Get_Decision(aCup, aYatzyProtocol, Throw 3)
Evaluate(aCup)
Evaluator???
Note(Score, Row, this)
Get_Decision(Round 2)
Rule(aCup, Round2)
Play(aCup, aYatzyProtocol, Round 1)
Get_Decision(Round 17)
Rule(aCup, Round17)
Get_Winner(aYatzyPlayer[1..2])
Elect(aYatzyPlayer[1..2])

B.3.1.3 Greed

Use case 1: Start of game

Two players start playing with five dice according to Greed rules. They write their names at the top of a column each in the Greed protocol.

Use case 2: One turn 1

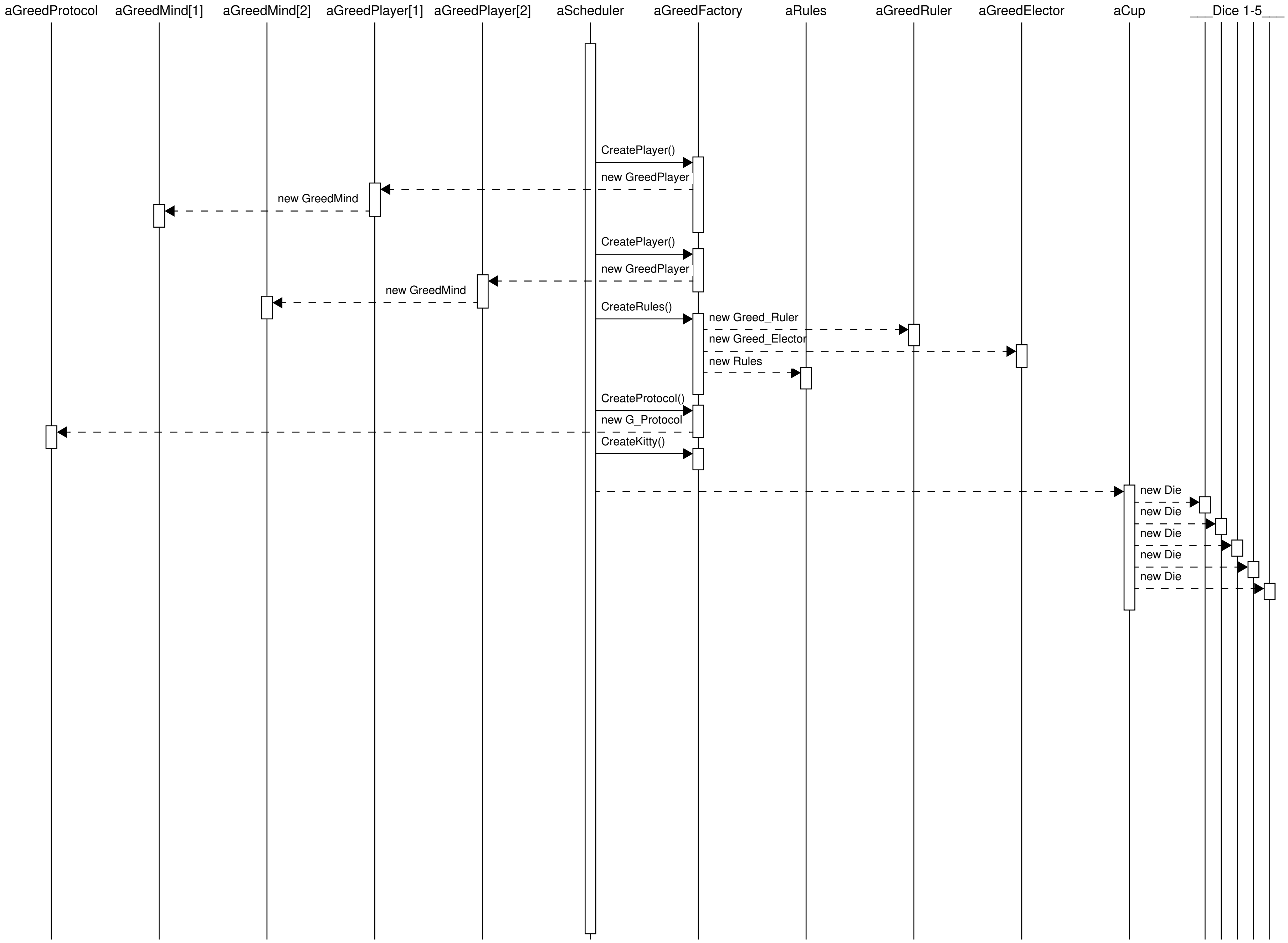
The first player throws all five dice, and get three ones, a two and a three. He keeps the ones and throws the other two dice again. He get two threes, and scores zero, which he notes in the current square of his column in the Greed protocol.

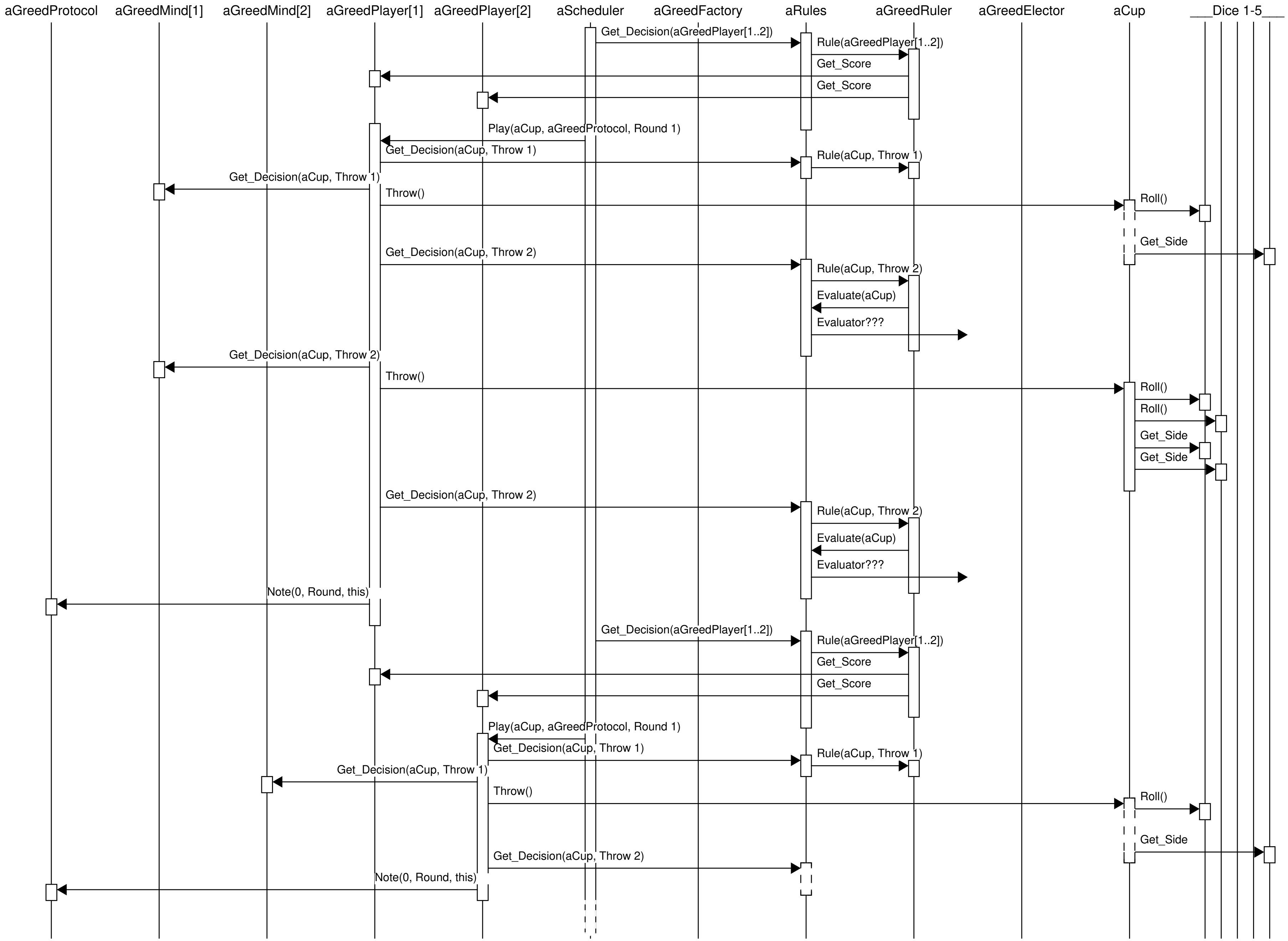
Use case 3: One turn 2

The second player throws all five dice, and get two fives, a two, a three and a one. Since the score is 200, and less than 300, he scores zero, which he notes in the current square of his column in the Greed protocol.

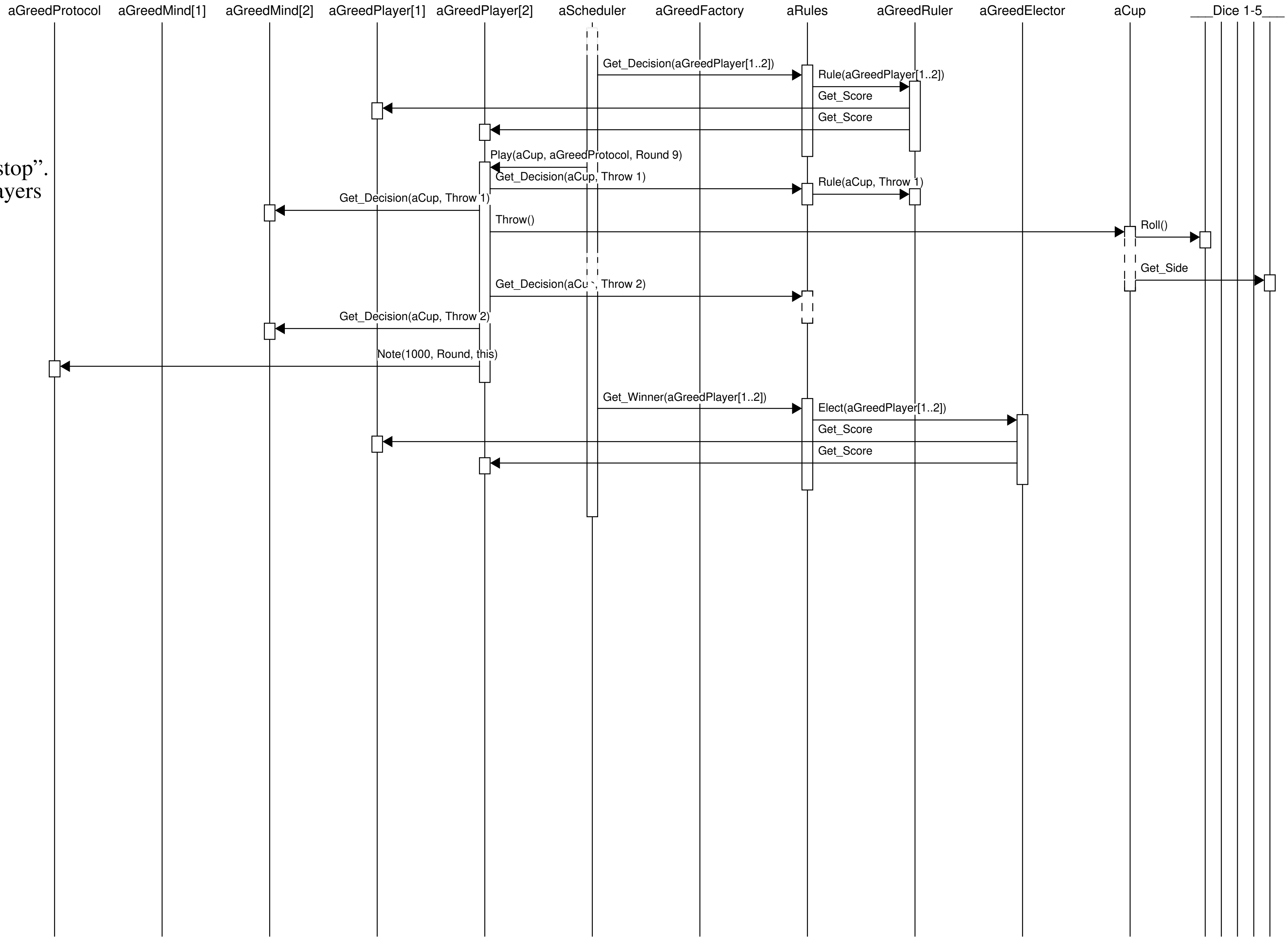
Use case 4: End of game

The first player reaches 5020 p, and the second player gets a last chance. The second player throws all five dice, and gets three ones, a two and a three. He chooses to stop, and scores 1000 p, which he notes in the current square of his column in the Greed protocol. His total score is 5100, and he wins.


aGreedProtocol
aGreedMind[1]
aGreedMind[2]
aGreedPlayer[1]
aGreedPlayer[2]
aScheduler
aGreedFactory
aRules
aGreedRuler
aGreedElector
aCup
___Dice 1-5___
CreatePlayer()
new GreedPlayer
new GreedMind
CreatePlayer()
new GreedPlayer
new GreedMind
CreateRules()
new Greed_Ruler
new Greed_Elector
new Rules
CreateProtocol()
new G_Protocol
CreateKitty()
new Die
new Die
new Die
new Die
new Die


aGreedProtocol
aGreedMind[1]
aGreedMind[2]
aGreedPlayer[1]
aGreedPlayer[2]
aScheduler
aGreedFactory
aRules
aGreedRuler
aGreedElector
aCup
___Dice 1-5___
Get_Decision(aGreedPlayer[1..2])
Rule(aGreedPlayer[1..2])
Get_Score
Get_Score
Play(aCup, aGreedProtocol, Round 1)
Get_Decision(aCup, Throw 1)
Rule(aCup, Throw 1)
Get_Decision(aCup, Throw 1)
Throw()
Roll()
Get_Side
Get_Decision(aCup, Throw 2)
Rule(aCup, Throw 2)
Evaluate(aCup)
Evaluator???
Get_Decision(aCup, Throw 2)
Throw()
Roll()
Roll()
Get_Side
Get_Side
Get_Decision(aCup, Throw 2)
Rule(aCup, Throw 2)
Evaluate(aCup)
Evaluator???
Note(0, Round, this)
Get_Decision(aGreedPlayer[1..2])
Rule(aGreedPlayer[1..2])
Get_Score
Get_Score
Play(aCup, aGreedProtocol, Round 1)
Get_Decision(aCup, Throw 1)
Rule(aCup, Throw 1)
Get_Decision(aCup, Throw 1)
Throw()
Roll()
Get_Side
Get_Decision(aCup, Throw 2)
Note(0, Round, this)

Player 1 has 5000 p., the decision is “stop”. The other players get a chance



B.3.1.4 Craps

Use case 1: Start of game

Two players start playing with two dice according to Craps rules. They have a kitty to place the bets in.

Use case 2: One turn 1

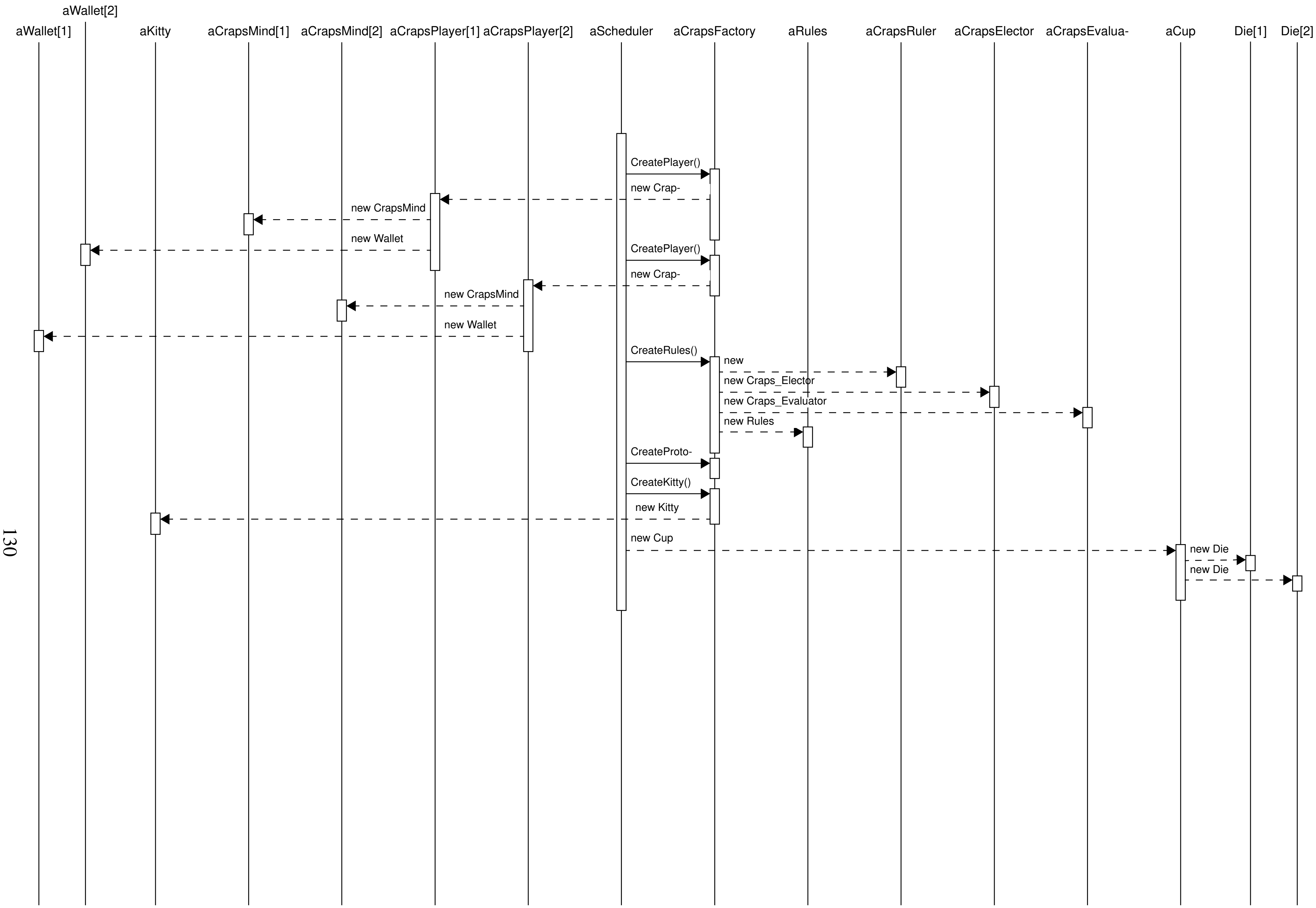
The second player bets $10 to the kitty, saying the first player will fail. The first player sees there are $10 in the kitty and places the maximum $10 bet. The first player throws the two dice and the sum is 7, and he gets to take $20 from the kitty.

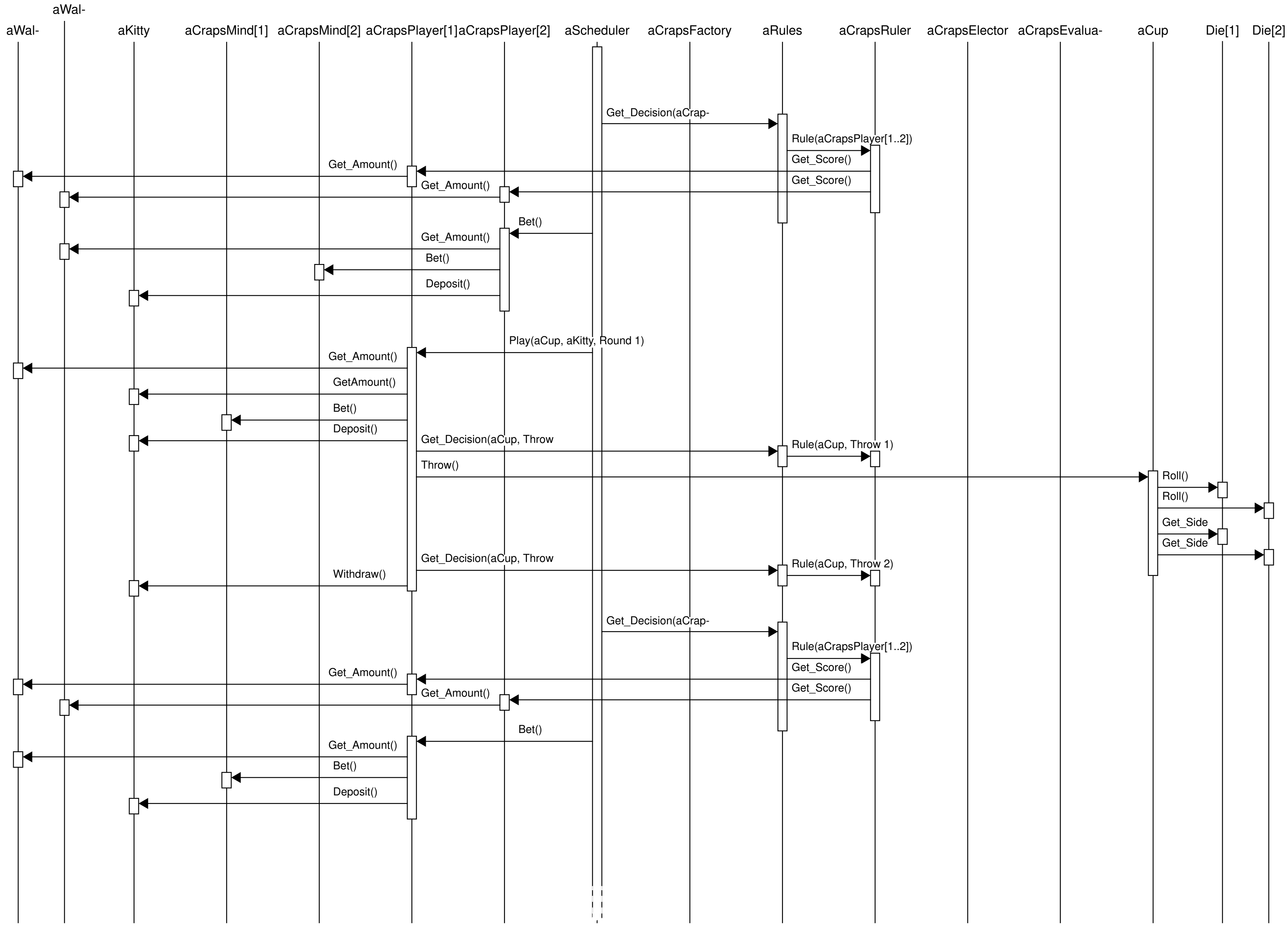
Use case 3: One turn 2

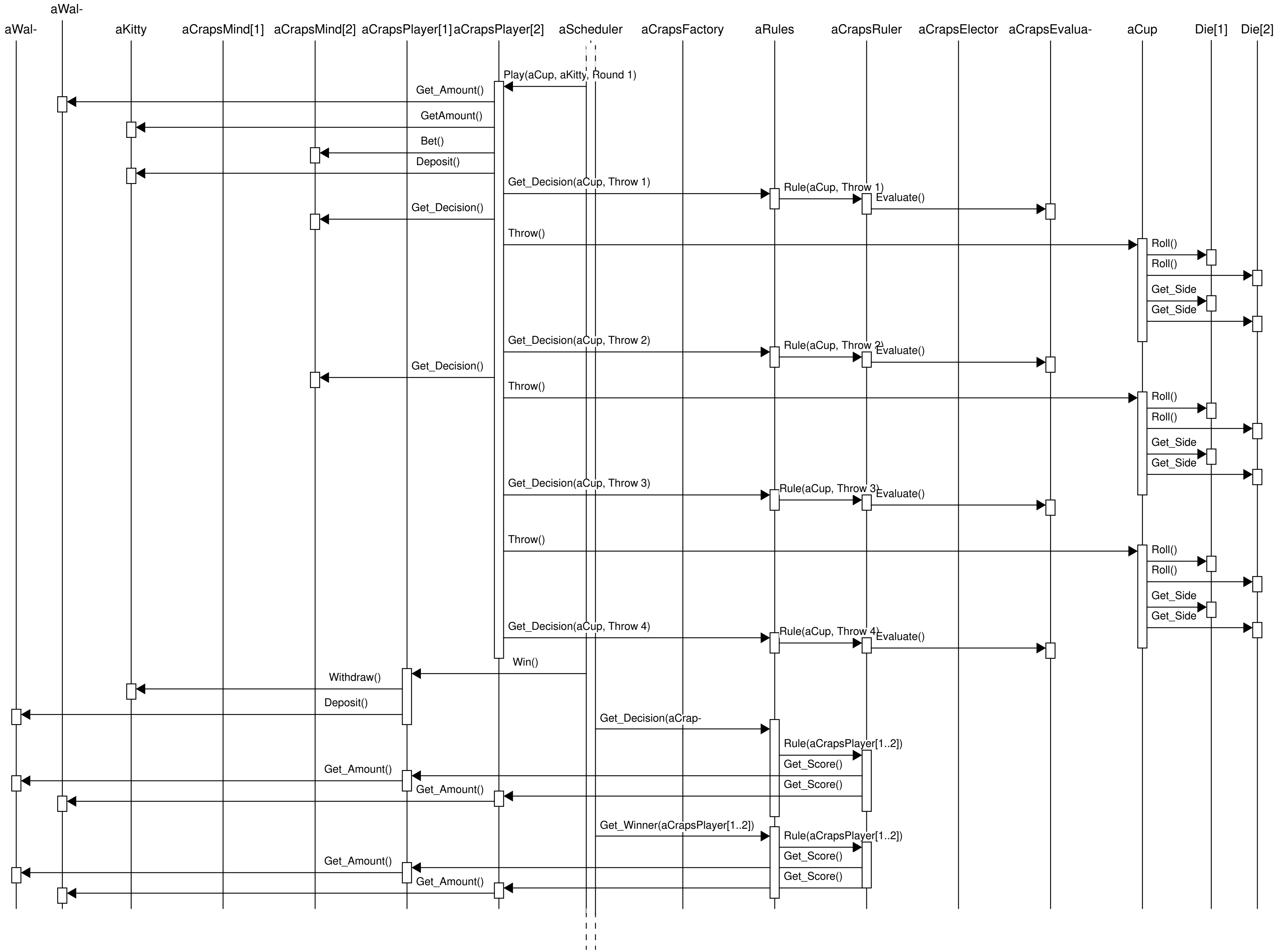
The second player bets $5 to the kitty, saying the first player will fail. The first player sees there are $5 in the kitty and places a $2 bet, saying he will make it. The first player throws the two dice and the sum is 4, and he gets a second chance. He throws the two dice and the sum is 6. He throws the two dice and again the sum is 6, and he gets to take $4 from the kitty.

Use case 4: End of game

The first player goes bankrupt, and the second player wins the game.


aWallet[2]
aWallet[1]
aKitty
aCrapsMind[1]
aCrapsMind[2]
aCrapsPlayer[1]
aCrapsPlayer[2]
aScheduler
aCrapsFactory
aRules
aCrapsRuler
aCrapsElector
aCrapsEvalua-
aCup
Die[1]
Die[2]
CreatePlayer()
new Crap-
new CrapsMind
new Wallet
CreatePlayer()
new Crap-
new CrapsMind
new Wallet
CreateRules()
new
new Craps_Elector
new Craps_Evaluator
new Rules
CreateProto-
CreateKitty()
new Kitty
new Cup
new Die
new Die


aWal-
aWal-
aKitty
aCrapsMind[1]
aCrapsMind[2]
aCrapsPlayer[1]
aCrapsPlayer[2]
aScheduler
aCrapsFactory
aRules
aCrapsRuler
aCrapsElector
aCrapsEvalua-
aCup
Die[1]
Die[2]
Get_Decision(aCrap-
Rule(aCrapsPlayer[1..2])
Get_Score()
Get_Amount()
Get_Score()
Get_Amount()
Bet()
Get_Amount()
Bet()
Deposit()
Play(aCup, aKitty, Round 1)
Get_Amount()
GetAmount()
Bet()
Deposit()
Get_Decision(aCup, Throw
Rule(aCup, Throw 1)
Throw()
Roll()
Roll()
Get_Side
Get_Side
Get_Decision(aCup, Throw
Withdraw()
Rule(aCup, Throw 2)
Get_Decision(aCrap-
Rule(aCrapsPlayer[1..2])
Get_Score()
Get_Amount()
Get_Score()
Get_Amount()
Bet()
Get_Amount()
Bet()
Deposit()


aWal-
aWal-
aKitty
aCrapsMind[1]
aCrapsMind[2]
aCrapsPlayer[1]
aCrapsPlayer[2]
aScheduler
aCrapsFactory
aRules
aCrapsRuler
aCrapsElector
aCrapsEvalua-
aCup
Die[1]
Die[2]
Play(aCup, aKitty, Round 1)
Get_Amount()
GetAmount()
Bet()
Deposit()
Get_Decision(aCup, Throw 1)
Rule(aCup, Throw 1)
Evaluate()
Get_Decision()
Throw()
Roll()
Roll()
Get_Side
Get_Side
Get_Decision(aCup, Throw 2)
Rule(aCup, Throw 2)
Evaluate()
Get_Decision()
Throw()
Roll()
Roll()
Get_Side
Get_Side
Get_Decision(aCup, Throw 3)
Rule(aCup, Throw 3)
Evaluate()
Throw()
Roll()
Roll()
Get_Side
Get_Side
Get_Decision(aCup, Throw 4)
Rule(aCup, Throw 4)
Evaluate()
Win()
Withdraw()
Deposit()
Get_Decision(aCrap-
Rule(aCrapsPlayer[1..2])
Get_Score()
Get_Score()
Get_Amount()
Get_Amount()
Get_Winner(aCrapsPlayer[1..2])
Rule(aCrapsPlayer[1..2])
Get_Score()
Get_Score()
Get_Amount()
Get_Amount()

B.3.3 Detailed Design

In the detailed design, we fully define the public interfaces and attributes of the framework and application classes. We use a C++ syntax with no regard to pointers etc.

B.3.3.1 Framework

```
enum TDecision {stop, continue, win, lose}


class FDice_Player{

  protected:
    int nThrow;
    FMind aMind;
    FUnit_Container aWallet;

  public:
    FDice_Player();
    virtual int Play();
    int Get_Score();
}

class FScheduler{

  private:
    FDice_Player array Players[100];
    int nRound;

  public:
    FScheduler(int nPlayers, int nDice,
               FDiceGame aGame);
    Play_Game();
}

class FDie{

  private:
    int nSideUp;

  public:
    int Roll();
    int Get_Side_Up();
}

class FCup{

  private:
    int nDice;

  public:
```

```
    Throw();
    int Get_First_Result();
    int Get_Next_Result();
    int Get_No_Dice();
    Add_Die(Die aDie);
    Remove_Die();
}

class FProtocol{

  public:
    virtual Note_Score(int nScore, int nRow,
                       FDice_Player aPlayer);
}

class FRules {

  private:
    FRuler aRuler;
    FElector anElector;
    FEvaluator anEvaluator;

  public:
    FRules(FRuler aRuler, FElector anElector,
           FEvaluator anEvaluator);
    TDecision Get_Decision(FCup aCup, int nRound,
                           int nThrow,
                           FPlayer array Players);
    FPlayer Get_Winner(FPlayer array Players);
    int Evaluate(FCup aCup);
}


class FRuler{

  public:
    virtual TDecision GetDecisison(FCup aCup,
                                   int nRound,
                                   int nThrow,
                                   FPlayer array
                                   Players);
}

class FElector{

  public:
   virtual FPlayer Get_Winner(FPlayer array Players);
}

class FEvaluator{

  public:
    virtual int Evaluate(FCup aCup);
```

```
}

class FMind{

  public:
    virtual TDecisison Get_Decision(FCup aCup,
                                    int nRound,
                                    int nThrow,
                                    FPlayer array
                                    Players);
}

class FDice_Game{

  public:
    virtual FPlayer Create_Player();
    virtual FRules Create_Rules();
    virtual FProtocol Create_Protocol();
    virtual FUnit_Container Create_Kitty();
}
```

B.3.3.2 Yatzy

```
class Yatzy_Player: public FDice_Player{

  public:
    Yatzy_Player();
    int Play();
}

class Yatzy_Protocol: public FProtocol{

  public:
    Note_Score(int nScore, int nRow,
               FDice_Player aPlayer);
}

class Yatzy_Ruler: public FRuler{

  public:
    TDecision Get_Decision(FCup aCup, int nRound,
                           int nThrow,
                           FPlayer array Players);
}

class Yatzy_Elector: public FElector{

  public:
    FPlayer Get_Winner(FPlayer array Players);
}

class Yatzy_Evaluator: public FEvaluator{
```

```
  public:
    int Evaluate(FCup aCup);
}

class Yatzy_Mind: public FMind{

  public:
    TDecisison Get_Decision(FCup aCup, int nRound,
                            int nThrow,
                            FPlayer array Players);
}

class Yatzy: public FDice_Game{

  public:
    FPlayer Create_Player();
    FRules Create_Rules();
    FProtocol Create_Protocol();
    FUnit_Container Create_Kitty();
    // empty
}
```

B.3.3.3 Greed

```
class Greed_Player: public FDice_Player{

  public:
    Greed_Player();
    int Play();
}

class Greed_Protocol: public FProtocol{

  public:
    Note_Score(int nScore, int nRow,
               FDice_Player aPlayer);
}

class Greed_Rule: public FRuler{

  public:
    TDecision Get_Decision(FCup aCup, int nRound,
                           int nThrow,
                           FPlayer array Players);
}

class Greed_Elector: public FElector{

  public:
    FPlayer Get_Winner(FPlayer array Players);
}
```

```
class Greed_Evaluator: public FEvaluator{

  public:
    int Evaluate(FCup aCup);
}

class Greed_Mind: public FMind{

  public:
    TDecisison Get_Decision(FCup aCup, int nRound,
                            int nThrow,
                            FPlayer array Players);
}

class Greed: public FDice_Game{

  public:
    FPlayer Create_Player();
    FRules Create_Rules();
    FProtocol Create_Protocol();
    FUnit_Container Create_Kitty();
    // empty
}
```

B.3.3.4 Craps

```
class Craps_Player: public FDice_Player{

  public:
    Craps_Player();
    int Play();
}

class Craps_Protocol: public FProtocol{

  public:
    Note_Score(int nScore, int nRow,
               FDice_Player aPlayer);
}

class Craps_Rule: public FRuler{

  public:
    TDecision Get_Decision(FCup aCup, int nRound,
                           int nThrow,
                           FPlayer array Players);
}

class Craps_Elector: public FElector{

  public:
```

```
    FPlayer Get_Winner(FPlayer array Players);
}

class Craps_Evaluator: public FEvaluator{

  public:
    int Evaluate(FCup aCup);
}

class Craps_Mind: public FMind{

  public:
    TDecisison Get_Decision(FCup aCup, int nRound,
                            int nThrow,
                            FPlayer array Players);
}

class Craps: public FDice_Game{

  public:
    FPlayer Create_Player();
    FRules Create_Rules();
    FProtocol Create_Protocol();
    // empty
    FUnit_Container Create_Kitty();
}
```

C GLOSSARY

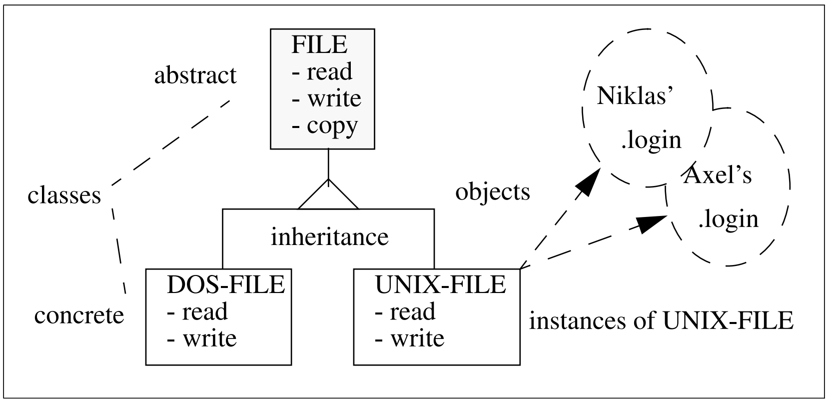


Figure 5.1 The relationships between abstract classes, concrete classes and objects.

Abstract Class An abstract class is a class without instances (objects). Objects are only created from subclasses to the abstract class. An abstract class is a template for its subclasses, see Figure 5.1.

Abstract Use Case See section 3.4.1.3.

Actor A use case concept, see section 3.4.1.3.

Architectural design In the architectural design activity the objects and their collaborations are defined.

Attribute The values of the attributes describe the state of the object. Also called member attribute.

Base Class A base class according to Taligent is a class which represent a logical object, see also mixin class [Tal94c].

Class A class is a template for a collection of objects that share the same data structure and support the same operations, see Figure 5.1.

Class Hierarchy A class hierarchy is a tree of classes related by inheritance. Commonalities between classes are extracted to common superclasses. For instance, a subclass should be a specialization of its superclass.

Class Library A class library contains several common classes, for strings, lists et cetera. An example of a class library are the Microsoft Foundation Classes.

Collaboration An object collaborates with another object if it has to invoke one or more of the other objects methods to fulfil its responsibilities [Karl95, p. 306].

Concrete Class A concrete class will provide the implementations of the abstract (virtual) operations of its abstract superclass, see Figure 5.1.

Concrete Use Case A use case concept, see section 3.4.1.3.

Coupling The coupling is used as a measure of dependency between classes. Either many different messages are passed, or one message is frequently passed.

Design pattern See section .

Detailed Design In the detailed design activity the classes and their methods and attributes are fully defined in the terms of the implementation languages.

Domain A domain is an application area for software products.

Dynamic Binding See section 2.1.2.

Extends A use case concept, see section 3.4.1.3.

Functional Requirement The functional requirements specify the services which should be provided by the system to the user.

Inheritance See section 2.1.2.

Non-functional Requirement The constraints imposed on the system which could not be categorized within the functional requirements is described in the non-functional requirements.

Method A method is the implementation of an operation in a class. Methods are also called member functions.

Mixin class A mixin class according to Taligent is a class that represents optional functionality. The concepts of *base-* and *mixin classes* support multiple inheritance. Taligent states that an optional number of mixin classes may be inherited but only one base class. The reason for inheriting a mixin class is to achieve its functionality. [Tal94c]

Multiple Inheritance Inheritance from more than one class is called multiple inheritance. Use of multiple inheritance may obscure the inheritance hierarchies.

Object An object is a data abstraction unit which encapsulates the associated operations. An object is said to be an instance of a class, see Figure 5.1.

Operation An operation on an object is used when an object shall be manipulated or extracted information from. Also called message.

Polymorphism See section 2.1.2.

Prototyping The development of an experimental version of some aspect of the system is referred to as prototyping [Mark94, p. 472].

Responsibility The responsibility of an object is defined as the knowledge it maintains and the actions it can perform [Wirfs90].

Stakeholder A stakeholder is a person or an organisation which have requirements or interest on the system. A typical stakeholder is the future user or the financier of the system.

Subclass Class B is a subclass of class A if B inherits from A, see Figure 5.1.

Superclass Class A is said to be a superclass of class B if B inherits from A, see Figure 5.1.

Use Case See section 3.4.1.3.

Virtual Operation A virtual operation is defined in a superclass but implemented in a subclass.

D References

[Boo94] Grady Booch. *Designing an Application Framework*. Dr. Dobb's Journal 19, No. 2, 1994.

[EST95a] *Reuse and object-oriented frameworks*, EST Frameworks, 1995

[EST95b] *State-of-the-art components Frameworks and Patterns*, EST AB, 1995.

[Gam94] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1994.

[Grif95] William G. Griffin. *Lessons learned in software reuse*. IEEE Software, July 1995

[Heni80] Heninger K.L., *Specifying software requirements for complex systems. New techniques and their applications.* IEEE Transactions on Software Engineering 6 (1), p. 2-13, 1980.

[Henr92] Mats Henricsson, Eric Nyqvist. *Programming in C++, Rules and Recommendations*. Ellemtel Telecommunication Systems Libraries, 1992.

[Jaco92] Ivar Jacobson et. al. *Object-Oriented Software Engineering, A Use Case Driven Approach*. Addison-Wesley, 1992.

[Joh88] Ralph E. Johnson, Brian Foote. *Designing Reusable Classes*. Journal of Object-Oriented Programming, June/July 1991.

[Joh91] Ralph E. Johnson, Vincent F. Russo. *Reusing Object-Oriented Designs*. University of Illinois tech. report UIUCDCS 91-1696, 1991.

[Joh92] Ralph E. Johnson. *Documenting frameworks using patterns*. OOPSLA '92 Proceedings, 1992.

[Joh95] Ralph E. Johnson. *How to develop frameworks*. Notes for OOPSLA '95, 1995.

[Karl95] Even-André Karlsson (editor). *Software Reuse, A Holistic Approach*, John Wiley & Sons, 1995

[Laj94] Richard Lajoie, Rudolf K. Keller. *Design and Reuse in Object-Oriented Frameworks: Patterns, Contracts and Motifs in Concert*. Proceedings of the 62nd Congress of the Association Canadienne Francaise pour l'Avancement des Sciences, Montreal, Canada, May 1994.

[Lar92] Johan Larsson. *Object-oriented frameworks*. REBOOT Consortium, 1992.

[Lien80] Lientz B.P and Swansson E.B. *Software Maintenance Management.* Reading MA: Adison-Wesley.

[Louc95] Loucopoulos P. and Karakostas V. *System Requirements Engineering*. Mc.Graw-Hill international series in Software Engineering. 1995.

[Marc94] J. J. Marciniak. *Encyclopedia of Software Engineering*, 1994

[Matt95] Michael Mattsson. *Draft for 3rd chapter of thesis*. August 1995.

[Mey88] Ware Meyers. *Interview with Wilma Osborne*. IEEE Software 5(3): 104-105, 1988.

[Mey94] Bertrand Meyer. *Reusable Software - The Base Object-Oriented Component Libraries*. Prentice Hall, 1994.

[Mil95] H. Mili, F. Mili, A. Mili. *Reusing software: Issues and research directions*. IEEE Transactions on Software Engineering, Vol. 21, No. 6, June 1995.

[NATO91a] Nato Communications and Information Systems Committee. *Software Reuse in NATO*, 1991.

[NATO91b] Contel corporation, *Standard for Software reuse procedures*, NATO contract number 5957-ADA, 1991.

[Ohl93] Lennart Ohlsson. *The next generations of OOD*. Object Magazine, May-June 1993.

[Rumb91] James Rumbaugh et al. *Object-Oriented Modelling and Design*, Prentice Hall. 1991.

[Som92] Ian Sommerville. *Software Engineering*. Addison-Wesley, 1992.

[Tal94a] *Building object-oriented frameworks*, Taligent, Inc., 1994

[Tal94b] *Leveraging object-oriented frameworks*, Taligent, Inc., 1994

[Tal94c] *Taligent's Guide To Designing Programs. Well-mannered Object-Oriented Design In C++*, Taligent, Inc., 1994

[Will91] H. Willars. *Amplification of Business Cognition through Modelling Techniques*. Proceedings of the 11th IEA Congress, Paris, July 1991.

[Wirfs90] R. J. Wirfs-Brock, R. E. Johnsson. *Surveying current research in object-oriented design*. Communication of the ACM, 33(9), pp. 104-124, September 1990.