# Integration of Design Patterns into Object-Oriented Design using Rational Rose

Diplomarbeit


Universität Rostock


Fachbereich Informatik


| | |
|---|---|
| vorgestellt von | Danko Mannhaupt |
| geboren | 1974-10-10 in Greifswald |
| Matrikel-Nr. | 094200997 |
| Betreuer | Prof. Dr.-Ing. habil. Peter Forbrig |
| Abgabedatum | 2000-05-31 |

## Zusammenfassung
Design Patterns haben sich in den letzten Jahren als große Hilfe bei der objektorientierten Softwareentwicklung erwiesen. Eine Unterstützung durch Werkzeuge ist aber leider noch in sehr geringem Umfang verfügbar. Aufbauend auf Untersuchungen zur Programmiersprache PaL, mit der ein methodisch neuer Ansatz zur Softwareentwicklung demonstriert wurde, wird ein Konzept zur Integration von Pattern in CASE-Werkzeuge, insbesondere Rational Rose, erarbeitet. Eine prototypische Implementierung weist die Tragfähigkeit des Konzeptes nach. Damit wird musterorientierter Softwareentwurf mit Rational Rose ermöglicht.

## Schlüsselwörter
Rational Rose, Entwurfsmuster

## Abstract
In recent years, design patterns have proven to successfully assist in object-oriented software engineering. Unfortunately, tool support is still very limited. A concept to integrate patterns with CASE tools, in particular Rational Rose, shall be developed that is based on investigations for the programming language PaL, which demonstrated a methodically new approach to software engineering. A prototype implementation demonstrates the feasibility of the concept. It enables Rational Rose to support pattern-oriented software design.

## Key Words
Rational Rose, design patterns

## CR-Classification
D.2.11 [Software Engineering]: Software Architectures — Patterns;
D.2.2 [Software Engineering]: Design Tools and Techniques —
Object-Oriented Design Methods, Computer-Aided Software Engineering (CASE);
D.2.13 [Software Engineering]: Reusable Software — Reuse Models;
K.6.3 [Management of Computing and Information Systems]:
Software Management — Software Development

## Remark
It is assumed that the reader is familiar with object-oriented concepts. Further information can be found in the reference list.

# Contents

# 1 Introduction

## 1.1 Motivation

Software continues to control an ever-increasing part of peoples' life, although it is not always perceived. The production of software is a very complex process with an unlimited number of influences. As for the production of any good, the objective of the producer is to make it as good, cheap and fast as possible to win the market and to yield a profit.

One of the most interesting results from the Total Quality Management (TQM) research [2, 11] is that in order to improve the quality of a product, one needs to document and adhere to a certain process. The sequence of steps and decisions are to be gathered and written down. Starting from a well-defined process that produces constant quality, process analysis and improvement can begin. This approach can also be applied to software engineering. Following a well-defined software production process stabilizes and enhances the quality of software products. The object-oriented approach, which consists of object-oriented analysis, design, programming languages, and other components is an improvement that has been introduced in research and commercial environments for a number of years. This thesis introduces another improvement to the software engineering process: the consistent and universal usage of design patterns during object-oriented design.

The current state of software engineering is often described with the term 'software crisis'. It refers to the fact that a large number of projects do not reach the final state, that is, they are not used. In contrast, projects are abandoned during the different phases or are not accepted by users and therefore, are not used. The goal of software development process models such as the Rational Unified Process [17, 18], OOTC or the V-Model [22] is to provide the developer or project manager with a template for the development. Successful application of these process models has increased the quality and acceptance of software products. Again, quality can only be the result of adherence to a well-defined process. The models mentioned above incorporate the knowledge and experiences from a large number of more or less successful software projects. Early involvement of the customer and users can greatly increase acceptance. Elements such as use cases become part of the development process and ensure that the focus remains on the solution of customers problems and the relief of users. This thesis does not discuss the advantages or disadvantages of either one of the process models. On the contrary, it introduces the application of design patterns, which can be integrated with each of the models.

Besides quality, costs and speed decide about success and failure of a software project. Companies in the software business always look for the competitive advantage, to improve their position on the market. A quickly executed project below the estimated price is the result every project manager is eager to achieve. Reuse has been the key to accomplish these targets.

Object-oriented technologies support reuse in different ways: Inheritance hierarchies in programming languages reuse class structures and method implementations. Object-orientated analysis enables the reuse of business and process

models, after they have been defined with use cases, class diagrams and other structured documentation. A good object-oriented design results in a balanced class diagram, which can be easily extended if the requirements of the current project change, and which can be used again with minor modifications for similar applications. It will be shown how best-practice knowledge in form of design patterns can further improve the reusability of object-oriented design results. Although the application of design patterns adds complexity to the development process, it can reduce the costs and increase the speed of project completion.

The development of complex software systems is normally a team effort. A number of people work together to specify, design, implement, test, and document a system. These kind of projects cannot be managed without the help of computers and software itself. Developers need a database that contains their knowledge, decisions and documents. Such a database is called repository. An integrated development environment supports the later stages of software development. Most often they are used for implementation, debugging, and tests. A general term for all tools that facilitate software development and communication between developers is Computer-Aided Software Engineering (CASE) tools. The functionality of these programs varies widely. Object-oriented CASE tools support the application of object-oriented technologies, for instance by providing the necessary diagrams. Examples include objectiF by microTOOL and Rose by Rational.

Both design patterns and CASE tools are becoming increasingly accepted and popular. This thesis will show how object-oriented CASE tools, Rational Rose in particular, can be enhanced to integrate an extended object-oriented design process, which is based on design patterns.

## 1.2 Outline

The thesis is divided into three parts: The first part deals with the object-oriented development process with regard to design patterns. After giving a short introduction to development phases in section 2 and object-oriented design in section 3, design patterns are presented in section 4. Their application, benefits, and costs are discussed. Section 5 explains the pattern model and the PaL pattern programming language that serve as a basis of this paper. Section 6 presents approaches to integrate patterns with object-oriented design.

Integration of design patterns into Rational Rose is covered in the second part of this thesis. Section 7 introduces two existing approaches and their limitations. Section 8 develops a concept to integrate design patterns with Rose based on the pattern model with patterns as model elements and system components. A prototype has been developed, which is presented in section 9. Screenshots illustrate its use.

Finally, the third part provides suggestions to extend the prototype that can be seized by other developers or Rose engineers. Conclusions finish the thesis. A glossary with several abbreviations, literature references, and a list of figures can be found in the appendix.

# Part I

# Object-Oriented Design with Design Patterns

The first part of this thesis introduces object-oriented design and design patterns, explains a new approach to programming with design patterns, and illustrates the construction of complex software systems with patterns as basic components.

## 2 Development Phases

The most common and well-known model for the software development process is the waterfall model. It assumes that a number of phases follow one after the other. The output documents from each phase are the input for the next step, hence the name waterfall. There is little or no interaction between the phases. The number and names of phases vary, the following should serve as an example:

1. Feasibility Study
2. Requirements Specification (Analysis)
3. Design
4. Implementation
5. Test
6. Delivery

Object-oriented methods do not commonly use this strict phase model. In contrast, iterative, incremental models are used, for example Boehm's Spiral Model [4] or others, which are more oriented towards Rapid Prototyping. However, software development requires a number of tasks to be executed, no matter which process model is used.

Object-oriented application development can follow a use case driven approach [12, 23]. Use cases are developed systematically at the beginning of a project. They form the basis for further development. Use cases are applied to capture and document requirements. They describe processes in the application domain from a client's perspective. Every use case is accompanied with a scenario description.

The use case driven approach enables the partition of the complex system into smaller components, based on one or more use cases. These subsystems can then be developed by teams. The results are synchronized and reviewed regularly. The development process is divided into iterations. Every iteration

consists of a planning phase and an analysis-design-implementation cycle. The functionality of the system incrementally grows with every step.

During analysis, a system is modeled in an ideal environment [3], whereas the design realizes the specified application on a platform with required technical parameters. Analysis determines and describes system requirements. Object-oriented analysis (OOA) is concerned with real-world objects, which can be tangible objects or persons as well as events or processes in the application domain. A real world object becomes part of the object model by abstraction. Objects with similar properties and behavior belong to the same class.

Often, the analysis model is separated into two parts: static and dynamic model. The static model describes the classes of the system, associations, and inheritance relationships. Packages partition the system. The dynamic model contains business processes, scenarios of object collaboration, and state machines, which define an object's life cycle. A user interface might be required during the analysis phase for a discussion with the client.

# 3 Object-Oriented Design

The object model that was constructed during OOA does not contain information about user interface presentation of objects, about storage, distribution, or performance optimization. It models *what* the system should perform. The design on the other hand is concerned with all the issues mentioned before. It defines *how* the performance will be achieved.

The fact that object-oriented analysis and object-oriented design (OOD) utilize the same set of techniques simplifies the transition and interaction between both phases. The Unified Modeling Language (UML) [30] represents the key: It defines syntax and semantics of a variety of models and diagrams. Examples include use case models, class diagrams, interaction diagrams or state charts. Through the use of a common communication language between developers, the development process became easier, more efficient, and less error-prone. UML diagrams are used throughout the complete development cycle of object-oriented systems.

OOD [5, 23] can be divided into 2 major phases: architecture or system design, and implementation design. Design of system architecture refers to the separation between business logic, user interface, and data storage. The objective is to separate these layers in order to preserve flexibility. It must be possible to update or exchange one of the system layers with only a limited and small number of changes on other layers. For instance, it might be necessary to exchange the database management system to utilize advanced functionality or increased performance. Required changes on the system must be confined to the data access layer. For another system, the client might demand a web interface, where the original implementation only supported a traditional workstation interface. With the user interface layer separated from business and interaction logic, it should be relatively easy to implement another system interface. Implementation design refers to the refinement and adaptation to a particular environment and programming language. Properties of the target software and hardware architecture are considered.

Models developed during OOA are refined in the design phase. The OOA object model becomes a draft for the business logic layer. It is revised for efficiency and reuse. Existing frameworks, class libraries, and interfaces are considered. It will be explained later how design patterns are used at this stage. The user interface prototype is systematically advanced and becomes the foundation for the user interface layer. Furthermore, system components for data access are developed, and software and service distribution is discussed.

In comparison to OOA, the differences between the OOD model and the future program are reduced. Every class, every attribute and operation of the model will be part of the program. The separation between static and dynamic model is continued. The static model contains the complete class model of the application and the partition into packages and subsystems, whereas the dynamic model describes the complex communication between objects.

A more detailed description of object-oriented design principles and guidelines can be found in the literature [5, 23]. The next section introduces the concept of design patterns.

# 4 Design Patterns

## 4.1 A Software Engineering Trend

Design patterns were one of the major recent trends in both the scientific and commercial software development community. Countless journal articles and a number of books were published. An entire conference series deals with design patterns — the annual Pattern Languages of Programming (PLoP) conferences [10]. This section gives an introduction and explains the benefits of design patterns.

Architect Christopher Alexander defines the term *pattern* as follows:

> Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution. [1]

He refers to the architectural domain, but patterns are also found in software architecture. Experts know patterns from practical experience and follow them in developing applications with specific properties. They are used to solve design problems both effectively and elegantly [9].

## 4.2 A Pattern For Patterns

Gamma et al., also known as the Gang of Four (GoF), defined in [14] the structure for a pattern description. This pattern for patterns has won recognition and is used with minor modification in different pattern catalogues. According to Gamma, these elements are used to describe a pattern:

- A *distinct name* that concisely defines the design problem, its solution, and impacts. The name simplifies communication between developers, and design discussions and documentation of design decisions.
- *Context and problem specification.* Context refers to the general situation in which the problem occurs. The problem description explains specific design problems. It can also contain conditions that must be met before a pattern application is reasonable. The general problem statement can be completed by a set of *forces*. Forces can be requirements the solution must fulfil, constraints that must be considered, and desirable properties the solution should have. Balancing the forces is a key to the problem solution.
- The *solution.* It describes the elements that form the design solution as well as their relationships, responsibilities, and interactions. In contrast to algorithms or other programming guidelines, design patterns do not contain a specific implementation, except as an example. Patterns are like templates that can be applied in a number of situations. A class diagram is often used to visualize the solution. However, the diagram itself cannot explain a pattern. Some patterns have the same class diagram, but the semantics, the meaning, is different. In addition to textual descriptions

and class or object diagrams, interaction or collaboration diagrams are used to define dynamic behavior of pattern members. Thus, interaction diagrams formalize the pattern documentation.

- The *consequences* section explains the effects of pattern application. Advantages and disadvantages of the design are provided. Understanding and documenting the consequences of design decisions objectifies system design. These decisions influence performance, efficiency, reusability, flexibility, extensibility and portability of the system.

- Finally, *examples* in particular programming languages and domains usually accompany pattern descriptions. Understanding and application of the pattern by a developer is simplified.

According to Gamma et al., design patterns are descriptions of cooperating objects and classes that are tailored to solve a general problem in a specific context. They are distinguished from algorithms and other concepts that can be reused as single classes on one side, and frameworks or subsystems and class libraries for application specific domains on the other side.

Class libraries are collections of classes, which the developer is able to use, i.e. define objects of these classes, run operations, and define subclasses. Simple class libraries support code reuse and can be seen as object-oriented functional libraries. Contrastingly, frameworks consist of a number of collaborating classes that implement a reusable design for a particular application domain. They contain concrete and abstract classes that define interfaces. A framework user typically creates subclasses that later receive messages from the framework. Frameworks allow design reuse. All applications developed with the same framework have a similar structure and are easier to maintain.

In comparison to frameworks, design patterns are more abstract. Application of a design pattern always requires a new implementation. Design patterns are smaller than frameworks. In fact, frameworks typically contain a number of patterns. In contrast to frameworks, design patterns are not limited to a single application domain.

## 4.3 Pattern Catalogues

The number of patterns discovered increases with every publication. The best-known catalogues of design patterns include books by Gamma et al. [14], Buschmann et al. [9] as well as the series 'Pattern Languages of Program Design', which started with [10] and was recently supplemented by a fourth book [16].

Design patterns can be classified according to their tasks as creational, structural, and behavioral patterns. Furthermore, class and object based patterns are distinguished. Class based patterns contain relations between classes. They are expressed with inheritance relationships. Object based patterns consist of relations between objects that can be changed during their life time. Creational patterns support the independence of a system from the way its objects are

created and composed. Examples from [14] include object based *Abstract Factory* and *Builder*, and class based *Factory Method*. Structural patterns deal with the construction of larger structures out of classes and objects. Object based structural patterns such as *Composite* and *Decorator* merge objects for additional functionality. They provide more flexibility than class based patterns because the object structure can change at run time. Behavioral patterns define interaction between objects and classes. Complex control flows are described. Examples include *Observer*, *Visitor* and *Iterator*.

Unlike design patterns, analysis patterns (see [13]) are used to capture typical class configurations during analysis. As explained earlier, this means that analysis patterns are not concerned with performance or reusability aspects, but are used to model reoccurring situations in domain analysis. More general examples include *List*, *Role* and *Group*, specific analysis patterns in the finance domain are for instance *Transaction* and *Portfolio*. Analysis pattern catalogues normally have a structure similar to those of design patterns.

## 4.4 Pattern Application

Developers can benefit from the use of design patterns. To exploit the advantages, they either need to be familiar with pattern collections including consequences of application, or they are required to have access to a pattern catalogue in form of a book or a database. Ideally, a developer meets both conditions, having a general understanding of patterns and an overview of examples, as well as a catalogue at hand, which can be used to look up further information.

A scenario for design pattern application will be given next. Say, a developer has a preliminary class model. A shortcoming is recognized in the model, for instance the lack of extensibility or flexibility if requirements change. The developer would then recall the pattern knowledge and study the catalogue to find a solution. Using the examples and solution, the model could be adjusted to reflect the pattern recommendations. This usually adds classes, attributes or methods to the model. The decision to apply a design pattern must be documented. This can be achieved by naming classes and properties accordingly and adding a note and a reference to the model documentation containing the reasons that have lead to the decision.

## 4.5 Benefits of Pattern Usage

The positive effects of design pattern application are not only predicted by authors of articles and experienced by software engineers in their projects. An experiment was conducted to capture the influence [24]. Its intention was to study the real influence of using design patterns and to prove or disprove several theses and the conditions under which they are true. Only the main conclusions of the article are given here. The authors used controlled experiments for scientific study of the theses. Both students and professionals participated in the investigation. Practical considerations suggested to study maintenance tasks.

First, it was detected that improved communication by design pattern doc-

umentation increases productivity and/or quality, depending on the situation. Second, the use of a design pattern instead of a simpler alternative solution can be either beneficial (increased flexibility with same or smaller costs) or disadvantageous. Generally, good common sense of software engineering is a sound indicator for the usefulness of patterns in comparison to alternatives. The results of the experiments recommend accurate documentation of design pattern usage in software because it distinctly decreases the maintenance efforts.

## 4.6 Disadvantages, Costs, and Alternatives

Design patterns can assist developers, but there are costs associated with them. Patterns add complexity to development. Successful application requires profound comprehension of pattern fundamentals and knowledge of pattern catalogues. Developers must be educated to acquire this knowledge. Moreover, equal education among team members is necessary. Otherwise, developers do not understand design decisions and architecture of systems.

To evade these additional costs and problems, a development team or project management might choose to ignore design patterns completely. This represents a short-sighted solution, because patterns do provide advantages, particularly in the later stages of a software's life cycle, during maintenance. Alternatively, frameworks with integrated patterns can be used. Thus, pattern application is simplified with the provision of predefined extension interfaces. To conclude, application of design patterns is an investment in software development, which is associated with startup cost but which will yield a profit, eventually producing products of increased quality.

# 5 Design Pattern-Oriented Programming

After giving an introduction to design patterns, this section presents a concept of programming with design patterns. The PaL approach, which forms the foundation for the main part of this thesis, will be explained.

## 5.1 Imperfections with Design Pattern Implementation

Several aspects triggered the development of a design pattern-oriented programming model. First, design patterns are not identifiable in a program listing. The designer chooses to apply a pattern in a certain situation. Classes and operations are added, names are changed. Already in design stage, without proper documentation it becomes hard to identify patterns in the finished design. However, for maintenance reasons it is vital to know in which place a pattern was used. Otherwise, development cannot benefit from the flexibility and other advantages of the pattern. The same consideration applies to design patterns in implementation stage. In order to correctly apply and maintain design patterns in source code, suitable documentation is required. However, there is another possibility: Design patterns become components of the program as structuring elements, larger than classes and objects and smaller than modules or subsystems. This approach to a design pattern-oriented programming model has been jointly developed by Normen Seemann and Stefan Bünnig [6, 7, 29].

The conventional approach of design pattern application in an object-oriented programming language would be to represent the pattern components as classes and objects of the language. Relations between components are modeled as relations between the respective classes. Interactions and collaborations are implicitly contained in operations. Besides the already mentioned problem of identifiability, these other issues emerge: [29]

- A design pattern can be seen as an encapsulated unit, which separates its components from the outside. Object-oriented programs generally contain a number of design patterns as well as other classes. They are all treated in the same way, a separation is impossible. In addition to this static encapsulation problem, there is a dynamic encapsulation problem, which refers to the fact that the objects that were created as instances of design patterns are not separated from other objects. A design pattern can be applied many times. The resulting objects are associated with the pattern rather than with conventional classes.

- A design pattern is an abstract concept that cannot be used in a program without modifications. It has a general component structure and it is not confined to a single application. Therefore, before a design pattern is applied, refinement takes place, which includes renaming of components, properties and operations, and implementation of operations. This refinement process cannot be expressed with object-oriented techniques, because relations to other components are broken. In the result, a design pattern cannot be reused in the same way a class can be reused. A developer is forced to re-implement the design pattern again and again.

- The refinement should be configurable. The developer is in control of the process, the granularity and the sequence of steps.

## 5.2 The Pattern Model

To meet these requirements, [29] develops a pattern model. A design pattern is the fundamental element of the model. It is a static element that can be instantiated at runtime, is referenced with an entity, modified by functions and can be deleted, just like objects of a class. A pattern becomes a programming language construct, similar to a class. The definition must contain the following elements:

1. *Components*, which form the parts of a pattern. They are comparable to classes, have properties and attributes. Component instances are associated with one particular pattern instance.

2. *Attributes*, which can be classified as internal and external attributes.

3. *Methods* implement the higher behavior of a design pattern. They can also be either internal or external. The higher behavior of a design pattern describes global integrity conditions and the pattern's reaction within the context of other design patterns or classes.

A program in this model is a system of design patterns with the form described above. Execution starts with the instantiation of a specified top design pattern (the application pattern) and the following execution of a designated main method of that pattern.

**Tiling Patterns** Before the concepts of refinement and combination in the pattern model will be explained, another approach shall be mentioned: In his article [20], D. Lorenz explains a way to *tile* design patterns. He uses patterns to describe the implementation of other patterns. It is understood that the issue of pattern relationships is one of the difficult aspects of patterns. In [14], patterns are classified according to purpose and scope. Furthermore, a diagram shows relationships between patterns. However, a pattern can simultaneously be (or share large portions with) an implementation of several other patterns.

Patterns can be put together, just like tiles. It can be shown how both *Interpreter* and *Visitor* pattern can be tiled together with other respective patterns to form a reflective *Interpreter* and a *Visitor* mosaic. Thus, patterns are program components, which can be put together for additional functionality. Furthermore, an additional degree of reuse can be gained by allowing inheritance, that is, letting visitors refine or extend other visitors. Meta-visitors and meta-interpreters can generate visitors, which in turn serve as building blocks in constructing more complex visitors. To conclude, some patterns or tiles fit nicely together and may always need to work together, such as *Interpreter* and *Visitor*. Others have the same shape but different purposes. A set of mini-patterns form the basis that is used to construct others.

**The Pattern Model** The pattern model treats design patterns as components of a program, similar to classes in conventional object-oriented programming languages. Among other concepts like encapsulation and delegation, reusability is achieved with the notion of class inheritance. Subclasses inherit structure and behavior of their ancestors by inheriting properties and operations from superclasses. Furthermore, functionality of classes can be combined with multiple inheritance. Thus, a class inherits properties and operations from more than one superclass. The pattern model supports the application of similar manipulations to design patterns – *Refinement* and *Combination*.

**Refinement** Refinement reuses and specializes a design pattern. For example, an abstract *List* can be refined to the more concrete *StringList*, which deals with string items. [29] defines the refinement relationship between source and refined design patterns: Each component of the source pattern is injectively refined to a component of the refined design pattern using inheritance. That means, refined components inherit properties and operations from source components. The component structure of the source pattern is preserved. The internal part of the refined pattern inherits from the internal part of the source pattern. The external part of the refined pattern subtypes the external part of the source pattern. During refinement, components, their properties and methods can be renamed. Generally, refined patterns add functionality by implementing existing and additional methods. Static and dynamic relationships between components remain effective after refinement.

**Combination** Combination is the pattern model's equivalent to multiple inheritance in the object model. A number of source design pattern are combined to compose a more complex pattern, which unites functionalities. In this process, key components from the source patterns are joined to one single component in the combined pattern. A new name must be selected for the merged component. Furthermore, attributes and operations are consolidated, that is, either selected, renamed, refined, re-implemented or joined with other attributes or operations of the respective source components. A combination example can be found in figure 8 later in this paper.

As a result, there are three ways to create a new design pattern: from scratch, as a refinement of another design pattern, or as a combination of a number of patterns.

## 5.3 The Programming Language PaL

To prove the concept of a design pattern-oriented programming model, a programming language that supports the model has been designed cooperatively by [6] and [29]. For comprehension of the design pattern support in CASE tools, a short introduction to one of this languages is helpful. The PaL (Pattern Language) language from [6] has actually been implemented. With the help of the Language Development Laboratory (LDL), the syntax was captured and the semantics defined in form of transformation steps into source code of the Eiffel programming language. The result is a Prolog program that is able to transform

a PaL source program into an Eiffel source file. To achieve this, PaL is closely related to Eiffel and adds constructs to the language to define and manipulate patterns. Before the syntax is presented, two other issues are discussed.

The pattern model can deal with arbitrary nesting depth, i.e. design patterns can contain components, components can be patterns that contain other components, and so on. However, for practical programming reasons, nesting was confined to three levels, which are also visibility levels: a global level, the level of all design patterns, and the level of all components inside design patterns.

The pattern model provides two ways to use a design pattern: reuse by refinement and combination, and instantiation. However, in PaL, there is no separation between patterns and application of patterns. Patterns are refined and combined until a pattern represents its application.

As an example, the simple design pattern *List* is shown. A refinement step is demonstrated with the definition of the well-known pattern *Composite* based on the *List*. The example is taken from [6].

```
pattern PLIST

  creation make

  component LIST
    creation make
    feature make is ...
    feature add(an_item:  ITEM) is ...
    feature delete is ...
    ...
  end -- component LIST

  component ITEM
    creation make
    feature make is ...
    feature next:  ITEM is ...
    feature set_next(an_item:  ITEM) is ...
  end -- component ITEM

  extern feature make is
    do
      !!the_list.make
    end -- make

  intern feature the_list:  LIST

end -- pattern PLIST
```

Figure 1: PaL Source for *List*

The primitive design pattern *List* (actually `PLIST`, see figure 1) consists of 2 components: `Item` and `List`. The properties and operations of those components are self-explaining. The internal pattern attribute `the_list` contains a

reference to an instance of a `List` component. It is initialized by the external creation method `make`.

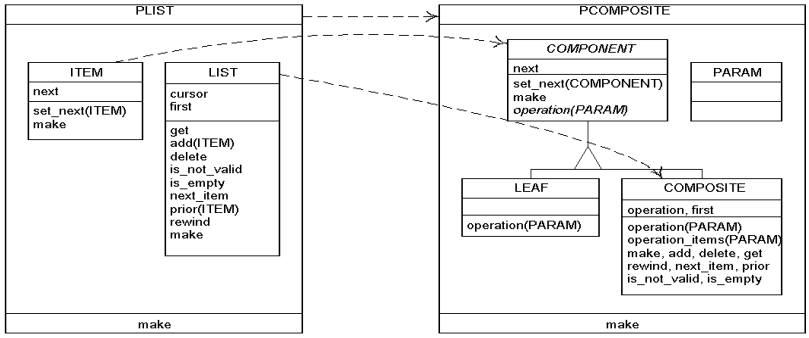


Figure 2: Refinement from *List* to *Composite*

The design pattern *Composite* refines the *List* (see figures 2 and 3). The components in *Composite* are `Component`, `Composite`, `Leaf`, and `Param`. The latter encapsulates the parameters used in component operations. Such a component will be refined in a concrete application to contain the required parameters. `Composite` contains a list of components. Consequently, `Item` is refined to `Component`, `List` becomes `Composite`.

In [8], it is shown that an entire application (*DrawIt*) can be constructed based on design patterns. This was achieved by implementing 3 basic design patterns (*Container*, *List* and *Parameter*). They were used to build a library of patterns based on [14]. All 23 patterns from the book are implemented. Some patterns are combined, because these combinations, for instance *List* and *Iterator*, are used frequently. Finally, the functionality of the drawing program has been implemented using the patterns. The applications itself is also a design pattern, which starts the program.

The developers of the design pattern-oriented programming model see a number of extensions to their model and its implementation in PaL: patterns as components inside other patterns are a reasonable extension, which creates more opportunities for pattern design. Furthermore, refinement and combination can in certain circumstances create redundant inheritance relationships. Those redundancies should be removed.

The authors conclude that the design process must be adapted to support the extended possibilities of the model. In order to effectively utilize the benefits of pattern-oriented programming, a software engineer must familiarize himself with the pattern libraries available. The model adds more complexity to the process than, for instance, the usage of APIs. However, application of the pattern-oriented model increases reusability of software components and improves maintainability and flexibility of the system implementation.

```
pattern PCOMPOSITE

  refine
    PLIST
      rename
        LIST as COMPOSITE,
        ITEM as COMPONENT,
        the_list as the_compos
    end

  creation make

  component COMPONENT
    feature operation(a_parameter:  PARAM) is
      deferred
      end
  end -- component COMPONENT
  component LEAF
    inherit COMPONENT
    creation make
  end -- component LEAF

  component COMPOSITE
    inherit COMPONENT
    feature operation(a_parameter:  PARAM) is ...
    feature operation_items(a_parameter:  PARAM) is ...
  end -- component COMPOSITE

  component PARAM
  end -- component PARAM

  extern feature make is
    do
      !!the_compos.make
    end -- make
end -- pattern PCOMPOSITE
```

Figure 3: PaL Source for *Composite*

# 6 Design Patterns in Object-Oriented Design

This section presents an approach to use design patterns during object-oriented system design. The identification and application of patterns will be discussed. Furthermore, it will be shown how they can be used as building blocks to construct complex software systems while increasing quality and reusability of the product.

## 6.1 Creation of Design Pattern Description

As written in [26], design patterns are not invented but discovered. The creation of a pattern description represents a challenge for the designer. The following phases can be identified in the process of pattern description development:

**Pattern Mining** The developer discovers a recurring problem that has been solved repeatedly. The essence of the recurring solution has to be characterized and put into words.

**Pattern Writing** The designer chooses a suitable pattern format and writes a pattern description including situations when the pattern can be applied and properties of the solution.

**Shepherding** Another person with the role of a shepherd reviews the pattern description with the author in form and content. Generally, multiple revisions are required. A recommendation is given pro or against review in a writers' workshop.

**Writers' Workshop** A group of equal authors reviews the description. Members underline good elements and criticize form and writing performance.

**Author Review** The author reviews the description according to recommendations. The revision can be either reviewed in another workshop or published.

**Pattern Repository** Respected publishers keep pattern descriptions in online archives. A selection of them is used to compile future pattern books.

**Anonymous Peer Review** Final design pattern review is anonymous and focussed on technical content.

**Pattern Book Publication** Patterns are published in books such as the PLoPD series (eg. [10, 16]) or others. Thus, developers study pattern catalogues and apply patterns to their systems.

## 6.2 Application of Design Patterns

In section 2, the software engineering process for object-oriented systems was explained. The results of object oriented analysis were a static model, for instance in the form of a class diagram for relevant business or other core classes, and a dynamic model, which could be described by object interactions and state machines.

The design of a software system adds further complexity. In addition to business classes, the system is extended with a user interface, interactions facilities, data management and other properties that are required for every real-world software system. Flexibility and quality are of paramount importance in system design. Specific functionality, components and complete subsystems must be kept separated from each other to enable their exchange in the case of updated requirements, performance demand or to integrate third-party solutions. This represents one of many reasons to utilize design patterns during object-oriented system design.

Based on results from analysis, design refines the system specification. In particular, pattern oriented design requires a number of steps to be executed (compare also [27]). As there is always interaction between analysis and design, the following steps cannot clearly be associated with either phase.

**Identify Classes** Requirements specification and analysis provide a class model with suitable class names.

**Identify Responsibilities and Interacting Classes** Every class provides a set of services. These class responsibilities form the basis for interactions between classes or their respective objects. The dynamic model contains this information. It can be represented with textual description or, more formal, more visual, and more detailed: in form of interaction and sequence diagrams.

**Identify Class Groups and Interactions** Based on class interactions, classes can be grouped into collections. These collections have different sizes, one class is typically participant in a number of class interactions and, therefore, class groups. For every class group, the interactions within the group as well as the characteristics of the group are to be identified.

**Abstract Group Interactions and Pattern Identification** The group interactions and characteristics from the last step are to be restated at an abstract level. Abstract interaction descriptions are derived, which do not refer to concrete class names but represent the nature of the object and/or class interaction. These descriptions can be compared with existing design patterns. The developer needs to identify the patterns that match the interactions and characteristics. To support this process, profound knowledge of available patterns is required. Furthermore, the use of an efficient retrieval tool that contains descriptions and examples of a library of design patterns is recommended. With

help of keyword and full-text search, the developer is assisted in search for a pattern that reflects the interactions and responsibilities.

**Pattern Application and Rough Design** For every pattern identified, the developer has to introduce new classes or remove existing classes from the class diagram, based on the pattern structure. Names for the pattern components are to be found, roles are identified. It might be advisable not to apply the pattern in its original structure, but to reduce flexibility by changing or removing components. These alterations need to be well documented to restore original functionality if required. In case there exist more than one pattern which affects the same class, one should try to integrate both patterns. If patterns cannot coexist, a selection between different approaches to solve a problem will have to be made. The developer has to value each solution and decide according to priorities. In [27], it is suggested to calculate the tradeoff for every rough design. This can be done by calculating a quantitative measure of four characteristics, namely coding effort, static and dynamic adaptability and performance, and adding the values for each design solution. Based on the results, one of the designs is chosen.

**Detailed Design** After deciding which pattern to apply, attributes and operations from pattern classes as well as relationships between classes are added. The rough design is refined and, eventually, includes all required properties from the design pattern.

**Application of Patterns to Meet Requirements** The approach that has been presented presumes that the developer has already designed a detailed class model that can then be extended to reflect design pattern structures. The situation will be different if the developer has only a vague vision of the system to be created. However, the requirements specification might demand certain properties, such as exchangeability of algorithms or the separation of data model and representation.

Consequently, the developer will search the pattern catalogue for a solution that can be applied in this particular situation. As it was explained before, retrieval tools will support this process. As a result, a selection of patterns is found that implements the required properties. In the example mentioned above, this could be the patterns *Strategy* or *Model-View-Controller*. The next step is to build the structure of the class model according to the pattern.

Instead of applying the pattern to supplement the analysis class model, system classes are arranged to bring a pattern to life. However, such distinct separation can hardly be found in reality: The developer combines business classes and a design pattern: Both are rearranged to fulfill the requirements.

**Granularity** The pattern application in object-oriented design as it is described here can be applied at any level. On one side, architecture design can be improved with design patterns. As for architecture design, specific requirements are generally the starting point that demand a particular system struc-

ture. Architectural patterns such as *Model-View-Controller* are applied at this stage. Interaction participants are larger entities such as subsystems or components. The abstract pattern is refined and adapted to the particular situation in which it is used. The process of refinement in pattern-oriented design will be explained in the next subsections. On the other side, detailed implementation design is supplemented with design patterns. Interactions are typically modeled between classes. Refinement and combination of patterns represent the methods of pattern application.

**Static and Dynamic Pattern Aspects** Object-oriented design with patterns so far has mainly dealt with the static model. Class diagrams, attributes, operations and relationships were discussed. However, design patterns cannot be reduced to their class diagram. For many patterns, dynamic behavior creates their advantage over conventional solutions. There are reasons for the focus on static aspects: First, graphic representation of static aspects is easier for a designer. Drawing a class diagram requires less effort than the layout of an interaction diagram. Second, and more important: The information from a class diagram can be transformed almost automatically into source code of a programming language.

In contrast, interactions modeled in a sequence or interaction diagram document design solutions, but they cannot be directly transferred to source code. However, as it is pointed out in [19]: Although the creation of interaction diagrams is one of the most time-consuming (and worthwhile) steps, the "assignment of responsibilities and development of interaction diagrams is the most significant creative step during the design phase".

To summarize, dynamic behavior is an important part of a pattern. Thus, pattern application means that static and dynamic aspects of the system change. These changes can be represented with diagrams, but the actual implementation profits only indirectly from it. The programmer uses diagram information to implement class behavior. The second part of this thesis also demonstrates how dynamic aspects of design patterns can be represented with CASE tools.

## 6.3 Patterns as System Components

After explaining the general usage of design patterns during object-oriented analysis, this section introduces a design concept with patterns as components that are combined to construct complex software systems. First, two approaches by other authors are presented[1]. Following, an approach based on the pattern model for programming languages (see section 5.2) is introduced.

### 6.3.1 Explicit Representation of Design Patterns in *FACE*

Meijler et al. present in [21] a model of pattern development and application called *FACE*. The name stands for *F*ramework *A*daptive *C*omposition *E*nvironment. It intends to bridge the gap between high abstraction level design and lower level implementation, which emerges when automated source code generation from a design pattern definition conflicts with "hand-made" changes. This should be achieved by supporting incremental development using frameworks at the abstraction level of design patterns. The pattern framework development is clearly distinct from application modeling and development. Implementation will be hidden, modeling is just a matter of defining the roles and relationships of classes in pattern-specific terms. For example, in the abstract factory pattern, a factory class must be specialized by specifying its creation operations and specifying which creation operation instantiates which product class.

**Schemas** The application developer composes a model called *schema*. It consists of classes and their roles in the pattern, operations and the roles they play in the pattern, relationships between classes and/or operations, and parameters. When creating such a schema, the application developer is said to *instantiate* the pattern. The *primal schema* consists of a basic set of abstract classes and their relationships. It is cloned and becomes the *kernel* of the instantiation, which is then extended by defining concrete classes with roles and operations, creating corresponding relationships and specifying necessary parameters. Patterns can be seen as "mini-frameworks", which do not have a standard implementation, but must always be adapted to the specific usage.

The concept of a FACE schema is related to the concept of a class-diagram in an object-oriented modeling technique. A graphical notation similar to OMT [28] is used. Parts of a schema are called components. In a pattern instantiation diagram, there is a separate component that indicates, which pattern has been used. It contains references to the most important abstract classes of the pattern. These classes are called class-components. Major operations and relationships with a context-specific runtime meaning are transformed into explicit components of the schema. Class-components are typed corresponding to the role they play in the pattern. The schema only makes aspects explicit that are relevant, others are not shown. Internal structure is hidden. The definition of the meta-schema of a pattern is a separate process that will in general be

[1] Although the approach in this paper has been developed independently from [21] and [31], the author acknowledges their earlier work

executed by a different developer. Every class-component in the primal schema is an instance of a metaclass-component in the meta-schema. This is also true for all schema components.

An interesting aspect is how the parameterization and linking of class-components and relationship components may lead to the corresponding runtime behavior of the objects. This is achieved with an interpreted approach where the schema at runtime is represented explicitly with objects that stand for the class-components and relationships. Class-components are objects but they function as classes, in the sense that they can be requested for instances. Instance objects will query the objects in the schema to adapt their behavior to the parameters.

To summarize, FACE introduces a model for pattern meta-schemas. These mini-frameworks are developed by experienced designers and can be applied by application developers. Patterns are thus seen as separate components, that become parts of the system. The authors of [21] agree that visual development support is required to efficiently develop patterns and support their application.

### 6.3.2 Pattern Oriented Frameworks

Yacoub and Ammar present in [31] an approach for constructing object-oriented design frameworks using design patterns as building blocks. Architecture of frameworks is expressed with pattern diagrams. Furthermore, the authors propose a development process that expresses the framework in two design levels — a pattern diagram and a class diagram. It is shown how frameworks are instantiated.

Pattern-oriented frameworks are white-box frameworks. The framework user has to understand the design as interacting patterns (pattern diagrams) to instantiate the framework in the application. Application specific parts will be implemented by the framework user in a particular application.

**Diagrams** Pattern diagrams are used to describe the framework in terms of subsystems, design patterns and associations. In addition to well-known UML elements such as classes and associations, these new elements are introduced: Internal subsystems refer to independent groups of patterns that collaborate to fulfill a set of responsibilities in the framework. In contrast, external subsystems are not part of the framework. Pattern associations (dependencies) are relationships between patterns. At a later stage, they are refined by translation into class associations between two classes of communicating patterns. Subsystem relationships are general dependencies between subsystems. Finally, design patterns are represented as rectangular boxes with pattern name and type. Type refers to the name of a known documented pattern that is applied, e.g. "`SearchStrategy : Strategy`".

**Development Process** The framework development process starts with system analysis. Subsystems are identified and the requirements of each subsystems are studied. Candidate design patterns are chosen. Each subsystem is

represented in the form of design patterns and associations — as a pattern diagram. The framework is built by gluing design patterns. The top-down design approach proceeds with pattern instantiation. Patterns in each subsystem are expressed in terms of their collaborating classes. Classes are renamed to have meaningful names in the particular context. Next, reduction eliminates replicated abstract classes that have appeared because the same pattern type has been used in more than one subsystem. The grouping step merges concrete classes together depending on their interaction and responsibilities.

Framework instantiation is distinguished between pattern-level and class-level instantiation. Using pattern-level instantiation, a framework user instantiates individual patterns into classes and carries out reduction and grouping steps. Users need to map high-level associations between patterns into class associations. Alternatively, with class-level instantiation, the framework users instantiate the reduced class diagram.

To conclude, the pattern level represents a design layer higher than class diagrams. Generic frameworks constructed based on patterns as components can be used to design applications. Architecture designs based on patterns can be reused since they provide a high level of abstraction. Thus, development time and effort is reduced. Formalization of the visual presentation for patterns and their interfaces is required, as well as tool support for pattern diagrams. The authors of [31] imagine a tool that integrates pattern diagrams with a class diagram view to instantiate patterns and to facilitate the reduction and grouping process.

The approach in this thesis is closely related to the pattern oriented frameworks described before. The concept of patterns as components to construct complex systems is also pursued here. Furthermore, the elements used and their graphical notation in diagrams represent a reasonable starting point for further refinement. However, the development model in this paper differs: It does not distinguish between a framework (pattern) level and an application (class) level. Transition from an abstract pattern taken from a pattern book to its application in a concrete application can and, commonly, will be executed in a number of steps. That is, multiple iterations of the refinement/application process are supported. Thus, a pattern can be refined and combined several times before the final pattern is constructed. Details are explained in the next section.

### 6.3.3 Patterns as Model Elements

Previous sections have already introduced the notion of design patterns as components. This section will summarize their representation and properties.

**UML Notation for Design Patterns** Effective usage of design pattern requires graphical representation and manipulation tools. UML has become the standard language for software engineering. Therefore, it is unsurprising that it also provides model elements to visualize design patterns [30]. It is suggested to use an element *Collaboration*. Collaboration refers to complex behavior that can only emerge if elements cooperate. It is represented in a class diagram with

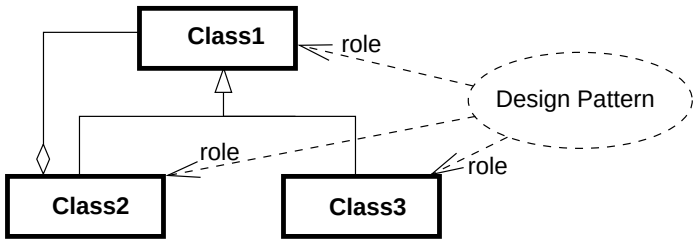


Figure 4: Collaboration/Design Patterns-Notation in UML

a dotted ellipse that contains the name of the pattern. Arrows point from the ellipse to classes that participate as components. They are supplemented with role names. Figure 4 shows an example taken from [23].

UML documentation also correctly notes that design patterns involve many nonstructural aspects, such as heuristics for their use and lists of advantages and disadvantages. Such aspects are not modeled by UML and may be represented as text or tables. Moreover, the bottom-up approach used in the UML notation guide does not show a pattern as a separate design component.

**Static Elements** The following elements of the static model must be displayed in a design pattern diagram:

- Design Pattern – container with name
- Pattern Interface – public pattern features
- Components – participating classes
- Association and Inheritance – relationships between pattern components
- Pattern Associations – relationships (dependencies) between patterns

The diagrams developed in [6] (figure 2) provide these features. Patterns are represented as rectangular boxes with their name in the upper frame and the pattern interface in the lower frame. In the main frame of the pattern, the component structure with classes and their relationship is shown. Standard UML notation is used for these features. Pattern associations are represented with dotted arrows between patterns. They refer to refinement and combination relationships.

Pattern diagrams from [31] differ in that they also provide a top-down view on frameworks including patterns. Framework views include external and internal subsystems that are connected by dependency relationships. Patterns in turn enclose a type and associations with other patterns. These distinct properties support the two-level concept of pattern-oriented frameworks. Instantiated frameworks are represented with class diagrams, which contain original patterns as dotted line around their components. The reduction process removes these frames. Eventually, the final class diagram does not contain pattern references.

**Dynamic Elements** In addition to static aspects that are described with class diagrams, design patterns consist of dynamic aspects. UML provides diagrams and elements for their representation. Interaction diagram is the collective term for these 3 types of diagrams provided: sequence diagram, collaboration diagram and activity diagram.

A sequence diagram shows a set of interactions between a set of selected objects in a certain situation (context) with emphasis on chronology. In comparison, a collaboration diagram shows a set of interactions between objects in a context with emphasis on the relationships between objects and their topology. Sequence and collaboration diagrams provide different views on the same sequence of interactions. An activity diagram is a special kind of state diagram that consists predominantly or exclusively of activities. It specifies either the sequence of a single operation or a work flow, that is, a sequence in which several objects participate.

It has been explained earlier that, in contrast to class diagrams, interaction diagrams only have indirect value for the developer. Their development determines roles and responsibilities of classes and objects. Consequently, implementation of operations follows these scenarios. Interaction diagrams are object based. That means that interactions and collaborations between objects are shown.

**Non-formal Elements** Design patterns consist of more than class structures and collaboration diagrams. Textual documentation with description, advantages and disadvantages, usage scenarios, and keywords for pattern retrieval are important components of a design pattern. Consequently, a design pattern description must provide room for these elements.

**Properties** A pattern becomes an identifiable component of the system during object-oriented design. In combination with its components, a design pattern represents a self-contained entity that provides an interface to its environment. Its components collaborate to fulfill pattern's responsibilities. Design patterns interact with classes and other design patterns. In addition to dynamic interaction between patterns, they can also be statically combined and appear as a single, merged, entity. The next subsection recapitulates refinement and combination of design patterns in the context of object-oriented design.

## 6.4 Refinement and Combination

Design patterns taken from catalogues like [14] cannot be copied directly into an application model. The process of pattern application involves customization and specialization, as explained in section 4.4. Classes are added, pattern components and operations are renamed, interactions between classes are identified. The pattern model used in this paper does not strictly distinguish between pattern and application level. Transition is a continuous process between these levels.

**Refinement** The process of completing and extending patterns, renaming components, renaming and re-implementing component and pattern features is called *Refinement*. The foundation is provided by basic patterns such as *List* or *Container*. Patterns from pattern catalogues, such as *Composite*, are either base patterns by itself or are refinements of first-level patterns. For example, *Composite* can be designed as a refinement of the *Container* pattern. Further steps specialize the pattern. Reuse of refinement steps results must always be kept in mind. Therefore, the designer has to find the right granularity of steps. Eventually, a structure is developed that has all properties for the concrete application. This final pattern is part of the application and will be instantiated just like classes are instantiated at runtime.

**Combination** Multiple inheritance between classes merges responsibilities and properties of two or more classes. Objects of these classes possess attributes and methods from their original classes. In addition, new properties and operations can be added. The respective equivalent of multiple inheritance for the pattern model is *Combination*. By combination is meant the process of joining several patterns and thus constructing a new design pattern. This product integrates responsibilities and functionalities of its base patterns.

A number of manipulations can take place during a combination step. Since combination relates to refinement like multiple inheritance relates to single inheritance, obviously all refinement transformations are also allowed for combination. If no changes are made, the union of components from all original patterns will form the set of components for the new pattern. Components can be merged to form a new component. By default, properties and operations of the two or more original components are combined. Alternatively, individual or all properties and operations can be merged.

A special case is self-combination. This refers to combining a pattern with itself. The application of this opportunity should be explained: Basic design patterns are generally developed with a minimal structure. For example, the *Composite* pattern in its basic form only provides a single `Leaf` component. However, a concrete application will certainly require a number of concrete classes to fill the composite. In order to create a modified *Composite* pattern with 3 leafs, the designer uses 3 composite patterns. Class components `Composite` and `Component` are merged completely, that is, all their attributes and operations are merged, too. Their names might be adapted for the concrete application. The `Leaf` components are also renamed, but continue to exist as separate components. All of them are in an inheritance relationship with `Component` (see also example in figure 27). Other applications of self-combinations could be found, if an attribute or an operation should be multiplied. In that case, the identical components are merged together including all but the one feature that is to be replicated. The replicated feature has to be renamed to reflect its contextual meaning.

Combination and self-combination open many opportunities for creative pattern design. It was shown that some patterns can be combined very naturally, such as *List* and *Iterator*. More examples including a complex application have been developed in [8].

## 6.5 Review

This pattern model is suitable for patterns that can be expressed with class diagrams, for example the GoF patterns. In contrast, patterns whose essence lies in more abstract principles cannot be applied with this approach. Examples can be found in the General Responsibility Assignment Software Patterns (GRASP) from [19]. Objectives and principles such as encapsulation and distinctive roles are important design guidelines and can thus be called patterns, but they cannot act as system components due to the lack of a self-contained structure.

The question remains: Which influence will pattern-oriented system design have on the quality of software systems? Further studies are required to provide definitive statements. Only few, academic, applications have been developed using a pattern model. However, some observations can be made already.

On the negative side, design with patterns adds further complexity to software engineering. Qualified engineers are required to, first, implement a library or framework of patterns and, second, to apply, refine and combine patterns to construct a complex application. However, a library of base patterns and frequently used combinations and refinements will be developed by pattern experts and provided to the end-developer similar to functional libraries and object-oriented frameworks nowadays. Nevertheless, developers need profound knowledge of their pattern library as well as education on the refinement and combination process, which enables them to fully benefit from the advantages of the pattern model. Consequently, learning effort for new developers is higher compared to conventional object-oriented programming.

On the positive side, advantages of design patterns are fully realized with pattern model design. Application developers can utilize best-practice solutions, thus enhancing flexibility and reusability of systems. In addition, effort of pattern implementation is reduced because base patterns can serve as templates for refinement and combination. Developers only need to supply and implement application specific details.

The usage of design patterns to construct systems adds documentation and increases maintainability because the properties of patterns are known to software engineers. Therefore, patterns must be identifiable in pattern diagrams, even after refinement and combination steps. Documentation remains an important activity. It will be supported by tools that keep track of the refinement and combination process. Thereby, the history of a pattern can always be determined.

To summarize, different approaches of object-oriented design have been introduced. Finally, the pattern model was presented, which has been successfully applied to construct a complex application. Based on this pattern model, pattern-oriented design utilizes the concepts of patterns as model elements and system components, refinement and combination of design patterns.

The second part of this thesis introduces support for practical application of the pattern model during object-oriented design: The successful CASE tool Rational Rose is to be extended to provide design pattern support.

# Part II

# Integration of Design Patterns into Rational Rose

Previous sections have shown the benefits of design pattern usage in object oriented design. Considering the visual nature of object-oriented concepts including diagrams for static and dynamic properties, development of visual support for the pattern-oriented design model is reasonable. In fact, the experience of several authors [6, 21, 31] has shown that graphical tool assistance is necessary for design and implementation of design patterns and pattern-oriented applications. Pattern manipulation, refinement and combination is a complex and demanding process that requires help to keep track of models and provide well-structured documentation.

Computer-Aided Software Engineering (CASE) tools already provide software engineers with software development assistance. Functionalities vary from product to product. Commonly a selection of these features is available:

- Diagram (model) creation and processing
- Code generation
- Reverse engineering
- Team development
- Central repository of models and files

CASE tools are available for structured and object-oriented software development. Examples of object-oriented CASE tools include microTOOLS's *objectiF*, Aonix' *Software through Pictures (StP)* and Rational Software's *Rose*. Typically, UML notation and various diagrams are supported by current products. However, none of the tools that have been investigated provides design patterns support, although design patterns have gained popularity a few years ago. One of the reasons could be the lack of a clear UML notation guideline for pattern representation. The main goal of this thesis has been to develop a concept of pattern integration in object-oriented CASE tools, in particular Rational Rose.

**Rational Rose** Rose is a product of Rational Software Corporation[2]. Development of its principles has been driven by three of the leading scientist in the field of software development: Grady Booch, James Rumbaugh and Ivar Jacobson. Among other features, Rose provides support for component-oriented, iterative development, an architecture based on models and diagrams, and round-trip engineering. Since the scientists leading UML development are Rational associates, the language is used intensively in Rose. Modeling is based entirely on UML diagrams.

[2] http://www.rational.com/

# 7 Existing Approaches and Solutions

Rose does not provide design pattern support. There are no model elements or documentation fields referring to patterns. Considering the success of patterns in object-oriented software engineering, it is an interesting fact that Rational did not implement pattern support, yet. However, third-party software developers are only limited substitutes. Two different concepts will be presented before explaining an approach based on the pattern model and object-oriented design using patterns as introduced in the previous sections.

## 7.1 Blueprint Technologies — Framework Studio

Framework Studio by Blueprint Technologies[3] can capture and apply content, such as code segments, components, patterns, and frameworks. It uses a database that stores artifacts and additional information for retrieval. Framework Studio provides a library of patterns from the GoF and Buschmann books [9, 14]. Figure 5 shows the *Composite* pattern entry. Meta-information from

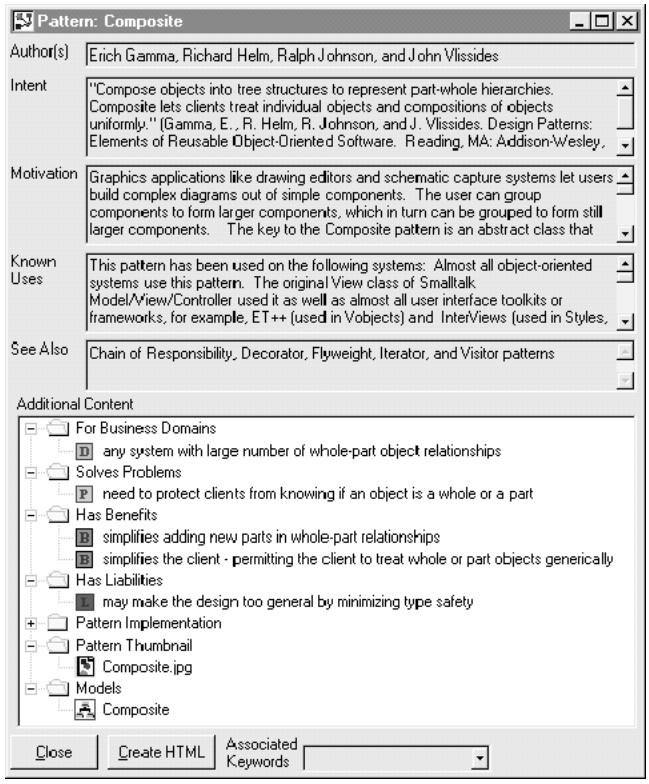


Figure 5: Framework Studio: Documentation Window for *Composite*

the book such as intent, motivation and known uses is available. Furthermore, keywords and formalized content, for example benefits and liabilities, increase

[3] http://www.blueprinttech.com/

usability. These fields can be queried when the developer looks for a pattern with certain properties or a particular application domain. In addition, every artifact and, thus, every pattern contains a model, that is, a Rose diagram consisting of classes with operations and relationships.

Framework Studio is a program separated from Rose. It has access to Rose models, the developer selects classes within Rose that are to be manipulated with Framework Studio. Following, the process of capturing and applying patterns with the tool is explained.

**Pattern Capture** Capturing artifacts is not complicated. First, one typically creates a class diagram of the pattern with Rational Rose, including attributes, operations, associations, and documentation. Second, a new pattern definition is entered in Framework Studio, including description, domains, keywords, benefits, and liabilities. Third, one selects the elements of the pattern within Rose. A menu command exports these components to Framework Studio. The pattern editor selects classes that are required for the pattern and others that are optional. Attributes and operations can also be marked as required or optional. Furthermore, implementation files can be added. Files can include source code fragments, such as sample implementations or frameworks for extensions. In addition, supporting documents such as use cases can be appended. After final review, a new pattern is created in Framework Studio's repository, which is a Microsoft Access database.

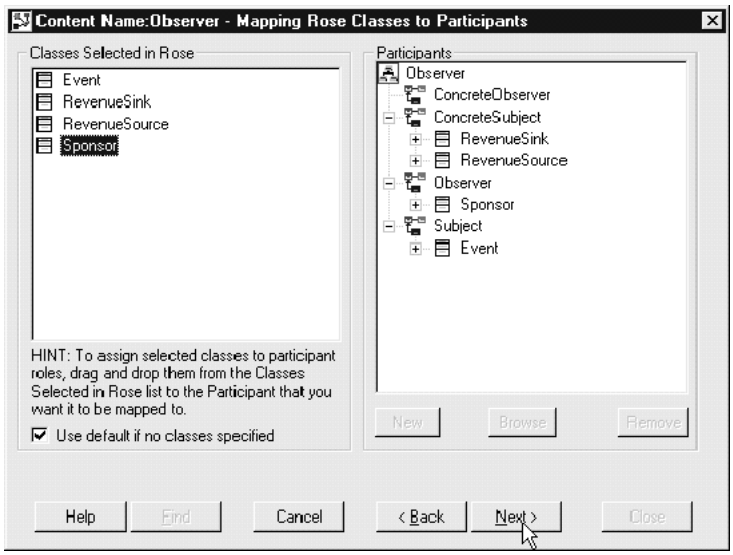


Figure 6: Framework Studio: *Observer* Pattern Application

**Pattern Application** Applying a pattern to a concrete application is a process guided by a wizard. The repository browser is used to select a pattern from the library or to search description and keywords. The developer selects the pattern components that shall be used if there are any optional classes, attributes

or operations. Classes from patterns can be either applied to classes already present in a Rose model or, alternatively, new classes are created. In the first case, the existing classes must be selected in Rose. The selection is shown in a dialog box (see figure 6). The developer uses the dialog to map pattern components to model classes. If no classes are selected, the pattern and its components will be replicated exactly the same way as they have been captured. Names of classes, methods and attributes can be modified. The final review presents the changes to be made. Eventually, Framework Studio updates the Rose model. It is also possible to create a new class diagram with the pattern classes.

**Review** Framework Studio represents a useful tool to capture, process and reuse artifacts of object-oriented design. Model elements can be supplemented with documentation and hints for later retrieval. Particular support is provided for design patterns: In addition to a template of documentation suitable for patterns, the product provides a library of well-known design patterns.

Nevertheless, some deficiencies can be observed. Although it is possible to edit documentation and additional information of patterns, the model could not be altered. The reasons are probably the complex dependencies upon the class configuration. Thus, it is not possible to correct an error found in a pattern. Alternatively, the developer can import the pattern into Rose, apply the changes, and re-capture the pattern.

Framework Studio does not treat patterns as separate system components. Therefore, the pattern model concept introduced in this paper cannot be applied. Because pattern editing is not possible, refinement and combination of patterns are not supported. Moreover, the pattern cannot be identified in the system model after pattern application. This represents a major requirement to improve maintainability of applications developed with patterns.

## 7.2 QOSES — Quarry

*Quarry* is a concept of QOSES[4] to define design patterns. It uses UML's Object Constraint Language (OCL, see [30]) to define patterns that can be used to generate designs, documentation, and code. The user of the design pattern selects the model element, which the design pattern should be applied to. Quarry takes that model element and generates the corresponding model elements according to the design pattern definition. It is also possible to simplify enforcement of design standards. For example, company design standards might require that all classes must have a `debug_flag` attribute. The developer can create an "OCL pattern" and apply it to all of the classes in the model.

Quarry regards patterns as rules that can be applied to a class or a set of classes (input) and create an updated class or pattern (output). UML elements are used to define the pattern: classes, methods, attributes, states, transitions, messages, objects. UML's OCL, which typically defines constraints for model elements is used here to select the context of the pattern element and define property names. Figure 7 shows the application of the *Singleton* pattern to

[4] http://www.qoses.com/

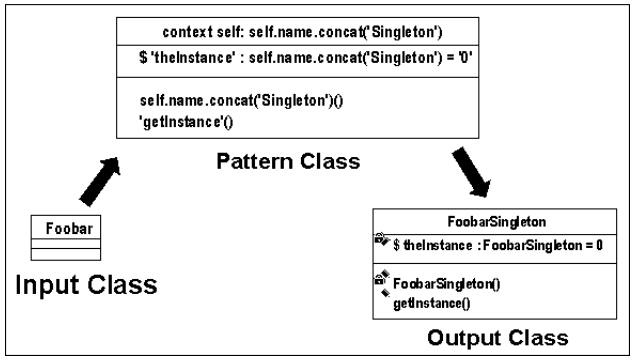


Figure 7: Quarry: Application of *Singleton*

a class using a pattern definition in OCL[5]. The `context` keyword defines the context for the definitions that follow. In the example, `self` refers to the input element. Class and attribute names are derived from names of input elements. All OCL expressions should resolve to character strings.

In another example that is not shown here, a *Model-View-Controller* pattern is presented. The definition of all classes (Model, View, Controller and Observer) is based on one single source class. Thus, pattern application to this class creates a set of four classes including relationships.

The process of pattern definition starts with prototyping the pattern in generic terms using UML. Next, the context is defined for each element in the pattern. Finally, generic terms are replaced with OCL expressions that recreate the pattern based on its context.

**Review** Authors of Quarry have provided Rose integration. Thus, it is possible to apply the Quarry Generator to a selected class. For the two examples mentioned earlier, OCL definitions and models are available.

The usage of OCL as a template language to define pattern elements including class definitions represents an interesting approach. However, Quarry is a limited solution. Only a small number of patterns can be derived from a single original class. In contrast, most patterns generally integrate different classes in a flexible way. The transformation of a class into a singleton might be reasonable, but it is not possible to create the *Abstract Factory* pattern from a single class while providing understandable names.

Consequently, Quarry can only be applied to a fraction of patterns. For others, a pattern definition can be created with OCL. However, this definition will be limited to recreating the pattern template without providing naming support for classes, attributes or operations.

[5]Example taken from [25]

## 7.3 Limitations of Existing Approaches

Two approaches have been presented that pursue different concepts of integrating design pattern support with the Rational Rose CASE tool. Although they provide interesting details, several limitations can be observed that will prevent widespread application.

**Integration with Rose** Developers look for an integrated development environment. Additional programs require additional training and support. Therefore, a solution would ideally be fully integrated within Rose or would be a seamless extension. Definition, manipulation and application of patterns should be executed within Rose. Unfortunately, Rose does not provide full access to its internal data structures and menu options.

**Patterns as System Components** The design approach based on design patterns presented in this paper understands patterns as building blocks of systems. Tools should support this concept by providing patterns as model elements. Existing tools can create patterns, but identification of patterns in program design is not possible. This reduces comprehensibility of program design.

**Refinement and Combination** Editing patterns to correct errors and to increase usability, functionality or to adapt a pattern for a particular application is a major requirement. The pattern model approach understands the refinement and combination of pattern as the essential system construction process. A tool must support it by displaying the patterns to be edited.

It is understood that Framework Studio's functionality comprises more than pattern management. Nevertheless, a Rose add-on designated for design patterns is required. The next section presents a concept for such an extension to Rational Rose.

# 8 Integration Concept

This section proposes a concept that integrates object-oriented design using patterns with Rational Rose. Technical or organizational conditions do not limit this approach: full control of Rose's internals is assumed. Thus, this section presents a solution that could be implemented by Rational developers. In contrast, the next section (9) presents a prototypical implementation limited by the interface provided by Rational to access Rose's internal structures.

## 8.1 Model Element Pattern

Section 6.3.3 presented patterns as model elements of object-oriented design. Notations from UML standard and other authors have been compared. It has been concluded that the graphical notation of patterns must include patterns as a container of pattern elements, an interface, and relationships among pattern components and between two patterns.

**Static Aspects** In addition to the already existing static elements *Package*, *Class*, and *Interface*, a new model element *Design Pattern* will be added to Rose. It carries a name and may or may not be part of a package. A pattern in Rose is a container for components, similar to a package. Classes and interfaces can be pattern components. It may also be possible to have patterns as components inside other patterns. However, this extension adds unnecessary complexity to the model. The pattern model provides combination as a means of pattern cooperation. Associations and inheritance relationships complete the static model of the new pattern element.

The Pattern specification will include a detailed description as present today in pattern catalogues. Potential documentation fields are motivation, application, domains, benefits, liabilities, consequences, examples, and related patterns.

Patterns are related to packages. Their representation in Rose will be similar. The model tree includes pattern as structuring elements with its members: components and relationships. Each pattern will have at least one class diagram that shows pattern structure. Standard UML representation is used. A new type of diagram, the *Pattern Diagram* shows patterns and relationships between them. These relationships include associations, refinements and combinations. Thus, the development of the pattern model and the architecture of the system is displayed.

**Dynamic Aspects** The importance of documenting dynamic aspects of design pattern has been explained before. Therefore, it must be possible to create collaboration, sequence, and activity diagrams for a pattern and its components. These diagrams document cooperations and interactions of class-components. They support developers with detailed design, because responsibilities and interfaces of components are identified. Furthermore, interaction diagrams simplify implementation of pattern classes with object-oriented programming languages.

## 8.2 Working with Design Patterns

Handling patterns with Rose should fit into Rose's model-diagram-architecture. This also means that patterns are displayed and manipulated in diagrams. This subsection illustrates development of and with design patterns in Rose. An example accompanies the description of design pattern handling. Creation, editing, refinement, and combination will be covered.

### 8.2.1 Creation

Patterns can be created using the menu bar or the context menu of packages or patterns in the model tree view, or in a diagram presentation. The command creates an empty design pattern without components at the current level, that is, as part of the currently selected package.

**Example** For instance, a new pattern is created in an empty model. It is immediately displayed in the model browser. It can be renamed to *List*. This pattern serves as an example for this and the next section. In addition, the system creates a pattern interface that contains public operations and properties of the pattern. Furthermore, a class diagram for the pattern components is created. Initially, it is empty or contains only the interface component.

### 8.2.2 Editing

Pattern properties and components can be edited similarly to packages: either the model browser or diagram views are used. Changing the specification and documentation of the pattern itself is started by selecting a command from the pattern's context menu. A dialog would be displayed that shows pattern properties.

Pattern components are edited as if they were member classes of a package. Consequently, components are added by selecting a command from the menu bar or the pattern's context menu. They are edited using standard Rose tools. Attributes and operations can be added and changed, specification and documentation can be edited.

Interaction diagrams are added accordingly. Pattern components and their instantiated objects participate in these diagrams. Interaction diagrams document pattern's behavior in certain scenarios. Therefore, a number of different diagrams is required to describe a pattern in detail.

**Example** The *List* example is resumed with adding 3 components and the interface to the pattern[6]. Interface and components are added to the class diagram. Next, associations are created between the components. A sequence diagram shows interactions between component objects when an element is added to the list.

[6]Screenshots are found in the next section, which covers the prototype implementation. See also figure 13

### 8.2.3 Refinement and Combination

The refinement and combination process represents the core of the pattern model. Although it has been shown that pattern-oriented programming is feasible without graphical tool support (see [8]), assistance by CASE tools can greatly simplify development and management of design patterns.

The difference between refinement and combinations is the number of source patterns that participate in construction of a new design pattern. With refinement, one pattern is edited toward a particular domain or application. This can be achieved by editing a pattern's properties, and adding or changing pattern components. Combinations refers to the creation of a pattern based on a number of source patterns. The result integrates functionality and, thus, compacts system design. However, flexibility might be lost during this process.

Combination and self-combination represents a challenge to CASE tools. Essentially, a number of model elements are to be merged into one. Several requirements are taken into consideration:

**Documentation and Specification** Information from documentation fields cannot simply be merged. The developer must review description, keywords etc. from source pattern and use this information to compile documentation for the combination. Rose can support this process by showing source documentation in different windows and including it in the documentation of the combined pattern.

Updating the pattern's specification could be supported similarly: The system provides a default taken from one of the source patterns. By displaying specifications of other participating patterns, the designer is supported to correctly and completely specify the new pattern's properties.

**Components** By default, the component set of the new pattern comprises the union of component sets of all source patterns. Mathematically, it can be seen as a bag. That means, it can contain more than one exemplar of the same component after a combination of pattern with identical components or self-combination.

The pattern designer has several options to change the default component setup. Components can be combined, that means, merged. Generally, only components from different source patterns are allowed to be merged. The number of combined components is not limited to 2, in rare cases where more than 2 patterns are combined it might be required to merge more than 2 components.

By default, the combined component contains all attributes and operations from its source components. Nevertheless, as components in combined patterns can be joined, attributes and operations may or may not be joined. For instance, each of the two source components have an attribute *date*. It is only necessary to have one attribute of this kind in the combined component. Deletion of properties is not supported because functionality would be reduced. Instead of deleting it, both attributes are joined. As a result, specification and documentation of both source attributes are sustained and semantics remains intact.

Associations of source components are preserved. The combined component will contain all associations from its source components. Once again, associations can be joined to remove redundancies. The same applies to inheritance relationships. Joining inheritance relationships can create situations of multiple inheritance.

Pattern support for Rational Rose must support all these options and configurations. Different possibilities can be imagined:

1. *Full graphical control:* The designer combines components by selecting a number of them and choosing a menu option. Attributes, operations and relationships are rearranged with mouse clicks. Because of the size of patterns and the number of components, which might fill the screen, it must be possible to selectively control the display of components and patterns.
2. *Wizard control:* A wizard-like applet guides the designer through the process and provides options and assistance. Steps are executed successively. Graphical displays support the combination process. For instance, source patterns their components can be shown, alternatively the new pattern and its components are displayed.
3. *Dialog control:* A complex dialog shows all potential source patterns with their components. The designer selects the patterns that should be combined. The dialog displays the corresponding attributes and operations. Attributes and operations are merged by selecting two or more candidates followed by a menu command or button.

It is common to all 3 methods that the specification developed is eventually applied to the model, creating a new pattern with the desired component setup.

**Diagrams** Interaction diagrams are associated with patterns. In case of pattern combination, diagrams must be updated accordingly. Rose might support by replacing source components with their new equivalents. Not all modifications can be executed automatically, the developer needs to review diagrams and check their plausibility.

**Pattern Interface** The interface defines behavior superimposed on a pattern's components, for instance to initialize a pattern. It is implemented in attributes and operations of the pattern interface, which are called pattern features. The name *interface* might be misleading, since it is not an interface in Rose or UML meaning. A pattern interface can be understood as a class with an instance that serves as a proxy for the pattern: Clients do not communicate directly with pattern components, but by using the interface.

A designated component could serve as the pattern interface in Rose. Its attributes and operations represent the set of pattern features. Generally, one pattern feature will provide a reference to a component instance that can be used to access other component instances. Therefore, the interface component contains one or more attributes of a component type, completed by a number of operations to invoke pattern functions.

**History** The process of pattern combination and refinement creates dependencies of patterns among each other. Management of changes becomes a significant task. How are changes in source patterns reflected in combined or refined patterns? Several solutions for this problem are possible:

1. *Change Prohibition:* The system will prevent changes on patterns if refinements or combinations exist that are based on it. Consequently, no problems of updated patterns occur. However, this greatly reduces the flexibility of the pattern design model: It is not possible to remove known errors from a pattern. To conclude, the restrictive handling of pattern changes does not allow reasonable software design and development.
2. *No Passing of Changes:* Changes to patterns are always allowed, but are not passed on to dependent patterns. This solution assumes greater differences between development steps. Thus, modified patterns provide different functionality and are in the majority of cases not affected by errors or updates in source patterns. If modifications are required, they must be executed manually, that is, the designer applies changes to all dependent patterns.
3. *Update Propagation:* Modifications to source patterns are automatically passed on to dependent patterns. It is assumed that changes can be reapplied to dependent patterns, because development steps are small and a non-ambiguous correlation can be found between source and new pattern. A more detailed discussion of this approach follows after this enumeration of options.
4. *Alternatives:* The user is given a choice between three alternatives presented. The policy is selected individually for each pattern or each refinement step. This options provides greatest flexibility in the face of a variety of scenarios and configurations that are possible.

Update propagation requires the storage of information at the time patterns are combined. At the time of combination, this information must be recorded:

- Configuration of source patterns participating in a pattern combination (Pattern elements: pattern name, components, attributes with types, operations with signatures, associations, inheritance, class dependencies). Storage must enable simple comparison with updated pattern.
- List of applied transformations (Component combination, renaming, attributes, operations, etc.)

It is assumed that update propagation is too complex and ambiguous to be executed entirely automatically. User intervention is required to resolve ambiguities. To prevent unauthorized changes, the developer manually starts the process of propagation, perhaps after one or more source patterns have been updated. A command is executed that takes the current combined pattern as reference. In practise, this process will run like this:

1. Compare current configuration of source patterns with the state stored at time of combination.

2. Involve designer to resolve potential ambiguities, for instance to associate components or their features with source elements after they have been renamed.

3. Create list of changes on source patterns.

4. Apply changes to target pattern. Consider renaming and component combinations during combination step.

Rose must be extended to provide support for this scenario. A wizard-like dialog is required to guide the developer through the process.

**Example** To continue the example, the *List* pattern is combined with an Iterator to form the *List Iterator*, a pattern that provides a list as well as services to iterate it. The example is taken from [8]. The combination is visualized in figure 8.

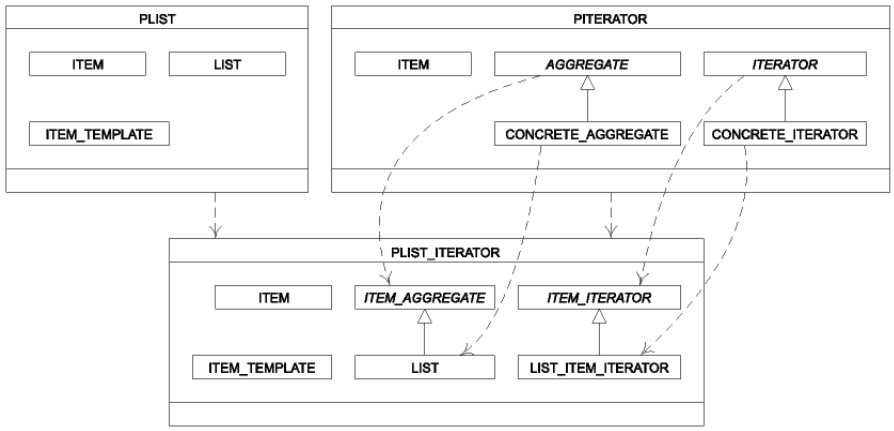


Figure 8: Combination *List* with *Iterator*

As can be seen in the diagram, the `LIST` component from `PLIST` is combined with `CONCRETE_AGGREGATE` from `PITERATOR`. Other components, such as `AGGREGATE` and `ITERATOR` are refined, in particular, renamed. Several operations are renamed, too, which cannot be seen in the diagram. More details are to be found in the next section.

## 8.3 Final Result

Rational Rose is used by different developers for various reasons. Its role ranges from documentation repository to integrated development environment with code generation and reverse engineering. Consequently, design pattern support should provide similar options.

**Repository and Documentation** Rose can be used as a pattern repository. Thus, patterns catalogues from various books are accessible from one single environment. Retrieval tools support the designer in finding patterns with certain characteristics. Pattern repositories can be edited and appended as appropriate. Registration of new pattern requires profound knowledge of the catalogue to avoid redundancies. Often new entries can be created by refining or combining existing patterns.

Rose also serves as a design and documentation tool for software projects, which use patterns. System design is executed with Rose, patterns are used as required. Predefined patterns can easily by applied for projects, because they are stored within the same tool. No conversion is necessary, designers can concentrate on their main tasks – refining patterns for specific applications.

**Code Generation** Continuous maintenance and synchronization of the Rose model with the actual system guarantees maintainability at a later stage. However, Rose also provides functions to automatically synchronize between model and implementation. Source code generation is available for Java, C++, and other programming languages. For example, for every package in the Rose model, a corresponding Java package is created. It contains class definitions complete with attributes and operations for each class in the package. Code generation can be customized to a certain extent with a properties dialog within Rose.

The developer continues development by providing method implementations. Moreover, names can be changed, attributes or operations can be added. To synchronize model with implementation, Rose provides functions to reverse-engineer the updated source files. Changes are reflected by model updates.

Pattern-oriented design with Rose can provide two different options of code generation:

1. Conventional *object-oriented* source code generation. Eventually, completely refined design patterns represent class packages that become packages in one programming language. Implementation continues at source code level.

2. PaL or similar *pattern-oriented* source code generation. Section 5.3 presented the language which is completely based on design patterns. PaL source can be generated with complete refinement and combination history. Implementation of operations can be supported by specification fields in Rose. Implementation proceeds by editing PaL source.

Because the pattern-oriented design approach is based on the pattern model that also serves as the foundation of PaL, this language is particularly suited to be the target language. Manual code manipulation cannot be avoided though, because completed operation implementation is hardly possible with Rose.

**Example** The *List Iterator* pattern developed in the previous example section is actually not yet prepared for code generation, further refinement is needed. However, for demonstration purposes, it will be used. To generate source code for a *List Iterator*, the developer chooses a command from the context menu of a pattern component. One out of two language options can be chosen:

First, the Java programming language can be selected. Thus, a package with six classes is created. Classes will have attributes and operations as exist in Rose pattern components. For each component, a separate file is created. Documentation is added as comments to classes and features.

Second, PaL can be selected. Code is generated for the current pattern and all its source pattern, in this case *List*, *Iterator*, and *List Iterator*. PaL syntax is used to express combination and refinement steps. As a result, one file contains the pattern definition for the current and its base patterns.

# 9 Prototypical Implementation

After introducing the concepts of object-oriented design using design patterns and presenting an approach of integrating patters with Rational Rose, this section explains the implementation of a prototype that extends Rose with pattern support. Furthermore, the proposed usage of the new features is explained with examples.

## 9.1 Extending Rational Rose

Rose provides an extensibility interface that allows users and developers to enhance its functionality. An interface is provided to access model elements. Thus, existing objects can be manipulated or removed, and new objects can be added. Access is provided via an ActiveX control or using *RoseScript*, a Basic environment within Rose.

Furthermore, menu bars and context menus can be extended. A text menu file defines menu extensions. The menu file for the pattern extension is presented in figure 9. It defines a new sub-menu *Pattern* under the *Tools* menu in Rose.

```
Menu Tools {
  Separator
  Menu "Pattern"
  {
    option "&New Pattern"
    {
      RoseScript $PATTERN_PATH\Scripts\new_pattern.ebs
    }
    option "&Combine Patterns"
    {
      RoseScript $PATTERN_PATH\Scripts\combine.ebs
    }
  }
}
```

Figure 9: Pattern Extension Menu File `pattern.mnu`

It consists of 2 commands to create a new pattern and to start the combination dialog. The new menu is shown in figure 10. Rose supports extensions to the


Figure 10: New Pattern Menu

environment with the *Add-In Manager*. To create a new add-in, the Microsoft Windows registry database is updated. Thus, the add-in is registered with its name, menu file, and other properties. The add-in manager in Rose is used to activate and deactivate add-ins. Figure 11 shows the file that updates the

```
[HKEY_LOCAL_MACHINE\SOFTWARE\
 Rational Software\Rose\AddIns\Pattern]
"Active"="Yes" "InstallDir"="C:\\Users\\Danko\\Pattern\\Pattern"
"LanguageAddIn"="Yes" "MenuFile"="pattern.mnu" "Version"="0.01"

[HKEY_LOCAL_MACHINE\SOFTWARE\
 Rational Software\Rose\StereotypeCfgFiles]
"File3"="PatternStereotypes.ini"
```

Figure 11: Registry File `pattern_addin.reg`

registry database. The second registry entry refers to a stereotype definition file. Its meaning will be explained shortly.

## 9.2 General Concept

The target of the prototype implementation was to provide tool support for pattern oriented system design. This has been demanded by various authors (e.g. [6, 31]) to simplify development and increase usability of the concepts. Although it was tried to closely follow the approach developed in the previous section, limitations prevented the implementation of more interesting ideas.

A new model element with unique properties could not be implemented, because a fundamental extension as this is not supported by Rational Rose. As an alternative, packages have been used as a container for pattern components. Details are found in the next subsection.

In order to serve as a pattern repository, complex documentation dialogs are required, which present pattern information as detailed as it is found in pattern books. However, customized specification dialogs for pattern could not be implemented. Structured information stored in documentation fields is used instead.

In the works of Bünnig and Seemann [6, 7, 29], refinement and combination is always illustrated with diagrams. This graphical notation might serve well to give an overview of the result. However, it is not suited to execute the process. The number of details to be adjusted during combination requires a text-oriented dialog that displays components, attributes and operations as lists. A detailed description is given in the next subsections.

## 9.3 Model Element Pattern

UML packages are defined as follows:

> Packages are collections of model elements of arbitrary type that subdivide the entire model into smaller clear units. Every package defines a name space, that is, names of elements must be unique in each package. Every model element can be referenced in other

> packages, but it belongs to exactly one (home) package. Packages can contain other packages. The topmost package comprises the entire system. [15]

In comparison to patterns in the pattern model, packages are very similar elements. Patterns can contain model elements of different, but not arbitrary, type. They do subdivide the model and define a name space. However, referencing a pattern component in other patterns is not possible in the pattern model. Patterns containing other patterns have not been completely analyzed. A reasonable application has not been found. Therefore, it is not yet supported by the pattern model. Furthermore, it cannot be said that the topmost pattern contains the entire system. In contrast, patterns are combined until the final pattern represents the system itself.

The functionality of packages as containers for other elements enables them to serve as pattern representations in Rational Rose. Components with relationships and diagrams can be created, stored and edited effortlessly. To separate the new pattern packages from ordinary packages, UML stereotypes are used: Every pattern is marked with `<<Pattern>>`. The stereotype was added to Rose standard stereotype list with a file listed in figure 12. Creation, management

```
[Stereotyped Items]
Package:Pattern
Class:PatternInterface

[Package:Pattern]
Item=Package
Stereotype=Pattern

[Class:PatternInterface]
Item=Class
Stereotype=PatternInterface
```

Figure 12: Stereotype Definition File `PatternStereotypes.ini`

and combination of design patterns is explained in the next subsection.

Components of patterns are regular Rose classes without stereotypes or extensions. Therefore, whenever it is written about pattern components, it can be assumed that they contain all properties and features of classes.

The pattern interface is represented as a designated component of the pattern. It is marked with the stereotype `<<PatternInterface>>`. Similarly to the pattern model, the interface contains public properties of the pattern and provides services to other patterns or independent classes. Properties are represented with attributes. These attributes often refer to an instance of one of the pattern components – the starting point of a pattern structure.

Documentation of patterns can be placed in standard documentation fields of packages. Every Rose element provides space for about 12 pages[7] of documen-

[7] About 28000 characters equal 12 pages

tation text. It is recommended to structure the documentation text to follow standard pattern descriptions with context, problem description, solution, consequences, and perhaps examples. Furthermore, references to other patterns can be given. Keywords, benefits and liabilities complete pattern documentation.

Static and dynamic diagrams visualize pattern properties and behavior. Class diagrams are used to show components and their relationships, including associations and inheritance. Figure 13 shows a class diagram of the *PList*

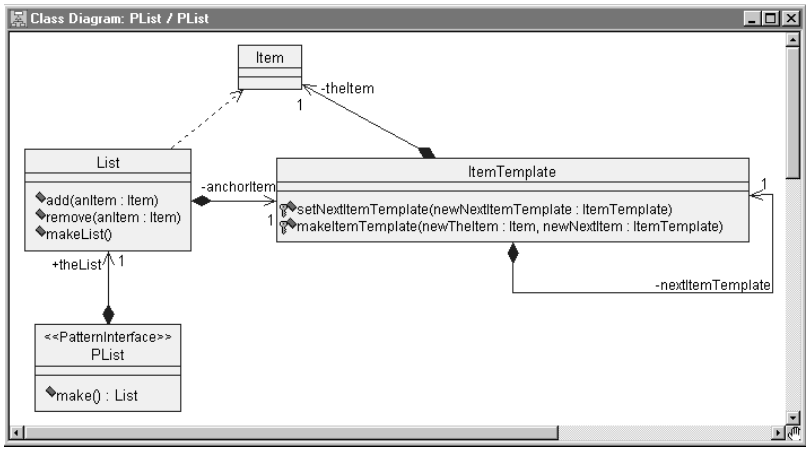


Figure 13: Class Diagram *List* Pattern

pattern, as it is implemented in Rose. It consists of 3 components. Furthermore, there are a number of aggregations and a class dependency between `List` and `Item`. The pattern interface in the lower left part of the diagram contains a reference to an instance of the `List` component and an operation to create an instance of the pattern, that is, instances of its components.

Sequence diagrams are used to illustrate the sequence of messages in a certain scenario. An arbitrary number of sequence diagrams can document different scenarios. Figure 14 shows the addition of a new element to the list. Other types

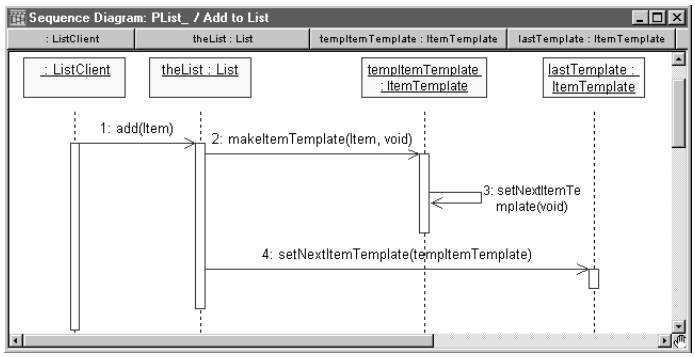


Figure 14: *PList* – Sequence Diagram – Add Item to List

of diagrams can be created. In its current version[8], Rose supports collaboration, statechart, and activity diagrams.

## 9.4 Working with Design Patterns

**Creation** Although it is possible to manually create a new pattern, a menu command is provided to execute this task. It creates a new pattern in the currently selected category. The default name is `NewPattern`. If this name is not available, `NewPattern1`, `..2`, etc. will be used. The pattern contains a class diagram and an interface. Both carry the name of the newly created pattern. Figure 15 shows the new pattern diagram with the interface. The pattern is marked as being a base pattern in its documentation field.

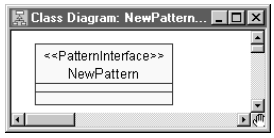


Figure 15: Empty Design Pattern

**Refinement** Development of design patterns with Rose proceeds with adding components, and inheritance and dependency relationships. For example, the abstract base pattern *FactoryMethod* is created by adding 4 classes. Figure 16 shows the class diagram.

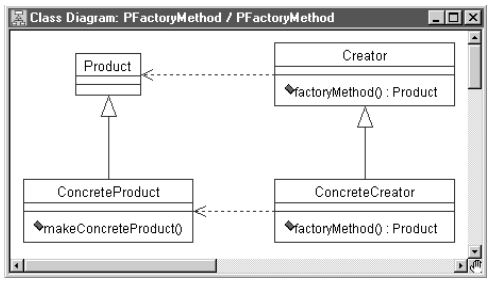


Figure 16: Design Pattern *Factory Method*

## 9.5 The Combination Dialog

The Rose dialog to combine design patterns represents the core of the prototype implementation. It is used to select patterns for combination, merge and rename components, attributes and operations. Its application will be explained with an example: The *Iterator* pattern in [8] is implemented as a combination of the *Factory Method* and a *Container* help pattern (see figure 17). Consequently, 2 source patterns are required. It is assumed that both patterns are available in

[8] Rational Rose 2000, Rose Enterprise Edition, was used for development

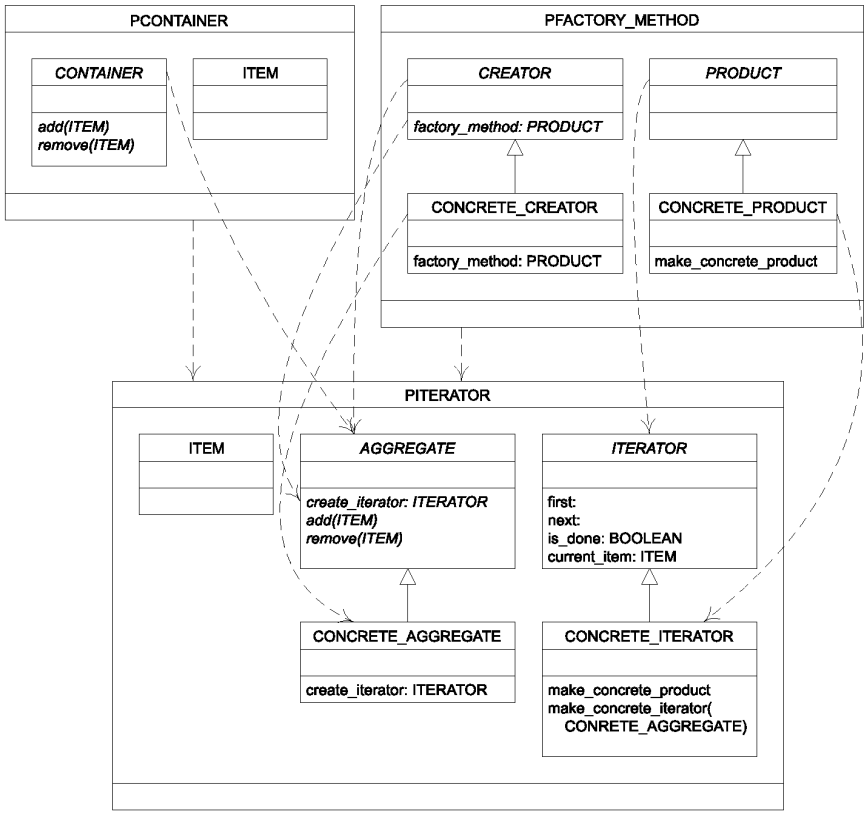


Figure 17: Combination to Create *Iterator* Pattern

the Rose pattern repository. Their Rose pattern diagrams can be seen in figures 16 and 18.

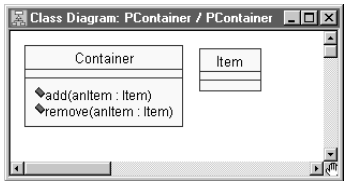


Figure 18: Pattern *Container*

To display the dialog, the command *Combine Patterns* is selected from the pattern add-in menu. A list of all available patterns is displayed on the left side of the dialog. Below, the components from the currently selected pattern are displayed. The developer selects the pattern that should be used and presses the button with the arrow (>). Thus, the pattern is added to the *Combined Patterns* list. Its components are displayed in the *Components* list below. Figure 19 displays the dialog with the 2 source patterns mentioned.

A new name for the pattern can be given in the upper text field *New Pat-*

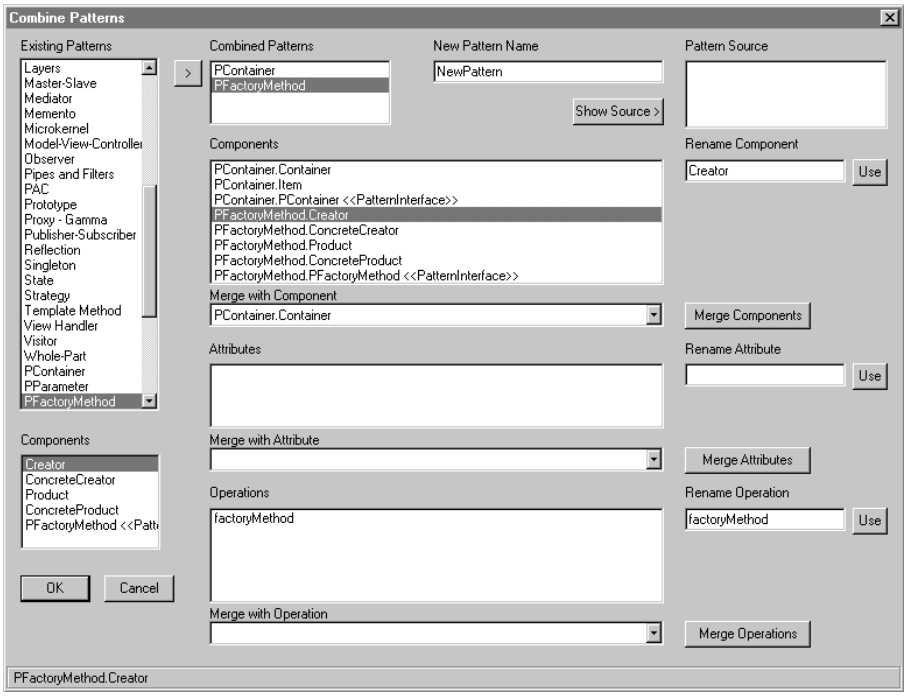


Figure 19: Combination Dialog with Source Patterns

*tern Name*. To distinguish between components with equal names from different source patterns, every component is shown with its full path name, that is, pattern and component name, separated by a dot (.). For example, the fourth line (`PFactoryMethod.Creator`) refers to component `Creator` from source pattern `PFactoryMethod`. The status line at the bottom of the dialog displays the full name of the currently selected item.

**Merging and Renaming Components** Development proceeds with the combination of components. To combine (or merge) 2 components, the first one is selected in the *Components* list box. The second component is selected in the drop box directly below labeled *Merge with Component*. To proceed, the button *Merge Components* must be pressed. Consequently, the 2 original entries in the component list are replaced by one entry that contains the name of the new component and its source components.

The components `PContainer.Container` and `PFactoryMethod.Creator` are combined in the *Iterator* example. The components are selected as described above, and the button is pressed. The resulting line in the component list contains `Container (PContainer.Container, PFactoryMethod.Creator)`.

By default, the name of the new component is derived from the original name of the first source component. However, names can be changed with this dialog. Next, the combined component should get a new name. To rename a component, a new name is entered in the text field to the right labeled *Rename Component*. After pressing the *Use* button, the update is reflected in the component list.

The new name `Aggregate` is used. Figure 20 shows the component list. In it,

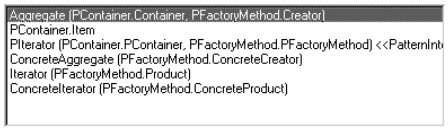


Figure 20: Component List after Combination

two combinations can be seen in lines 1 and 3. The components in line 4, 5 and 5 are renamed. The component `Item` remains unchanged.

**The Pattern Interface** The component that represents the pattern interface is marked with its stereotype `<<PatternInterface>>` at the end of the line in the component list. It is handled like other components. There may only be one interface per pattern. Therefore, interfaces will usually be merged if patterns are combined.

**Renaming Attributes and Operations** The list boxes below the component box contain attributes and operations for the currently selected component. Attributes and Operations can be renamed just like components. The text boxes next to the list boxes are used. In the example, the operation `factoryMethod` from the new component `Aggregate` (originally `PFactoryMethod.Creator`) is renamed to `createIterator`. This is displayed in the operations list box as follows: `createIterator (PFactoryMethod.Creator.factoryMethod)`. For every attribute or operation with a new name, the name is followed by pattern name, component name, and original feature name in parentheses.

**The Definition** The definition of the pattern that is being created can be generated by pressing the *Show Source* button in the upper part of the dialog. The text box in the upper right corner displays the information. The definition of the `PIterator` pattern is created as follows:

```
=== Pattern Definition ====
PIterator
=== Pattern ====
PContainer
PFactoryMethod
=== Components ====
Aggregate (PContainer.Container, PFactoryMethod.Creator)
PContainer.Item
PIterator (PContainer.PContainer, PFactoryMethod.PFactoryMethod)
ConcreteAggregate (PFactoryMethod.ConcreteCreator)
Iterator (PFactoryMethod.Product)
ConcreteIterator (PFactoryMethod.ConcreteProduct)
=== Attribute Renaming ====
=== Attribute Merge ====
=== Operation Renaming ====
```

```
PFactoryMethod.Creator.factoryMethod|createIterator[Aggregate]
PFactoryMethod.ConcreteProduct.makeConcreteProduct|
  makeConcreteIterator[ConcreteIterator]
PFactoryMethod.ConcreteCreator.factoryMethod|
  createIterator[ConcreteAggregate]
=== Operation Merge ====
=== End Pattern Definition ====
```

Although the component list is exactly the same as displayed in the dialog, the attribute renaming is handled differently. Every single line can be understood as a renaming rule. The first line, for example, represents the renaming discussed above. The first part before '|' contains the original operation path with pattern name, component name, and old operation name. The new operation name follows. Finally, the new component name is provided in square brackets.

**Dialog Completion** Assuming the pattern combination is completed, the developer presses the *OK* button in the lower left part of the dialog. The pattern definition with combination and renaming rules is presented in a modal dialog window with the question whether to create a new pattern or not (figure 30). Processing can either proceed or return to the combination dialog. If the developer confirms pattern creation, the definition will be used to generate a new pattern.

Internally, components of source pattern are iterated and added to the new pattern. Combinations and renaming rules are considered and executed. Relationships in source patterns, such as class associations, dependencies and inheritance relationships are recreated. They would be passed on, if source components were combined. Documentation of the new pattern is supplemented with its pattern definition. Finally, a class diagram is created for the new pattern. The new class diagram for the *Iterator* pattern can be seen in figure 21. Notice

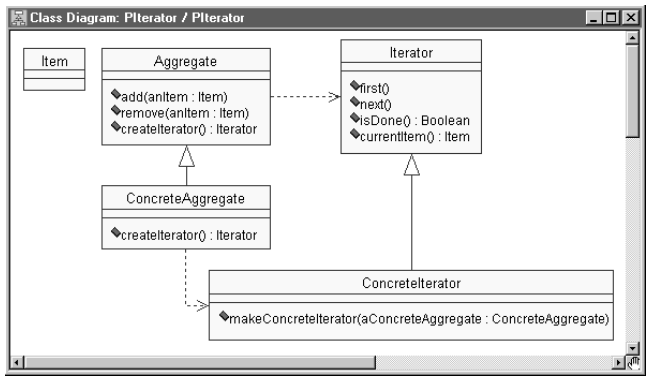


Figure 21: Final *Iterator* Class Diagram

that the return parameter of the `createIterator` operation in `Aggregate` component has been automatically changed from `Product` to `Iterator` to reflect the new component name. Attribute and parameter types are handled similarly. At last, 4 operations should be added to `Iterator`. Eventually, the class diagram

matches the target pattern (compare to figure 17).

**Self-Combination** The abstract *Composite* pattern provides only a single `Leaf` component (see figure 22). Concrete applications require more than one

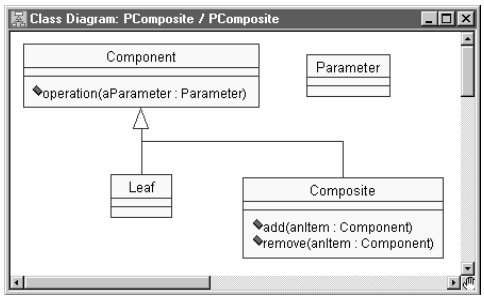


Figure 22: *Composite* Pattern

leaf. Self-combination can be used to clone components. In the following example, a *Composite* pattern with 2 leafs is created.

First, the source pattern `PComposite` is added twice in the combination dialog. Consequently, the component list contains every component of the source pattern twice (figure 23). Only one component should be duplicated. To remove

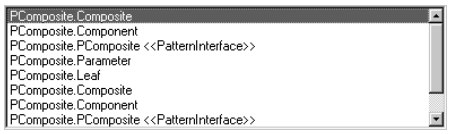


Figure 23: Component List *Composite* Pattern Combination – 1

the redundant components, each of them is merged with its counterpart from the second source pattern. Next, each of the `Leafs` is assigned a new name, for instance `LeafA` and `LeafB` (figure 24).

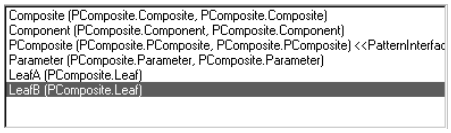


Figure 24: Component List *Composite* Pattern Combination – 2

**Merging Attributes and Operations** Because components with operations, such as `Composite` have been self-combined, each operation is contained twice in the operation list box (figure 25). Redundant operations are again removed by merging operations with the same name (figure 26). For every operation that has been merged, a line is added to the pattern definition. It contains the new name of the operation, its source operations with pattern and

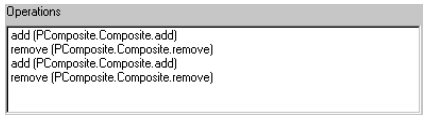


Figure 25: Operation List `Composite` of Component – 1

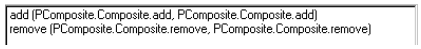


Figure 26: Operation List `Composite` of Component – 2

component name, and the new name of the component that was merged. These are the 2 lines for the operations that were merged:

```
=== Operation Merge ====
add (PComposite.Composite.add,
  PComposite.Composite.add)[Composite]
remove (PComposite.Composite.remove,
  PComposite.Composite.remove)[Composite]
```

Attributes are handled in the same way. Figure 27 shows the diagram of a *Composite* pattern with 2 leafs.

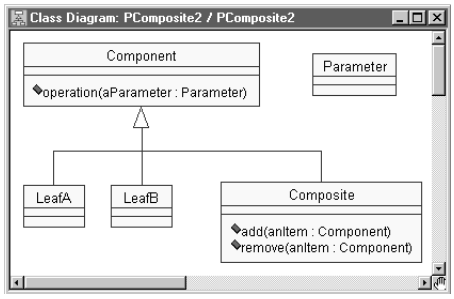


Figure 27: *Composite* Pattern with 2 Leafs

**Example List Iterator** Another complex example is presented that illustrates the ability of the prototype implementation. A *List* pattern (figure 13) is combined with an *Iterator* (figure 21). The new pattern enables the traversal of lists. Therefore, components `List` and `ConcreteAggregate` are merged. Figure 8 in the previous section presents both patterns as well as the combination with arrows pointing from source to target components. Figure 28 shows a part of the combination dialog that is used to execute the combination. The new pattern diagram is shown in figure 29. A number of operations were added to `ItemOperator` and `ListItemOperator`.

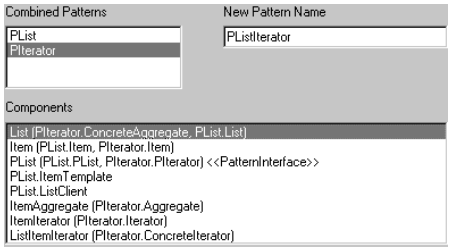


Figure 28: Combining *List* and *Iterator*

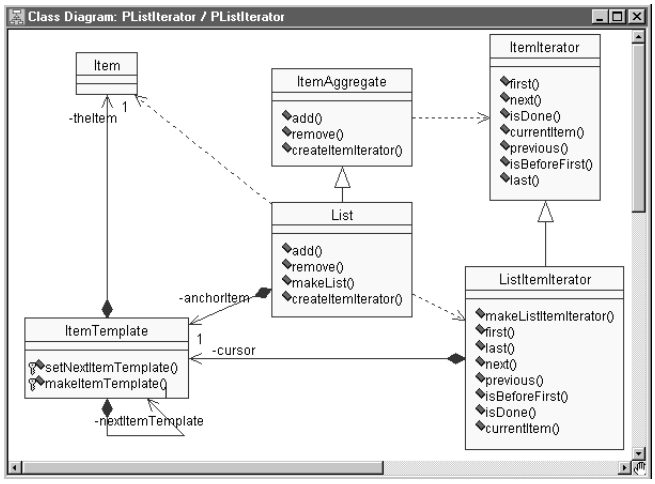


Figure 29: Class Diagram *List Iterator*

## 9.6 Implementation Details

The implementation of the combination dialog was divided into 2 parts, the user interface and the creation of a pattern based on a pattern definition.

**User Interface** The dialog and its controls were created with Rose's dialog editor. Internally, pattern and component lists as well as list of renaming and merging rules are kept. User interactions cause changes in these lists, which is represented by dialog updates. Basically, the dialog enables the developer to create a pattern definition without actually writing it. Instead, model elements are manipulated by button clicks. The program code that controls interactions and updates mostly contains difficult string and list management.

**Pattern Creation** After the developer confirms the creation of a new combined pattern (figure 30), the definition created with the dialog is executed. The implementation is concerned with parsing source patterns and recreating its structure in the new pattern while considering combinations and new names. Attributes and operations are handled according to the developer's specification, new names are used, or features are merged, that is, some features may disap-

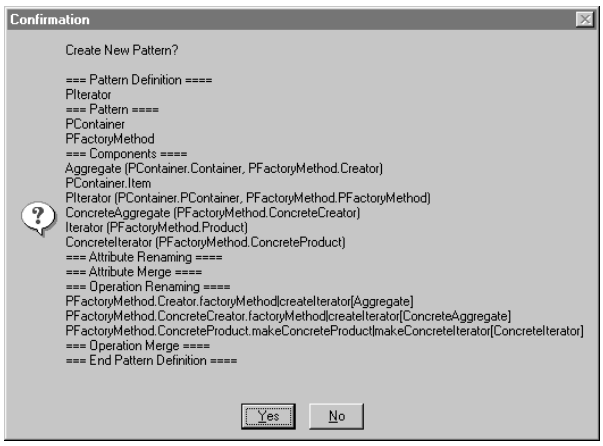


Figure 30: Confirmation Dialog *Iterator*

pear from components.

Lists are kept to manage the mapping of components between source and new pattern. These mappings are required to correctly recreate associations and inheritance relationships that existed before combination. For example, if two components were subclasses in inheritance relationships and before being combined, the newly created component would inherit from 2 different components.

**Limitations and Problems** There are a few limitations of the prototype implementation that should be mentioned:

- *Special Characters:* Rose allows the usage of almost all characters that are available in model element names. The pattern extensions does not support (, ), [, ], and | in names.
- *Correlation Management:* In certain complex cases of self-combination, it can happen that inheritance or dependency relationships are not recreated correctly in the target pattern. This does not represent a restriction, since all general cases including examples from [8] are executed as expected. If the problem occurs, it can be corrected manually.
- *Attribute vs. Association:* The combination dialog supports renaming and merging of attributes. However, attributes with class or component type are modeled in UML and Rose using associations and role names. Currently, no support is given in the dialog to rename role names.

## 9.7 Update Management

None of the concepts suggested in the paragraph about a pattern's history in section 8.2.3 have been implemented, yet. The definition of every pattern is kept in its documentation field. For successful management of updates on source

patterns, the state of every source pattern needs to be stored. The implementation effort is extensive. Therefore, update management is considered the first prototype extension.

## 9.8 Code Generation

Two options were suggested for code generation: the production of Java code and the generation of PaL source code.

**Java** The Java add-in of Rational Rose provides a code generation utility. It operates on classes and produces Java source files. Rose documentation is included in the generated files. Thus, implementation templates that were provided in pattern documentation are available to the developer.

As an example, Java code is generated for the `List` component from the *List Iterator* example in the previous section. Figure 29 contains the component in the center. The following source code is generated, documentation is not displayed:

```
package PListIterator;
public class List extends ItemAggregate {
   private ItemTemplate anchorItem;
   public List() {}
   public void add(PListIterator.Item anItem) {}
   public void remove(PListIterator.Item anItem) {}
   public void makeList() {}
   public ItemIterator createItemIterator() {}
}
```

This template can be used for implementation of the pattern and its components. However, implementation artifacts in source patterns are not available as code fragments. Therefore, developers do not profit from code development that has been done for other patterns in its refinement history.

**PaL** Implementation of PaL source generation has not been implemented, yet. All the necessary information is available; the documentation contains pattern history. The PaL source generation process differs from Java. The target file will contain the pattern definition, complete with the definition of all source patterns. The following tasks must be executed for PaL code generation:

1. Extract participating patterns from pattern definition
2. Follow pattern definition to extract all participating patterns
3. Generate code of base patterns used
4. Generate code of other source patterns in use
5. Generate code of current pattern

Because of the relatedness of pattern model with PaL and with Rose pattern support, PaL generation for patterns itself is not very complicated. A script would execute these steps for each pattern in Rose: (compare also PaL example in figures 1 and 3)

1. Create a `pattern PatternName ... end` block
2. Extract components from pattern, generate `component ComponentName ... end` block for each component
3. Extract features (attributes and operations) of components, generate feature blocks `feature featureName is ... end`
4. Use pattern interface (component with `<<PatternInterface>>` stereotype) to extract internal and external features of the pattern
5. If pattern is not base pattern without source patterns, generate block `refine SourcePattern ... end`
6. Consider component combinations and new names, add statements `rename OldCompName as NewCompName` to refine block

Documentation should also be considered. The contents od documentation fields could be added to the respective elements, similarly to Java code generation.

Implementation of operations proceeds with a separate source code editor. The next step would be to integrate code editing with Rose and to consider changes in the implementation file during the next code generation command. Thus, developers are able to implement operations and, at the same time, continue with pattern development and design, and add features and components with Rose.

# Part III
# Final Remarks

The final part of this thesis presents potential extensions to the prototype implementation. Finally, a conclusion is given that summarizes the results of this paper.

## 10 Future Extensions

The prototype illustrated the feasibility of pattern development with Rational Rose. However, a number of interesting features has not been implemented, yet.

**PaL Code Generation** PaL is the programming language that presented the capacity of the pattern model in [6]. It was used to develop the *DrawIt* application, completely based on design patterns. Section 9.8 presented the steps that need to be executed to generate PaL source. As a result, Rose would be used to visually design patterns and their relationships, refinements and combinations. If the developer decides that the design is stable, implementation of operations will proceed with a PaL source code editor, which could be started from within Rose. Updates to the Rose model are reflected during source code generation, the respective definitions are edited. Similar to the Java add-in already present, implementation already present in the source file is conserved.

**Update Propagation** Section 8.2.3 presented various concepts to handle changes to patterns that serve as source patterns in combinations. Successful implementation of these approaches permits design pattern development, which considers the dynamic nature of software development. New insights can be applied, patterns are updated, and changes are reflected in all dependant patterns. Without this feature, patterns must be designed perfectly before they could be used to construct other patterns.

**Diagrams** Rose supports a variety of diagrams to model static and dynamic properties of objects. Class diagrams represent static relationships between classes, such as associations, aggregations, inheritance, and class dependencies. Sequence, collaboration, activity, and statechart diagram are used to define dynamic behavior of classes and its objects.

The implementation of pattern combination does currently not take over Rose diagrams. Diagram support is limited to the creation of a new class diagram that contains all components of the new pattern with their relationships. In addition, support for existing diagrams is required. A wizard could guide the developer through the process, because automatic takeover is not possible. The wizard would use the pattern definition information gained from the dialog. In addition, the developer provides new names for objects that are present in dia-

grams. Eventually, diagrams are generated that recreate the information from source pattern diagrams.

**Pattern Diagrams** Relationships between patterns can best be visualized with pattern diagrams. Examples were given in [6, 8, 29] and [31]. Relationships between patterns range from general dependencies to refinement and combination. Visualizing these relations improves developer's imagination.

Since patterns have been implemented as packages in this Rose extension, only basic diagrams are supported be default. An example can be seen in figure 31, which shows dependencies between packages. It should represent that the *Iterator* pattern is a combination between *Container* and *Factory Method*. Unfortunately, arrows cannot be supplemented with names.

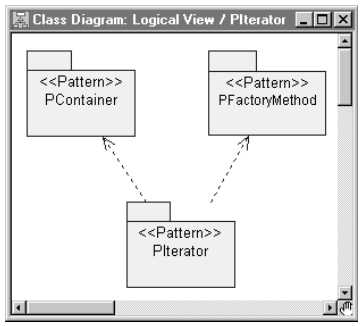


Figure 31: Class Diagram *List Iterator*

The implementation of pattern diagrams requires access to Rose internals. They would show patterns with its components and relationships as well as other patterns and relations between patterns. Commands control the level of detail. Thus, components could be masked, and only patterns are shown.

**Model Element Pattern** Finally, Rose developers could add a new model element to the system. This would require precedent standardization of design pattern in UML. Subsequently, concepts from this thesis could be applied. However, there are no indications that Rational is developing design pattern support.

# 11 Conclusions

This thesis has presented concepts to support design pattern-oriented development with Rational Rose. A prototype implementation has been developed that enables creation, filing, editing, management, refinement, and application of design patterns. Essentially, this paper is based on the pattern model and incorporates knowledge gained in development of the PaL pattern language. However, modifications and simplifications were necessary to apply the concepts in Rational Rose. Further refinement of the implementation is necessary before complete application development is possible.

Currently, Rose serves as a pattern catalogue tool. It is easy to use design patterns in new applications. Referring to pattern refinement and combination, the combination dialog serves well to reproduce design decisions. For example, patterns and combinations from [8] could be grasped without problems. However, designing a pattern combination using the dialog without having a vision of the result might be more difficult.

Extensive usage of design patterns in software development provides advantages and disadvantages. On one hand, proven solutions can increase software quality. On the other hand, pattern usage increases demands on developers, educational requirements, and complexity of development. Similar experiences will be made with pattern support in Rose. Although it provides advantages, application challenges developers.

# A Glossary

*ActiveX Control* — A reusable, stand-alone software component often exposing a discrete subset of the total functionality of a product or application. ActiveX controls cannot run alone and must be loaded into a control container such as Microsoft Visual Basic or Microsoft Internet Explorer. Formerly referred to as OLE control or OCX.

*CASE (Computer-Aided Software Engineering)* — Targets automation of software development process. Comprises methods and processes of software engineering for business applications and tool support by a software development environment.

*OCL (Object Constraint Language)* — Defines a language to describe assertions, invariants, pre- and postconditions and navigation within UML models.

*UML (Unified Modeling Language)* — Language and notation for specification, construction, visualization and documentation of the artifacts of software systems. Industry and Object Management Group (OMG) standard. Definition was led by Grady Booch, Ivar Jacobson, and Jim Rumbaugh. Supported by many tool vendors and authors. Current version: 1.3. See also [30].

*RoseScript* — Scripting language for Rational Rose. Provides access to Rose model elements. Can be used to extend Rose. Related to Basic.

Source: [15] and other.

# References

[1] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language : Towns, Buildings, Construction.* Oxford University Press, 1977.

[2] Lowell Jay Arthur. *Improving Software Quality : An Insider's Guide to TQM.* John Wiley & Sons, 1992.

[3] Heide Balzert. *Lehrbuch der Objektmodellierung : Analyse und Entwurf.* Spektrum Akademischer Verlag, 1999.

[4] Barry W. Boehm. A Spiral Model of Software Development and Enhancement. *Software Engineering Notes*, 11(4), August 1986.

[5] Grady Booch. *Object-Oriented Analysis and Design. With Applications.* Benjamin/Cummings, second edition, 1994.

[6] Stefan Bünnig. Entwicklung einer Sprache zur Unterstützung von Design Patterns und Implementierung eines dazugehörigen Compilers. Master's thesis, Rostock University, Department of Computer Science, 1999.

[7] Stefan Bünnig, Peter Forbrig, Ralf Lämmel, and Normen Seemann. A Programming Language For Design Patterns. *Informatik '99, Reihe Informatik aktuell, Springer*, 1999. Presented at ATPS'99.

[8] Stefan Bünnig and Normen Seemann. Patternorientierte Programmierung am Anwendungsbeispiel. Thesis, Rostock University, Department of Computer Science, 1999.

[9] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture : A System of Patterns.* John Wiley & Sons, 1996.

[10] James O. Coplien and Douglas C. Schmidt, editors. *Pattern Languages of Program Design.* Addison-Wesley, 1995.

[11] W. Edwards Deming. *Out of the Crisis.* Cambridge University Press, 1986.

[12] Ivar Jacobson et al. *Object-Oriented Software Engineering : A Use Case Driven Approach.* Addison-Wesley, 1994.

[13] Martin Fowler. *Analysis Patterns : Reusable Object Models.* Addison-Wesley, 1997.

[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[15] Object Engineering Process (OEP) Glossar. http://www.oose.de/glossar/, 2000.

[16] Neil Harrison, Brian Foote, and Hans Rohnert, editors. *Pattern Languages of Program Design 4.* Addison-Wesley, 1999.

[17] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process.* Addison-Wesley, 1999.

[18] Philippe Kruchten. *The Rational Unified Process.* Addison-Wesley, 1998.

[19] Craig Larman. *Applying UML and Patterns : An Introduction to Object-Oriented Analysis and Design.* Prentice-Hall, 1998.

[20] David H. Lorenz. Tiling Design Patterns — A Case Study Using the Interpreter Pattern. *ACM SIGPLAN Notes*, 32(10):206–217, October 1997.

[21] Theo Dirk Meijler, Serge Demeyer, and Robert Engel. Making Design Patterns Explicit in FACE : A Framework Adaptive Composition Environment. *ACM SIGSOFT Software Engineering Notes*, 22(6):94–110, November 1997.

[22] Gunter Müller-Ettrich. *Objektorientierte Prozessmodelle : UML einsetzen mit OOTC, V-Modell, Objectory.* Addison-Wesley-Longman, 1999.

[23] Bernd Oestereich. *Objektorientierte Softwareentwicklung : Analyse und Design mit der Unified Modeling Language.* Oldenbourg, fourth edition, 1998.

[24] Lutz Prechelt and Barbara Unger. Methodik und Ergebnisse einer Experimentreihe über Entwurfsmuster. *Informatik : Forschung und Entwicklung*, 14(2):74–82, June 1999.

[25] Darren Pulsipher. Defining and Using Design Patterns in Rational Rose, July 1999. http://www.qoses.com/ruc/Quarry/index.htm.

[26] Klaus Quibeldey-Cirkel. *Entwurfsmuster : Design Patterns in der objektorientierten Softwaretechnik.* Springer, 1999.

[27] D. Janaki Ram, K. N. Guruprasad, and K. N. Anantha Raman. A Pattern Oriented Technique for Software Design. *ACM SIGSOFT Software Engineering Notes*, 22(4):70–73, July 1997.

[28] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenson. *Object-Oriented Modeling and Design.* Prentice Hall, 1990.

[29] Normen Seemann. A Design Pattern Oriented Programming Environment. Master's thesis, Rostock University, Department of Computer Science, 1999.

[30] OMG UML. OMG Unified Modeling Language Specification 1.3, June 1999. UML Resource Center. http://www.rational.com/uml/documentation/.

[31] Sherif M. Yacoub and H. H. Ammar. Toward Pattern-Oriented Frameworks. *Journal of Object-Oriented Programming*, 12(8):25–35, January 2000.

# List of Figures

# Erklärung

Ich erkläre, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Rostock, 2000-05-31

Danko Mannhaupt

# Theses

1. Design patterns are descriptions of classes and objects that collaborate to solve a general problem in a specific context. Patterns description includes context, problem specification, solution, consequences, and examples.

2. Documenting development with design patterns can improve productivity and software quality. Common sense is a sound indicator of the usefulness of patterns in comparison to alternatives.

3. A design pattern-oriented model together with the pattern-oriented programming language PaL directly supports the notion of a design pattern, its refinement, combination, and instantiation.

4. Patterns are model elements. They can be understood as system components and are combined to build a software system.

5. To document and define design patterns, these elements are used: the pattern itself as a container of components, a pattern interface, relationships between components and between patterns, and different interaction diagrams.

6. Refinement and combination can be applied to design patterns to increase their functionality and prepare them for specific applications.

7. CASE tools provide software development assistance. Object-oriented tools apply UML. None of the current tools universally supports design patterns. The existing tools that integrate patterns with Rational Rose have several limitations.

8. Patterns are related to UML packages. Rose packages can be supplemented with UML stereotypes and used as design patterns.

9. Design pattern combination can be executed with a complex dialog that displays participating patterns, components, attributes, and operations.

10. Rational Rose can support design pattern-oriented software development. The pattern extension enables it to serve as a pattern repository. Its contents can be expanded effortlessly, pattern combination and refinement as well as pattern application is possible.