A Hybrid Model for
Object-Oriented Software Maintenance

by

Xinyi Dong

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2008

AUTHOR'S DECLARATION

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Xinyi Dong

Abstract

An object-oriented software system is composed of a collection of communicating objects that co-operate with one another to achieve some desired goals. The object is the basic unit of abstraction in an OO program; objects may model real-world entities or internal abstractions of the system. Similar objects forms classes, which encapsulate the data and operations performed on the data. Therefore, extracting, analyzing, and modelling classes/objects and their relationships is of key importance in understanding and maintaining object-oriented software systems. However, when dealing with large and complex object-oriented systems, maintainers can easily be overwhelmed by the vast number of classes/objects and the high degree of interdependencies among them.

In this thesis, we propose a new model, which we call the Hybrid Model, to represent object-oriented systems at a coarse-grained level of abstraction. To promote the comprehensibility of objects as independent units, we group the complete static description of software objects into aggregate components. Each aggregate component logically represents a set of objects, and the components interact with one other through explicitly defined ports.

We present and discuss several applications of the Hybrid Model in reverse engineering and software evolution.

The Hybrid Model can be used to support a divide-and-conquer comprehension strategy for program comprehension. At a low level of abstraction, maintainers can focus on one aggregate-component at a time, while at a higher level, each aggregate component can be understood as a whole and be mapped to coarse-grained design abstractions, such as subsystems.

Based on the new model, we further propose a set of dependency analysis methods. The analysis results reveal the external properties of aggregate components, and lead to better understand the nature of their interdependencies.

In addition, we apply the new model in software evolution analysis. We identify a collection of change patterns in terms of changes in aggregate components and their interrelationships. These patterns help to interpret how an evolving system changes at the architectural level, and provides valuable information to understand why the system is designed as the way it is.

Acknowledgements

I am immensely grateful to my supervisor, Professor Michael W. Godfrey, for his continual support throughout my doctoral studies. Michael gave me the freedom to pursue research while providing valuable guidance and advice when I needed. He substantially helped me to shape my ideas, supported me in technical and organizational issues, and reviewed the drafts of this document.

Special thanks to my dissertation committee members, Professor Daniel Berry, Professor Krzysztof Czarnecki, Professor Ladan Tahvildari, and Professor Ralf Laemmel, for their time and effort put into reading my thesis.

I sincerely appreciate the academic support of Professor Richard C. Holt and Professor Andrew Malton. I also thank many current and previous members of SWAG, in particular, Olga Baysal, John Champaign, Abram Hindle, Cory Kapser, Yuan Lin, Jingwei Wu, and Lijie Zou. I appreciate their great friendship as well as many interesting discussions that have broadened my perspective.

I am greatly indebted to my parents and my brother. Without their understanding and support, I would not have been able to come this far.

Last but not least, I thank my husband, Bing Li, who has been my best friend and help me through all of this. His patience, encouragement, and unconditional love are my greatest source of strength throughout this long journey.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software maintenance is the most expensive phase of the software life-cycle. Several case studies show that the cost of software maintenance accounts for 50% to 75% of the overall cost of a software system [53, 71]. Some even believe that as more and more software is legacy software, the percentage is growing asymptotically towards, but never at, 100%. The maintenance of poorly-documented, large-scale, legacy software is especially problematic, because necessary information for the maintenance task at hand is often outdated, incomplete or non-existent.

Maintainers can benefit from reverse engineering in that it helps them reconstruct the design information that has become lost or obscured over time. Chikofsky and Cross [11] define reverse engineering as "the process of analyzing a subject system to identify the system's components and their interrelationships, and create representations of the system in another form or at a higher level of abstraction." The ability to synthesize high-level representations is extremely desirable for the comprehension and maintenance of large software systems.

The work presented in this thesis aims at methods and tools to model large object-oriented systems at a coarse level of granularity in support of software maintenance. We have focused on the reverse engineering of object-oriented systems because object-oriented programming has been widely used when dealing with large-scale, modular and component-based software. Although the object-oriented paradigm is often claimed to support development and evolution of software through various techniques, such as data abstraction,

inheritance, and polymorphism, those features also present their own set of problems to software maintainers [105]. Reverse engineering techniques have to be customized to address the special problems that object-oriented paradigm poses to software engineers during the maintenance phrase. This includes extracting, modelling, and analyzing object-oriented systems at a higher level of abstraction than objects and classes.

1.1 The Importance of High-level Modelling

Creating, managing, and manipulating high level design views of large systems is a key aspect of software development. Recently, research in Architecture-Driven Modernization has explored ways in which design views of a system can be automatically extracted from and then kept consistent with the source code [73]. Our work concentrates on modelling high-level views of a system at varying levels of abstraction, from the class and package level up to the architectural.

Garlan describes the major impacts of a software architecture in six aspects of software development: understanding, reuse, construction, evolution, analysis, and management [27]. In this section, we will discuss the importance of high-level representations from the perspective of maintenance and evolution.

1.1.1 Aiding High-level Understanding

High-level understanding is especially important in the comprehension of large software systems. Cognitive studies show that maintainers who work with large programs usually evolve their internal mental representations — known as mental models — in a top-down manner using a pragmatic as-needed strategy [46, 88]. Instead of reading the entire source code of a system, a maintainer conjectures the goals of the program and program segments based on function and variable names, continually searches for the items of interest, and focuses only on the source code that he believes is related to the particular task at hand. During the comprehension process, he gains weak knowledge of the causal interactions between functional components. As a result, he may perform an unwary modification to one part of a system, which may in turn have a negative impact on other parts of the system.

An effective high-level representation captures the overall architecture of a system. It helps maintainers to identify the key components of the system, and reason about the interactions among them without having to descend into the source code. When maintainers focus on one portion of the system, the high-level representation provides a context for the understanding of more detailed information. Therefore, it is important for reverse engineering tools to have the capability of synthesizing representations at a higher level of abstraction.

1.1.2 Coping with the Scalability Challenge

Reverse engineering tools typically provide visualization of recovered information, since graphical representations have long been used as comprehension aids. The biggest design challenge for such tools is to capture a large amount of information using descriptive and understandable representations, while at the same time not overwhelming users with too much detail. A complex object-oriented system typically consists of hundreds of classes, which in turn may exhibit a high degree of inter-dependency among them. However, humans have limited information storing and manipulating abilities [64]. If the provided representation is too complex, maintainers may drown in the information overload and be unable to utilize the potentials of reverse engineering tools. Moreover, too much information on one diagram may degrade tool performance significantly, making for an unsatisfactory user experience [35].

Abstraction is a powerful technique in dealing with the complexity involved in the description of large software systems. Current reverse engineering tools, such as Swagkit [93] and Rigi [66], create representations of a software system at successively levels of abstraction, organize the representations in a hierarchical structure, and support navigation among different levels of abstraction. At the highest level, a maintainer can gain a concise overview of a system, while at a low level of abstraction, he can focus on the more detailed information required by a specific maintenance task.

1.1.3 Facilitating Architectural Analysis

Reverse engineering aims at recovering lost design information about the target software system. The recovered representations provide valuable information for subsequent maintenance activities. If a target system can be specified by a model at a higher level of abstraction, then new opportunities for analysis are provided.

The typical search for the big picture of a large system can bring up a clear understanding of its major components and their interactions. The big picture exposes the structural dependencies and constraints that are often unclear or hidden at a lower level of abstraction. The big picture can serve as the underlying model for performing various kind of analyses on the system, such as checking the conformance to a specific architectural style, identifying large reusable components, evaluating the design quality of a system, and assessing the consequences of change.

1.2 High-level Representations in OO Reverse Engineering

An object-oriented system is composed of a collection of communicating objects that cooperate with one another to achieve some desired goals. Objects that are similar to each other comprise classes, and class definitions provide the static description of the properties and behaviors that their instances will have. Since objects are the basic building blocks of object-oriented systems, the majority of the existing reverse engineering tools focus on modelling objects, their classes, and the interrelationships among objects.

However, when dealing with large and complex object-oriented systems, maintainers can easily be overwhelmed by the large number of objects and the high degree of interdependencies among them. A commonly used strategy to address the scalability problem is to synthesize high-level representations. As Figure 1.1 shows, there are two main approaches to creating high-level representations of object-oriented systems: One is to synthesize design- or specification-level class diagrams, the other is to create coarse-grained representations.

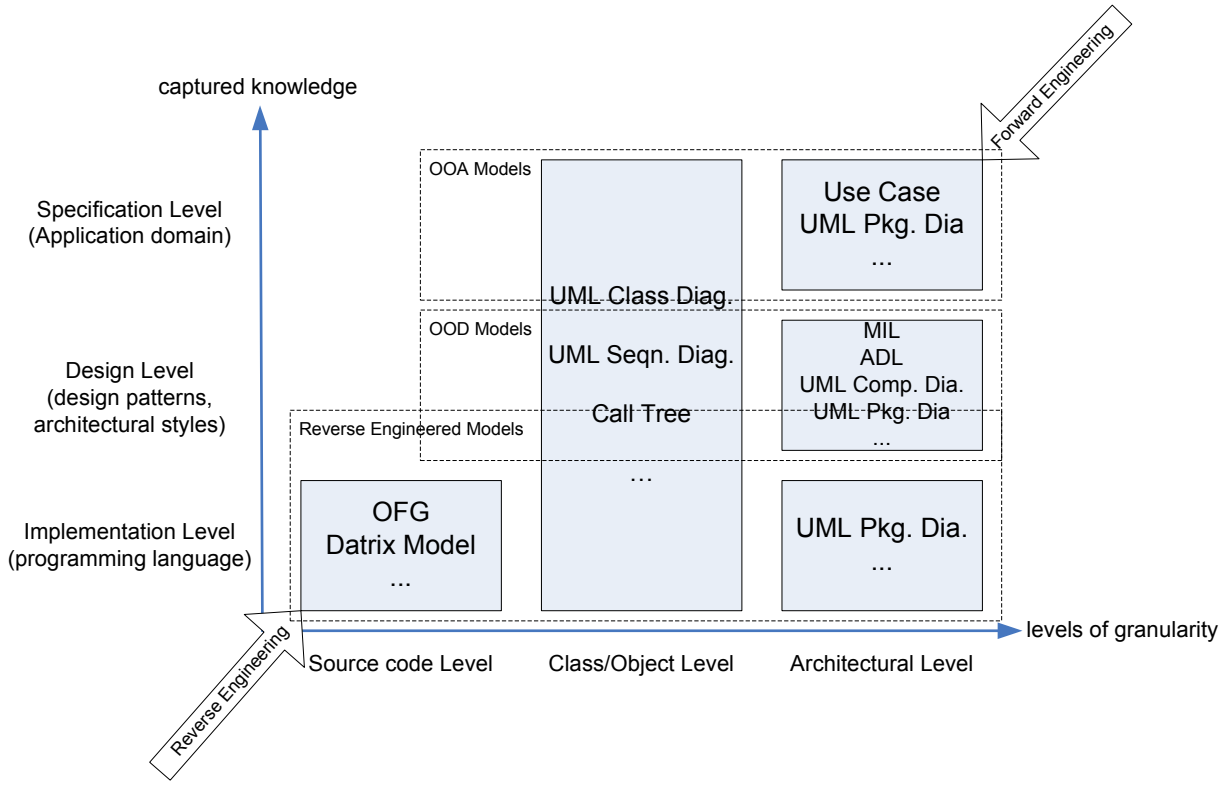


Figure 1.1: Object-oriented Representations

1.2.1 Design- or Specification-level Class Diagrams

One claimed advantage of OO development is that the developers can apply the same notation for representing objects and their relations throughout the development process. For example, UML Class Diagrams [72] can be used to describe an object-oriented system at different stages of development. However, those representations, as Figure 1.1 shows, are at different levels of abstraction, and capture different types of knowledge. OO analysis (OOA) is a process of modelling and understanding the system requirements. The main product of OOA is a model of a part of the real world. While an OOA model is in the problem domain, OO design (OOD) produce a model of the proposed system's internal construction. An OOD model describes how the system will be built without actually building it. It is often independent of the programming language used.

Reverse engineering tools often recover class diagrams from the source code to describe the structure and global behavior of object-oriented systems. The extracted class diagrams are typically at the implementation level of abstraction, and composed of programming language classes. One way to create high-level representations of object-oriented systems is to derive class diagrams at the design or higher level of abstraction.

Recovering Design-level Class Diagrams

Object-oriented design models are often documented in UML, which is a standardized object modelling language that is not tied to a particular programming language [82, 72].

For most OO programming languages, such as JAVA and C++, it is simple to infer the classes and inheritance relations that one would find in a typical class diagram. However, other relationships, such as associations, are much harder to infer from code. In UML, the semantic meaning of a relation may be modified by applying adornments, such as aggregation, navigability, multiplicity, and role name, to its ends. None of them are the first-class constructs in most object-oriented programming languages. As a result, it is difficult to precisely derive design-level class diagrams from source code.

Some reverse engineering tools, especially research tools, seek to bridge the semantic gap between UML class diagram and programming language. IDEA [47] and RevEng [98] are able to conjecture the existence of associations based on the presence of weakly typed containers; Pilfer [92] is able to identify associations, as well as aggregation semantics,

based on the mappings between the constructs of the UML class model and the C++ idioms; Ptidej [33] analyzes both static and dynamic information of Java programs to infer *use*, *association*, *aggregation*, and *composition* relationships.

These tools may help recover some lost design information. However, there is no standardized way of mapping between UML design models to source code: a given design model might be reasonably implemented in any of several ways, and a given implementation might reasonably by modelled by several different UML design diagrams. Furthermore, the recovered UML class diagram is still at a low level of abstraction, and unable to describe the whole system in a concise view.

Slicing Class Diagrams

Objects in analysis models often have counterparts in design models and source code. If the key objects that model real world concepts and their interrelationships can be extracted from the source code, then it is possible to reproduce models of object-oriented systems at the design, and even the specification level of abstraction.

Most reverse engineering tools, such as Rose® [81] and Together™ [96], allow maintainers to choose the classes that they are interested in and show the relationships among the selected classes. However, reasoning based on an incomplete class diagram can be error prone, as maintainers are unaware of the implicit dependencies that exist through unselected classes.

Egyed and Kruchten addressed this problem by proposing 60 rules, which are used to simplify a class diagram by excluding unselected classes and replacing them with transitive relations between remaining classes [20]. Such an approach relies heavily on maintainers' expertise in the application domain. Whenever maintainers change the level of abstraction using a different set of key classes, a new class diagram has to be created from scratch.

Furthermore, the transition from OO analysis to design may not be as smooth as it might seem. Several researchers believe that an OOA model cannot simply become an OOD model because the two models represent the aspects of a system that are inherently distinct [39, 44]: the OOA model is an abstraction of real world, while the OOD model reflects design decisions necessary to create a practical implementation of the abstraction. For any system, if they happen to be similar, then it's an explicit decision of the designer

that could have gone another way.

Detecting Design Patterns

A design pattern is a standard solution for a common problem in a given context. It is often described together with the problem it concerns and the trade-offs it must balance. If the solution part of a design pattern is identifiable from source code, then it is possible to deduce design intention and to understand what factors designers may have considered. Based on this assumption, a number of patten detection approaches have been proposed [2, 34, 36, 48, 70].

Most pattern detection approaches aim to identify a set of well-known patterns, especially, the twenty-three design patterns defined by the Gang of Four [26]. Researchers predefine a collection of search criteria, which formalize the constraints that a program must satisfy to make it an instance of a design pattern. However, the efficacy of design pattern recovery tools is variable for several reasons: patterns are often hard to encode precisely, exist in many possible variations, and may depend on run-time structures that are not statically identifiable. Furthermore, while identifying pattern instances can help maintainers to recognize and better understand design decisions that have been make, it does not typically provide a high-level overview of system design and evolution.

1.2.2 Coarse-grained Representations

Another approach to creating a high-level representation is to partition fine-grained entities into containers, which act as coarse-grained proxies for their contained entities. Such an approach can be applied recursively to produce a set of representations in a hierarchical structure.

Coarse-grained representations have already been widely adopted in reverse engineering for procedural programs because they provide an efficient way to recover the top-down functional decomposition. Swagkit [87], Rigi [66] and RMTool [67] decompose a program into a containment hierarchy and show it at different levels of granularity.

Some reverse engineering tools borrow the similar clustering approach to create UML Package Diagrams [72] for describing object-oriented systems at a coarse-grained level

of abstraction [40, 68, 80, 84, 99]. They group classes into packages, and then lift the interrelationships between classes in different packages to become package dependencies.

This approach has several advantages. First, a package diagram can be automatically generated from source code, as long as the containment hierarchy is given. Second, a package diagram improves the readability of an implementation-level class diagram, and provides a big picture of how source code is organized. Third, it is simple to customize existing reverse engineering tools for extracting, analyzing, and presenting package diagrams.

The biggest disadvantage of this approach is that package diagrams provide little help in design recovery. A coarse-grained representation is useful if it allows maintainers to focus on a small number of coarse-grained entities while omitting their internal details. However, when using package diagrams to understand an object-oriented system, maintainers cannot ignore the classes and their relations that are hidden within packages because a package cannot meaningfully represent its contained classes.

A class can be seen as a namespace that encapsulates a set of attributes and the set of operations performed on the attributes. More importantly, a class defines a type of objects, and represents the entire collection of the objects of that type [72]. There are two basic kinds of relationships between classes: inheritance and usage dependencies. An inheritance relation allows a class to inherit attributes and operations from another class. A usage dependency, such as *calls*, between two classes indicates the possible relations between the objects they represent. Since an object of a class is also polymorphically an object of the ancestors of the class, usage dependencies must be interpreted in the context of a class hierarchy.

A package is not a type but a namespace. It does not represent the objects that its contained classes represent due to the presence of cross-package inheritance. When classes are grouped into packages, classes become indistinguishable. Hence, knowledge about class hierarchies is not easily accessible at the package level, and cross-package usage dependencies no long connote the dependencies between objects because of the loss of interpretation context. As a result, a package fails to capture the important properties of classes as types of objects at a coarse-grained level.

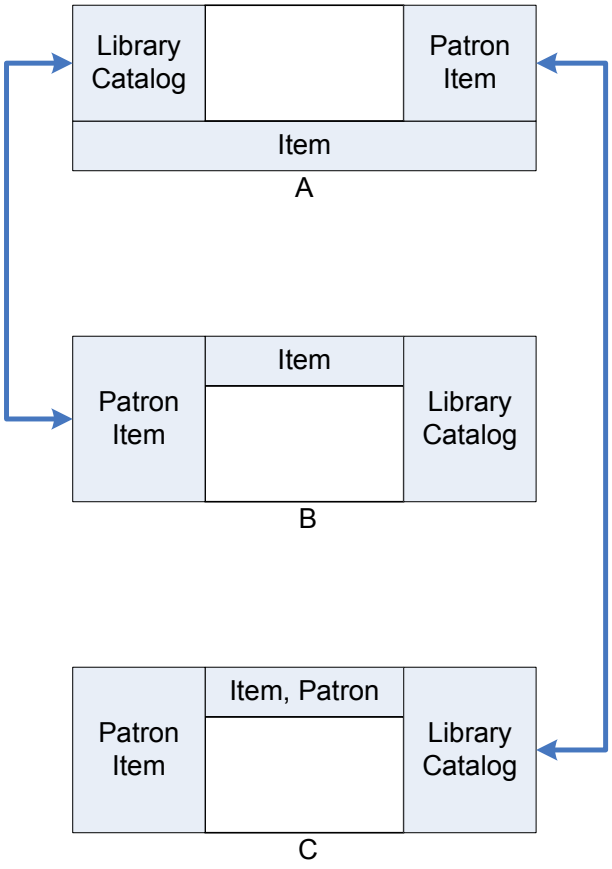


Figure 1.2: The Hybrid Model of a Library System

1.3 Thesis Statement

In this thesis, we propose a new program model, called *Hybrid Model*, for describing object-oriented software systems at a coarse level of granularity. A Hybrid Model is derived from a package diagram, and preserves the original package containment hierarchy, but it differs from the package diagram in that it summarizes the essential properties of classes as both namespaces and the types of objects.

Figure 1.2 shows an example Hybrid Model, which is derived from the package diagram, shown in Figure 2.4, of a library management system. Each node of a Hybrid Model represents a package, as well as the collection of objects that can be instantiated from the

concrete classes of the package. On the boundary of each package, we explicitly describe the classes that the package inherits from others (top boundary), and the classes that the package is expected to be extended by others (bottom boundary). In addition, we describe how the objects the package represent use (right boundary), and are used by (left boundary) objects outside the package. The edges of a Hybrid Model represent the communication path among objects.

We argue that

> *The Hybrid Model is useful in reverse engineering large object-oriented software systems. It provides a foundation for reverse engineering analysis at a coarse level of granularity.*

The goal of this thesis is to show the usefulness of Hybrid Models in software maintenance and evolution. In particular, we develop several approaches and techniques that make use of Hybrid Models to assist:

1. Program Comprehension. A Hybrid Model captures the essential properties of an object-oriented system on successively higher levels, and supports program comprehension at various levels of granularity. At a high level, the Hybrid Model describes the structural and behavioral information of packages that are externally visible to others. This information is useful for understanding a package as a whole, analyzing its responsibilities, and mapping it to real world concepts. At a low level, a Hybrid Model provides necessary context for a selected scope, and allows maintainers to study one package at a time. With the Hybrid Model, maintainers can comprehend an object-oriented system top-down, bottom-up, or both.

2. Architectural Analysis. A Hybrid Model explicitly captures cross-package inheritance relations and all possible usage dependencies between packages. It enables us to perform dependency analysis at the architectural level. We identify a selection of patterns based on the external properties of packages. Those patterns help better understand the overall structures of packages, the nature of their interdependencies, and the application of key object-oriented concepts, such as data abstraction, inheritance, encapsulation.

3. Evolutionary Analysis. The difference between the Hybrid Models of successive versions of a software system provides not only an overall picture of the system evolution, but also an opportunity to investigate the detailed structural change in a selection scope at a preferred level of granularity. Using the Hybrid Model, we identify a collection of change patterns, which help capture and better comprehend architectural evolution of object-oriented software systems.

1.4 Contributions

The contribution of this thesis can be summarized as follows:

1. Identify the important properties of an object-oriented program at coarse-grained levels.
2. Provide a coarse-grained program model notation to represent object-oriented systems at system level.
3. Provide tool support for constructing, analyzing and visualizing the new program model.
4. Support a divide-and-conquer strategy to deal with the complexity in program comprehension.
5. Provide a means to analyze the architecture of object-oriented programs.
6. Facilitate evolutionary analysis at the architectural level.

1.5 Organization of the thesis

This thesis is structured as follows:

- In Chapter 2, we present an overview of the state-of-the-art in model creation for object-oriented systems. We study existing coarse-grained representations used in the analysis and design stage of the software development process, and describe current

approaches to extract those representations from the source code, as well as the advantages and drawbacks of those solutions.

- In Chapter 3, we argue that a novel program model is needed to capture the essential properties of an object-oriented system, and we define such a model, called Hybrid Model. We present the notation of the Hybrid Model and elaborate its constructs.
- In Chapter 4, we apply the Hybrid Model to aid program comprehension and remodularization using LSEdit system as a case study.
- In Chapter 5, we use the Hybrid Model to analyze object-oriented systems at the system level of abstraction using Apache Ant as a case study.
- In Chapter 6, we apply the Hybrid Model to analyze the evolution of Apache Ant.
- In Chapter 7, we summarize the main contributions of our work and describe possible future work in this research field.

Chapter 2

Background and Related Research

This thesis concerns creating high-level design models of object-oriented software systems. The topic is not new. In fact, models are the major products of both forward and reverse engineering. Forward engineering may be defined as "the process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system." [11] Reverse engineering, on the other hand, typically starts from an existing implementation, and seeks to recreate representations of software systems at the design, architectural, or even specification level of abstraction.

This chapter presents the state of the art in representations for object-oriented systems. We particularly focus on coarse-grained representations, since they play an important role in modelling large software systems.

Structure of the chapter. The chapter is organized as follows. Section 2.1 presents some key object-oriented concepts. Section 2.2 discusses several programming abstractions that are coarser than classes. Section 2.3 review several influential descriptions of large systems at the architectural level of abstraction. Section 2.4 discusses existing representations used in reverse engineering. Finally, we discuss the problems in the area of model creation for reverse engineering purposes.

2.1 Key Object-Oriented Concepts

An object-oriented system is typically composed of a collection of communicating **objects** that cooperate with one another to achieve some desired goals. Each object has identity, state, and behavior. It can be thought of as a black box, which receives messages, processes them, and provides services that other objects require.

Similar objects form **classes**. According to UML 2.0 specification, a class is both a namespace and a type [72]. As a namespace, a class encapsulates a set of attributes and the set of operations performed on the attributes. At the same time, a class is a type of objects, and represents the entire collection of the objects of that type.

Besides objects and classes, object-oriented design is based on several powerful modelling techniques, including abstraction, encapsulation, inheritance, and polymorphism.

- **Inheritance** is the mechanism to form new classes based on existing ones. A new class (also known as a derived class or subclass) extends or tailors the properties and behaviors of an existing class (also known as a base class or superclass), while adding its own functionalities to meet special needs. With inheritance, programmers can reuse the common code among classes. A class, along with its ancestors, describes the common structure and behavior that are shared by the objects the class represents.

- **Abstraction** is the ability to deal with objects in a general sort of way, and only denotes the essential properties and methods of objects. A class is a unit of abstraction in object-oriented analysis, design and programming.

- **Encapsulation** (also known as information hiding) is the ability to separate interface from implementation. With encapsulation, each object hides its data and methods, while interacting with other objects through a deliberately designed interface.

- **Polymorphism** is the ability for objects of different types to respond to the same message in different ways. In object-oriented programming, an object of a derived class is also an object of the base class of the class. Programmers can treat the derived class's members just like the base class's members, and write generic code based on the interface of the base class. At run-time, a reference (or pointer) of the base class may refer to objects of the class or any of its inheritance descendants, and

dynamic dispatch may be used to decide which implementation of a given method should be called.

2.2 Coarse-grained Entities

Object orientation is a paradigm of software development that models the problem and its solution in terms of a collection of objects. Both analysis and design focus on identifying objects, and modelling their interrelationships. Various models can be created to show the static structure, dynamic behavior, and run-time deployment of these collaboration objects, such as UML class diagrams [72], sequence diagrams [72], etc. However, classes and objects are too fine-grained level to model large-scale systems. OO methodologists address the scale issue by inventing more coarse-grained entities for modelling software systems at a higher level of abstraction.

Existing coarse-grained entities in object-oriented analysis and design can be roughly classified into two categories: containers and conceptual entities. Both are composed of a collection of classes, but are introduced to software development for different purposes. A container is mainly used to organize classes, while a conceptual entity is a unit of abstraction in analysis and design.

2.2.1 Containers

A container is a mechanism for organizing identified classes. Containers are often introduced after the classes are well understood. OO methodologists often give general guidelines of how to partition classes. For example, the classes that change together belong together, and classes that are not reused together should not be grouped together [59]. However, these guidelines typically do not specify how containers should communicate with each other. Therefore, a containers contains a collection of abstraction units, but the container need not be a design abstraction, or represent real-world entities. Examples of containers include Coad and Yourdon's Subjects [13], Booch's Class Categories [6], and UML Packages [72].

Coad and Yourdon's Subjects

Coad and Yourdon create subjects based on initial OOA investigations. When there are too many classes in an analysis model, subjects are added to the model by promoting the classes at the top level of the class hierarchy. The main purpose of subjects is to guide readers through a large model. A class may be placed in more than one subject for the navigation purposes. Although the authors suggest that a large system can be decomposed into subjects just as a problem domain is decomposed into sub-domains, they provide little discussion about the relationships between subjects.

Booch's Class Categories

Booch introduces class categories for modularization of object-oriented systems. A class category is defined as "a cluster of classes that are themselves cohesive, but are loosely coupled relative to other clusters." Each class category represents an encapsulated name space, which can contain other class categories, and use other non-nested class categories. A *using* relationship between two class categories indicates classes in one category inherit or use classes in another. The idea of class categories is an informal one. Booch does not explicitly define what the structure or interface of a class category might look like, although he does state that it should be possible for a class category to enforce visibility constraints on its contained elements with respect to the outside world.

UML Packages

A UML packages is a construct that can contain a set of any UML model elements of the same kind, such as use cases, classes, and other packages. A package diagram describes the organization of packages and their elements. When used to organize classes, a package represents a namespace, and a package diagram provides a visualization of the namespaces.

In UML, there are three types of relationships between packages: *PackageMerge*, *PackageImport*, and *Dependency*.

- A *PackageMerge* relation indicates the model elements in the target package are merged into the source package. Model elements in the target package that do not

have a corresponding element with the same name in the source package are simply copied into the source package. If two elements have the same name, the the characteristics of both elements are merged together into one elements.

- A *PackageImport* relation indicate that model elements from the source package can refer the elements within the target package with unqualified names.
- A *Dependency* between two packages indicates that the source package depends on — that is, has semantic, structural, or syntactic knowledge of — the elements in the target package.

2.2.2 Conceptual Entities

Conceptual entities are typically used in top-down decomposition. Each conceptual entity is a unit of abstraction in analysis or design, which can be further decomposed into a collection of classes or objects. A conceptual entity hides its implementation details, and communicates with others through its explicitly defined interface. Examples of conceptual entities includes Wirfs-Brock et al. Subsystems [106], De Champeaux's Ensembles [15], UML Composite Objects [72], and Components in architectural representations (Section 2.3).

Wirfs-Brock et al. Subsystems

Wirfs-Brock et al. introduced subsystems to their responsibility-driven design. A subsystem is defined as a group of classes or other subsystems collaborating to accomplish a set of related responsibilities. Subsystems can be used in both bottom-up and top-down design. In bottom-up design, subsystems can be identified through inspection of collaboration diagrams, while in top-down design, subsystems allow designers to start with the system responsibility, partition system into parts, and assign responsibility to each part of the system. Wirfs-Brock et al. also stated that a subsystem is a kind of UML component [65]. Thus, a system can be represented using UML component diagrams, which will be discussed in Section 2.3.3, along with other architectural representations.

De Champeaux's Ensembles

De Champeaux uses Ensembles to partition large systems in order to support a top-down object-oriented analysis method. An ensemble encapsulates a group of objects or other ensembles that "naturally go together — usually because they participate in whole-to-part relationships". It hides the details of composing objects that are irrelevant to outside the ensemble, and acts as a proxy that forwards messages to external objects and ensembles.

An ensemble class is the class of ensembles. It allows designers to define the types of ensembles, instead of individual ensembles. In another word, ensembles are analogous to conventional objects and ensemble classes are analogous to conventional classes, especially those that make use of composition or aggregation.

Composite Objects

A composite object is an object that is composed of other objects. UML 2 introduces a new model, the composite structure diagram to describe the internal structure of a composite objects, and explore the collaborations that this structure makes possible [72]. The diagram provides an alternative to model the whole-part relations that is more powerful that UML class diagrams.

Figure 2.1 shows an composite structure diagram that depicts an instance of FTP session. It is composed of the following UML constructs.

- **Parts**, which are the instances of classes that participating in an internal structure. For example, ftpVisitor represents the instance that is owned by a containing classifier instance visitor.
- **Ports**, which are the externally visible properties of containing classifier instances. For example, ftpIn and ftpOut define the interface between the classifier Computer and its environment.
- **Connectors**, which represent communication links between parts. For example, FTPConnection binds visitor and host together, allowing them to interact at run-time.

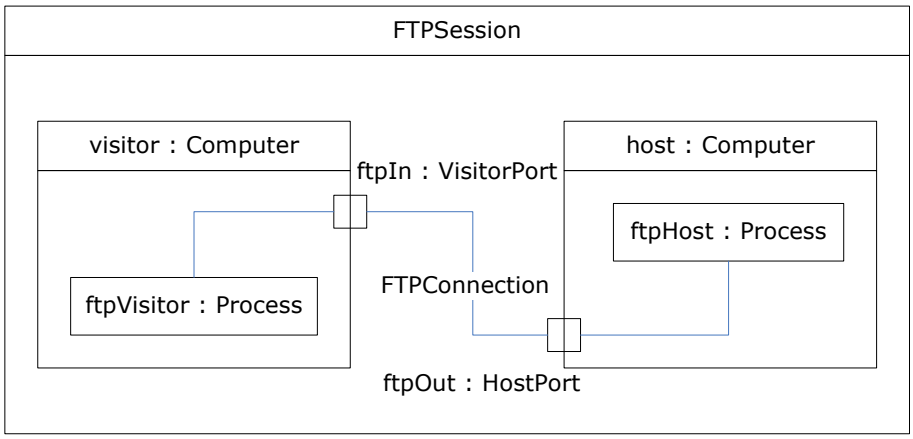


Figure 2.1: An Example Composite Structure Diagram (Adapted from Reference [5])

In addition, a composite structure diagram may contain **collaborations**, which defines the cooperation between instances, indicating the objects and the roles that they take within the collaborations.

2.3 Representations in Architectural Design

Architectural design is vital in the development of large software systems. It divides a system into components, and allows multiple teams to work on different parts of the system at the same time. Architectural design has become an important part of object-oriented design. Many OO methodologists claim that components should identified before the objects [32]. In this section, we review several influential descriptions of large systems at the *architectural* level of abstraction.

2.3.1 Module Interconnection Languages

In their foundational paper on Module Interconnection Languages (MILs), DeRemer and Kron distinguish programming-in-the-large (PitL) from programming-in-the-small (PitS). PitL focuses on the the composition of large systems out of modules, while PitS is concerned about the implementation of the individual modules. The authors believe that a separate

language, other than the traditional programming languages, should be used to accurately record the overall architecture of software system. They proposed such a language, called MIL75 [17].

A MIL75 specification describes a large system as an inverted tree. The root represents the whole system. A system is decomposed into modules, which can be further decomposed into sub-modules. Another key construct of MIL75 is resource. A resource is any entity that can be named in a programming language, such as a variable, a procedure, or a type.

A MIL75 specification also includes the description stating the relationships among modules and resources. These relationships can be divided into three categories.

- The *contain* relationship captures the hierarchical relation between modules and submodules.
- The function of the modules are described in terms of *define*, *provide*, and *require* relationships among the modules and the resources.
- The *access* relationship specifies the channels for resources flow among the constituent modules of the system.

A MIL75 specification can be checked for consistency by a compiler using static type-checking at an intermodule level of abstraction. For example, a resource can only be defined in one module; a provided resource comes from either the module or one of its submodules; the actual usage of resources by module conforms to the access channel.

Most modern views of software architecture are strongly influenced by the MIL75 approach. That is, a software system consists of a hierarchical containment tree of programming entities, such as procedures, variables, *etc.*, and their containers overlaid with relationships between those entities, which in turn induce relationships between their respective containers.

2.3.2 Architecture Description Languages

Architecture Description Languages (ADLs) are used to formally represent the architecture of software systems. A number of ADLs have been developed, primarily by researchers, to

allow for formal and unambiguous descriptions of architecture. Examples include C2 [62], Darwin [56], MetaH [4], Rapide [54], UniCon [86], Weaves [30] and Wright [1].

In their survey of existing ADLs, Medvidovic and Taylor summarized the key concepts that an ADL must be able to explicitly model about an software architecture: [63]

1. **Components** — The functionality of a system is divided into smaller computation units called components. Some ADLs utilize formal semantics to define component behaviors. Example formal specification theory includes Communicating Sequential Processes (CSP) [37], partially-ordered event sets [55], and the Z notation [90]. The others capture component behaviors as an implicit part of the component interfaces and the collaboration with others.

2. **Interfaces of components** — A component's interface specifies the services that the component provides. It is composed of a set of interaction points, through which the component communicates with others. All ADLs explicitly model a component interface besides the component that exhibits the interface. Most ADLs distinguish between provided and required interfaces.

3. **Connectors** — The connectors model the interrelationship among components. Connector semantics, as well as the rules that the connector must obey, can be used for reasoning about the behaviors of attached components.

4. **Architectural configurations** — Architectural configurations, or topologies, are connected graphs of components and connectors that describe architectural structure.

Due to the usage of formal methods, ADLs are typically expressive and powerful. Their rich semantics enables extensive analysis on designed architectures. But at the same time, they require a steep learning curve. Developers typically need specialized training to understand and use ADL technology, which may explain why they have had little use outside of research.

2.3.3 UML Component Diagrams

Unlike MILs and ADLs, component diagrams of UML 2.0 [72] use a graphic representation to model software architectures. The key constructs of a component diagram are

components and connectors.

Like an ADL component or a MIL module, a component of a UML component diagram is self-contained. The behavior of a component is defined in terms of the services it provides to its environment as well as the services that it expects from its environment.

A component communicates with other components through a set of interaction points, called ports. Interfaces associated with a port specify the nature of the interactions that may occur over a port.

Connectors specify the links that enable communication between two components. There are two types of connectors: assembly connectors and delegation connectors.

- An assembly connector specify a usage dependency between two component. One component provides the services that another component requires.
- A delegation connector links the externally provided interfaces of a component to the parts that realize or require them.

As a part of the standard modelling language, a component diagram offers a vehicle of expression and communication among wider users. Compared to MILs and ADLs, however, a component diagram is less formal and less powerful in terms of precisely describing the software architecture [63]. The relative simplicity of UML components is also an advantage. Developers intuitively understand that they model "containers" without requiring advanced mathematical expertise to create system models. when more advanced semantics are needed in the models, they can be added using Object Constraint Language (OCL) [103]; Medvidovic and Rosenblum introduced specialized extensions to the standard component diagram, exploring OCL to specify the ADL-specific concepts [61].

2.4 Model Creation in OO Reverse Engineering

Reverse engineering tools focus on abstraction and analysis of software systems in an effort to create accurate and meaningful representations from the source code. According to the level of details, the extracted representations of object-oriented systems can be classified into three categories: code level, class or object level, and coarse-grained level.

The representations at a lower level of abstraction are often used to create the ones at a higher level of abstraction.

In this section, we use an example program to discuss how existing representations depict an object-oriented program. Figure 2.2 shows partial code of a simple library management system. This example program is also used in Chapter 3.

2.4.1 Code-Level Representations

A code-level model of an object-oriented system contains almost the same amount of information as the source code of the system. Examples of the code-level representations include Datrix Model and Object Flow Graph (OFG).

Datrix is a source code analysis tool developed at Bell Canada [49]. It uses the Datrix Model, an Abstract Semantics Graph (ASG), to model software systems. The Datrix Model of a system is semantically equivalent to the source code of the system. Thus, it contains sufficient information for any kind of reverse engineering analysis, such as control-flow analysis and program slicing. CPPX, a C/C++ fact extractor, can also produce Datrix Models from C/C++ programs [14].

Tonella and Potrich use Object Flow Graphs (OFGs) to represent Java programs [97]. Each node on an OFG represents a program location, such a local variable, a parameter, etc. An edge is present between two nodes if one location is used to define the other location. The authors perform static analysis on OFG in order to create representations at a higher level of abstraction, including UML class diagram, UML sequence diagram, and UML state diagram.

2.4.2 Class- or Object-Level Representations

Most existing reverse engineering tools seeks to create representations at the class or object level of abstraction because classes and objects are the basic building blocks of object-oriented programs. A class- or object-level Representation contains information about classes or objects, their fields and methods that are visible to others, and their interrelationship. The representations at this level of abstraction can be further divided into two categories: object-centered and class-centered representations.

```
/* From Library.java */
package A;
public class Library extends Object {
      private Hashtable fUsers;
      private Catalog fCatalog;
      public Patron login(String id, String pwd) {
            user = (Patron)fUsers.get(id);
            if (user.verifyUser(pwd))
                  ...
      }
      public void logout(Patron){...}
      ...
}
/* From Patron.java */
package B;
public abstract class Patron {
      public abstract boolean verifyUser(String pwd);
      public void search (String key) {
            ...
            Catalog mCatalog = ...;
            mCatalog.retrieveInfo(key);
      }
      public void borrow(Book bk) {
            if (bk.okToBorrow(this))
            ...
      }
      ...
}
/* From Faculty.java */
package C;
public class Faculty extends Patron {
      public void verifyUser (String pwd) {...}
      public void borrow(Video vd) {
            if (vd.okToBorrow(this))
            ...
      }
      ...
}
```

```
/* From Catalog.java */
package A;
public class Calalog {
      public void retriveInfo(String key) {
            ...
            while (iter.hasNext()) {
                  Item it = (Item) iter.next();
                  it.showDetails();
            }
      }
      ...
}
/* From Item.java */
package A;
public interface Item {
      public void showDetails();
      public boolean okToBorrow(Patron p);
}
/* From Student.java */
package B;
public class Student extends Patron {
      public void verifyUser (String pwd) {...}
      ...
}
/* From Book.java */
package B;
public class Book implements Item {
      public void showDetails() {...}
      public boolean okToBorrow(Patron p) {...}
      ...
}
/* From Video.java */
package C;
public class Video implements Item {
      public void showDetails() {...}
      public boolean okToBorrow(Patron p) {...}
      ...
}
```

Figure 2.2: Source Code of an Example Library Management System

Object-Centered Representations

An object-centered representation depicts a sequence of messages that occurs among a number of objects. In object-centered representations, objects are the basic units since an object can not be partially instantiated. Those objects can be thought of as black boxes, which receive messages, process them, and provide services that other objects require. Since the behavior of an object-oriented system emerges from the interactions occurring among the run-time objects of the system, object-centered representations provide the behavioral information of the system. Examples of object-centered representations include sequence diagrams [72] and collaboration diagrams [82], and call trees [77].

Object-centered representations can be static or dynamic. A dynamic object-centered representation is extracted from the execution trace of a target system. Thus, it contains a precise description of the system's behavior during one particular run. The completeness of a dynamic representation relies on the execution scenarios chosen during extraction. Reverse engineering tools, such as CAFFEINE [33], Jinsight [77], and Fujaba [69], can produce dynamic object-centered representations to capture the behavioral perspective of object-oriented systems. Figure 2.3 shows a sequence diagram that represents the partial execution trace of the example library management system during a borrowing scenario.

A static object-centered representation is extracted from the information available during the compile time. It is intended to represent every possible run of a target system. Due to the use of polymorphism and late binding, the extracted static representations are generally conservative; that is, some of the extracted interactions may never occur in actual runs of the system. Borland Together™ can statically derive sequence diagram, which can approximate the run-time behavior of a single method [96]. RevEng can also produce static collaboration diagrams using OFG [99].

Class-Centered Representations

A class-centered representation focuses on composing classes and their relations. In a class-centered representation, a class is a software module, which is composed of a set of related variables and the methods that operate on them. A class can *inherit* attributes and methods from other classes. Some reverse engineering tools can also extract *association*, *aggregation*, and *composition* relationships between classes. As a class-centered represen-

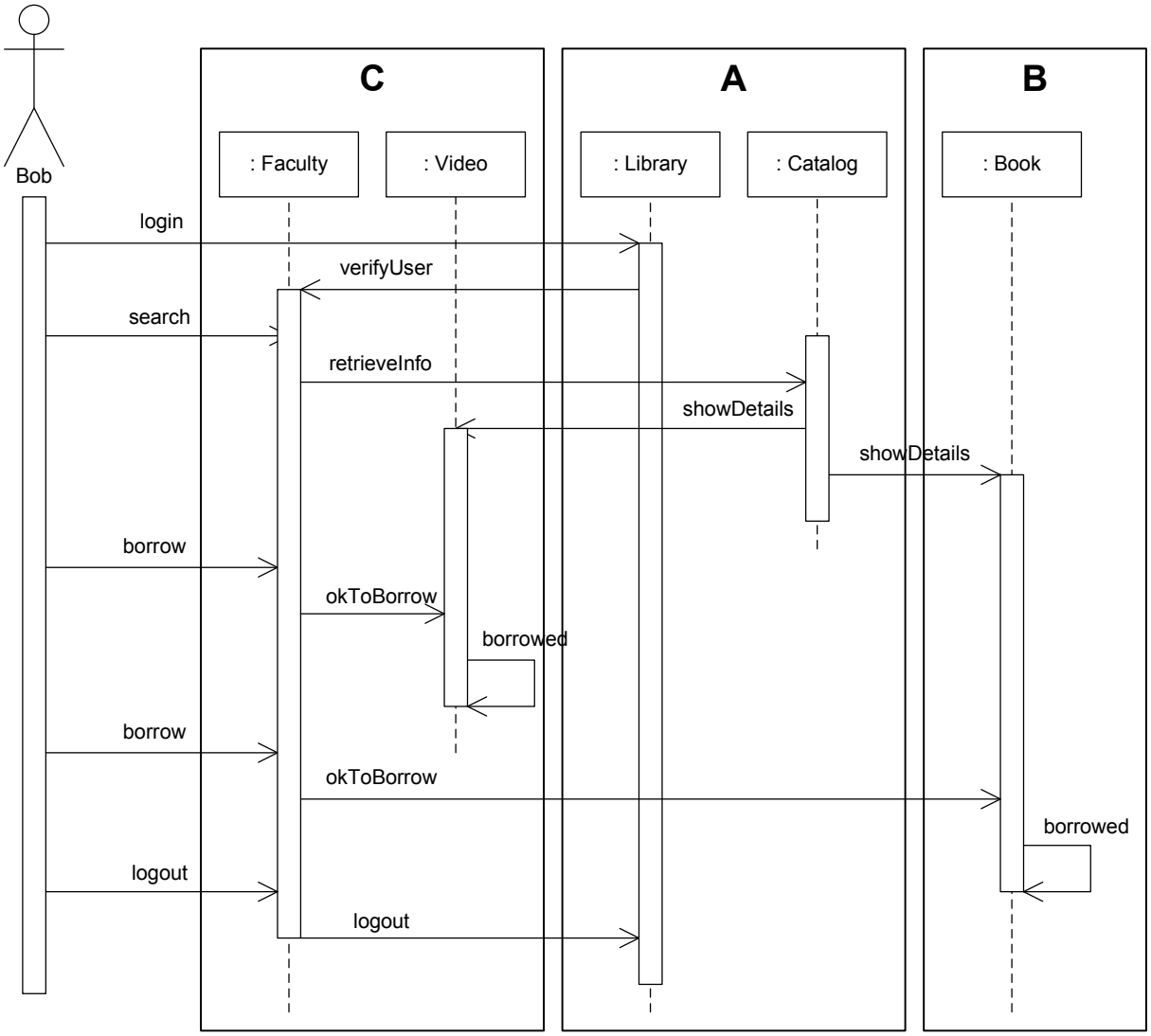


Figure 2.3: Reverse Engineered UML Sequence Diagram of the Example Library System

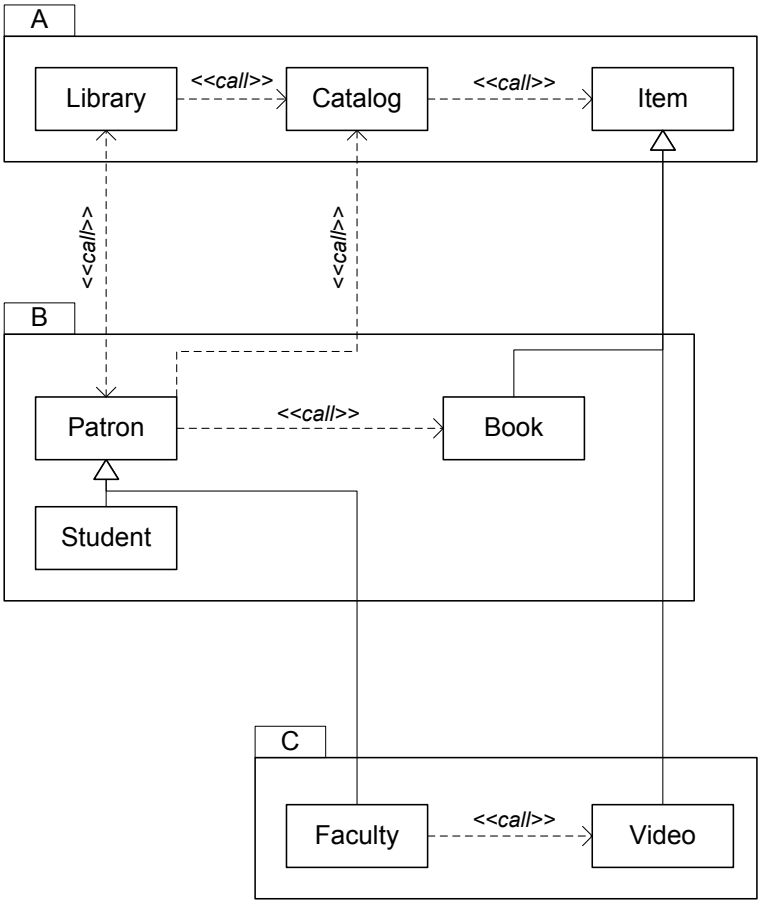


Figure 2.4: Reverse Engineered UML Class Diagram of the Example Library System

tation captures the static structure of an object-oriented system, it is the primary source of information used to acquire a logical view of the system. Examples of class-centered representations include ER-style diagrams and class diagrams [72]. Most existing reverse engineering tools can produce class-centered representations.

In addition, reverse engineering tools often use class-centered representations to describe the global behavior information of object-oriented systems. In the object-oriented paradigm, a class is a type of objects, and represents the entire collection of the objects of the type. Thus, a method invocation or variable access between two classes indicates the possible communication between the objects they represent. For example, Figure 2.4 is the class diagram of the example library management system. It depicts all possible *call* dependencies, which are viewed as stereotyped dependencies, between classes. A *call* dependency, as well as other usage dependencies, must be interpreted in the context of a class hierarchy, since an object of a class is also polymorphically an object of the ancestors of the class.

2.4.3 Coarse-grained Representations

Some object-oriented reverse engineering tools apply automated clustering techniques to generate coarse-grained representations of object-oriented systems in order to provide better readability of fine-grained representations. Since object-oriented systems are often modelled in terms of either objects or classes, there are two corresponding clustering approaches: class clustering and object clustering.

Object Clustering

In object clustering, the set of all run-time objects is partitioned into groups, and the communications between objects of different groups become the interrelationships at a coarse-grained level. AVID[102] and Program Explorer [50] support the visualization of dynamic execution trace at different levels of granularity by grouping run-time objects.

Figure 2.5 is a package diagram that summarizes the collaboration among objects at a package level of abstraction. It is generated from the sequence diagram of Figure 2.3 by clustering objects into their created classes, and then into packages. For each package, we

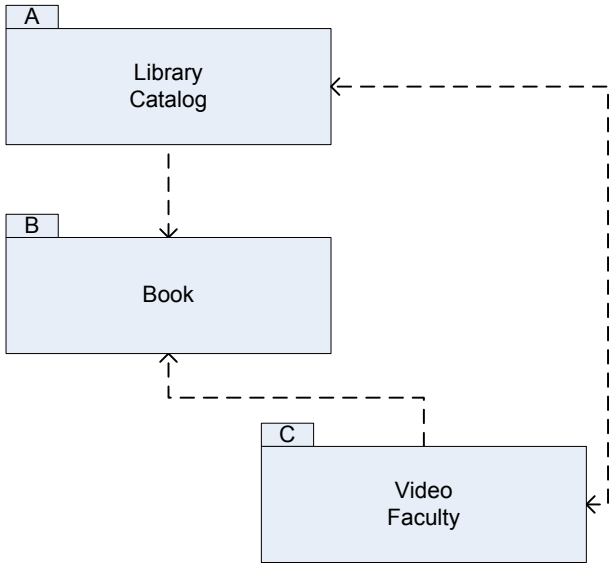


Figure 2.5: Coarse-grained Interaction Diagram of the Example Library System

list the types of the objects that are depended on by objects in other packages.

When used to organize objects, package diagrams help maintainers to understand the assignment of system responsibility. However, it can be difficult to identify the structure of a system, because classes and inheritance hierarchy are not part of object-centered representations. In Figure 2.5, there is no information about abstract classes, such as Patron and Item, because they cannot be instantiated, although they do contribute to the behavior of their subclasses, and message dispatching at run time.

Class Clustering

Class clustering combines the classes of a system into containers, typically packages or namespaces. For example, class clustering may be applied to a class diagram to create a package diagram, which is a higher-level structural model of a system. Both inheritance and usage dependencies between classes contribute to the inter-package dependencies. Most reverse engineering tools using static analysis support the generation of package diagrams [40, 68, 80, 84, 99].

Figure 2.6 depicts the package diagram of the example library management system,

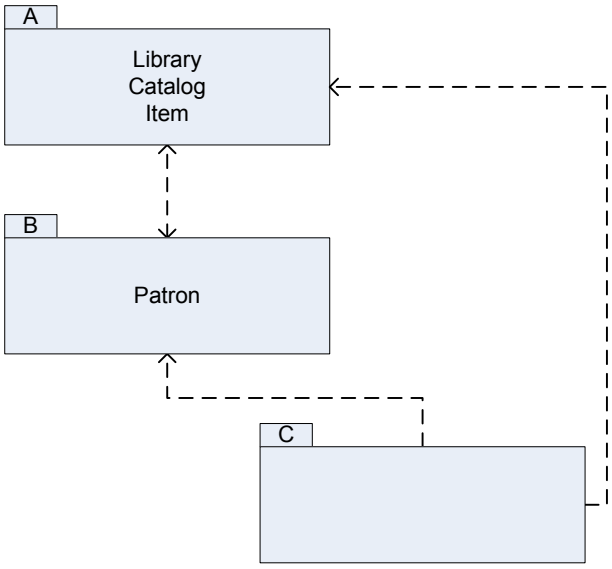


Figure 2.6: Package Diagram of the Example Library System

which was created from the class diagram of Figure 2.4 by clustering classes into packages. For each package, we list the classes that are depended on by classes in other packages.

When used to organize classes, package diagrams capture the compilation dependencies among a set of namespaces. While a package diagram provides better readability of classes and their interrelations, it harms the comprehensibility of objects as independent units, because the full semantics of an object can end up being distributed across multiple packages due to the presence of inheritance. In Figure 2.6, package A provides class Item, however, the behavior of an object Item is defined in the other two packages. It is difficult to capture the collaboration between objects using the package diagram, although the underlying class diagram does contain some behavioral information.

2.5 Discussion

Both forward and reverse engineering produce coarse-grained representations of object-oriented software systems. However, those representations are at different levels of abstraction, as shown in Figure 1.1. In software development, a large system is typically

divided into a collection of coarse-grained entities. Each entity fulfills certain responsibilities, and collaborates with others to achieve the system responsibilities. Those design-level coarse-grained entities are eventually implemented as a collection of classes in the source code. Reverse engineering tools often use package diagrams to describe object-oriented systems at the implementation level of abstraction, because package diagrams are the only coarse-grained representations that can be automatically derived from the source code.

The main purpose of reverse engineering package diagrams is to summarize the important properties of the fine-grained entities, and help maintainers to identify the counterparts of physical packages in the design-level models. To achieve this goal, reverse engineering tools must be able to capture both the structure and behavior information of a group of objects. However, for object-oriented programs, this is particularly hard because "An object-oriented program's run-time structure often bears little resemblance to its code structure" [26], and a coarse-grained representation fails to capture information from both structures.

An object-oriented system is typically composed of a collection of communicating objects that cooperate with one another to achieve some desired goals. While objects are the basic units of object-oriented systems, it is difficult to manipulate them directly as they exist only during run-time. Instead, programmers write classes to describe the properties and behaviors that their objects will have. As a result, the source code of an object-oriented program is organized in terms of classes, while the execution of the program is composed of run-time objects.

The use of OO features, such as inheritance and polymorphism, further widens the gap between the code structure and the run-time structure of an object-oriented program. Because of the presence of inheritance, full semantic description of an object can be dispersed within the inheritance hierarchy. The use of polymorphism makes it difficult to trace the communication between objects based on the static information. To acquire needed information from both the code and run-time structure of an object-oriented system, programmers rely heavily on the mapping relations between classes and objects. However, the only connection between the two structures is no longer available at the coarse-grained level of abstraction.

Due to the gap between the code and run-time structure of an object-oriented system,

neither object clustering nor class clustering can produce accurate and meaningful high-level representations that summarizes all important information about a target system. Each clustering approach promotes the comprehensibility of one structure while ignoring the other one. This make it more difficult to make use of information of both structures at the coarse-grained level of abstraction. Therefore, the traditional clustering approach is ill-suited for creating high-level abstractions of object-oriented systems.

To summarize, to recover high-level representations of an object-oriented system, we must be able to capture both structural and behavioral information of a group of objects. In the remainder of this thesis, we propose and validate a new program model of object-oriented systems, called Hybrid Model, which can be extracted from code using reverse engineering techniques and can aid software developers in performing a variety of maintenance tasks.

Chapter 3

A Hybrid Model

In this chapter, we present a new program model of object-oriented systems at a coarse-grained level of abstraction. We call the new model the *Hybrid Model*.

Structure of the chapter. We discuss the important design decisions we have made about the new model (Section 3.1). Then we introduce the new mode, and interpret its the notation (Section 3.2). We also present the approach to construct Hybrid Models (Section 3.3), and describe the means to customize Hybrid Models (Section 3.4). Finally, we compare the Hybrid Model with traditional architectural representations (Section 3.5).

3.1 Overview

Abstraction is the mechanism to omit details in order to expose the essential information that is relevant for a particular purpose. Our approach to modelling object-oriented systems applies the concept of abstraction to deal with the complexity involved in the description of large software systems. Thus, before describing the new program model, we have to consider the following questions:

- What are the objectives of the new model?
- What are the decomposition or composition units?
- What properties of the units are important and what are not?

3.1.1 Objectives

Chikofsky and Cross state that "the primary purpose of reverse engineering a software system is to increase the overall comprehensibility of the system for both maintenance and new development." [11] They list six key reverse engineering objectives: cope with complexity, generate alternate views, recover lost information, detect side effects, synthesize higher abstractions, and facilitate reuse.

We intent to create a program model for reverse engineering and maintenance of object oriented software systems. Our main focus is to automatically extract a coarse-grained level representation from source code, in order to help recover the architectural design of a target system. In particular, we seek to

- provide a concise overview of the implemented program, and improve the readability of fine-grained representations;
- summarize the essential properties of the programming language classes and objects, and form a foundation for various reverse engineering analysis at a coarse-grained level; and
- support the detection of design-level coarse-grained entities, facilitate the recovery of architectural design, and aid high-level program comprehension.

3.1.2 Units of Composition and Decomposition

Problem decomposition and solution composition is a problem-solving technique commonly applied in software development. A complex system is typically decomposed into components, subsystems, or objects. Those design abstractions focus on what they do, while hiding their detailed implementation. They collaborate with one another through their well-defined interfaces to achieve system-level responsibilities. Each design abstraction can be implemented independently into one or more run-time objects, which are manipulated through their static description organized in terms of programming language classes.

Reverse engineering is about moving backward from some physical implementation to more abstract models. To recover the design model, it would be ideal for reverse

engineering tools to group together the programming language objects that implements a design abstraction, and describe how the group communicates with others.

We consider a concrete class as the basic unit of composition and decomposition. Each concrete class not only is a compilation unit, but also represents the entire collection of the objects that can be instantiated from the class. Thus, when we divide the concrete classes of an object-oriented program into groups, we also divide all possible run-time objects of the program.

A group of concrete classes form a coarse-grained entity, called an Aggregate Component. It can be derived from a container that contains a group of programming language classes, such as a Java package, a C++ namespace, a file-system directory, or a user-defined container. However, unlike the original container, an aggregate component represents not only the container, but also the collection of objects that can be instantiated from the concrete classes that the container contains.

As discussed in the previous chapter, there is a semantic gap between objects and their static description due to the presence of inheritance. Conceptually, an abstract class contains a partial blueprint of the objects that its concrete subclasses will implement, and should be understood along with its subclasses. Therefore, while some classes may not be members of the underlying container, their presence is felt in the contributions they make to the concrete subclasses that are contained in the container. Some abstract classes, on the other hand, may indeed belong to the container, but logically they are a part of the container that contain their subclasses.

To bridge the gap between objects and their static description, we allow an aggregate component to contain the classes that are physically contained in the original container, or logically contribute to the static description of the objects that the component represents. In UML terminology, aggregate components are *compositions* of objects, and *aggregations* of classes, because an class may belong to multiple aggregate components if the descendants of the class belong to different containers.

3.1.3 Essential Properties

Our approach uses aggregate components to group objects and classes in order to yield a simplified view. The relationships of the collapsed classes and objects become part of the

interface of the aggregate components.

We treat inheritance relations and usage dependencies separately at a coarse-grained level, because they are different from each other. An inheritance relation is a structural dependency, in which one class requires another class for its full definition, while in a usage dependency, one class requires another class for its specification. An inheritance hierarchy provides the the necessary context for the interpretation of usage dependencies. As a result, the external visible properties of aggregate components come from two sources:

- Inheritance relations between classes:

  An aggregate component is derived from a container. It summarizes the inheritance relations the original container involves in terms of the classes that the container inherits from others, and the classes that the container expects to be extended by others.

- Usage dependencies between objects:

  An aggregate component serves as a proxy for the objects it represents. Its interface describes how those objects use and are used by the objects outside the component.

3.2 Notation

In this section, we present the notation of Hybrid Models in order to facilitate further discussion. Figure 3.1 graphically represents a Hybrid Model using a tree-like hierarchy, in which a container component is arranged on the top of its containee components. Alternatively, a Hybrid Model can be displayed using a nested graph, in which a containee component is nested inside its container component. For example, Figure 3.2 depicts the Hybrid Model of the example library management system.

As shown in Figure 3.1 and Figure 3.2, resources, components, ports, and connectors are the four key elements of a Hybrid Model.

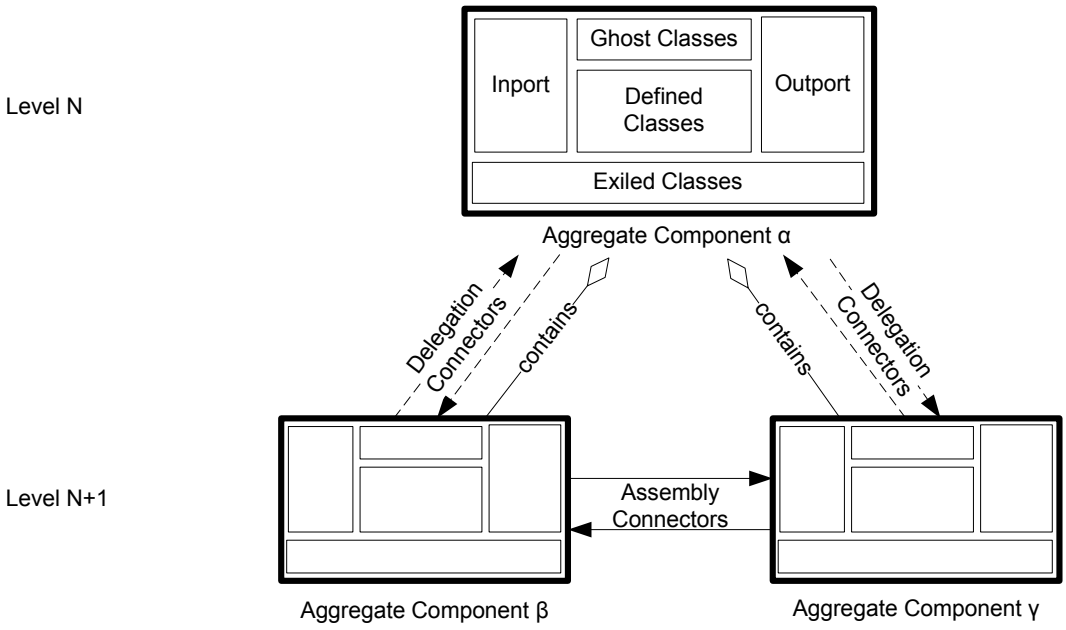


Figure 3.1: Hybrid Model Notations

3.2.1 Resource

We define a resource [17] as any entity that can be named in a programming language, such as an instance or class variable, a method, or a type. For example, login is a method resource, and Library is a type resource.

3.2.2 Component

A component [63, 72] is a logical computation unit that provides resources to its environment and may also require resources from its environment. The internal implementation is encapsulated and hidden from its environment. There are two types of components:

- A *Class Component*, *Class* for short, is derived directly from a programming language class. It is a specialized UML class that can be linked to other components via connectors. For example, Library, Book and Faculty in Figure 3.2 are class components.
- An *Aggregate Component*, *Component* for short, corresponds to a container in a

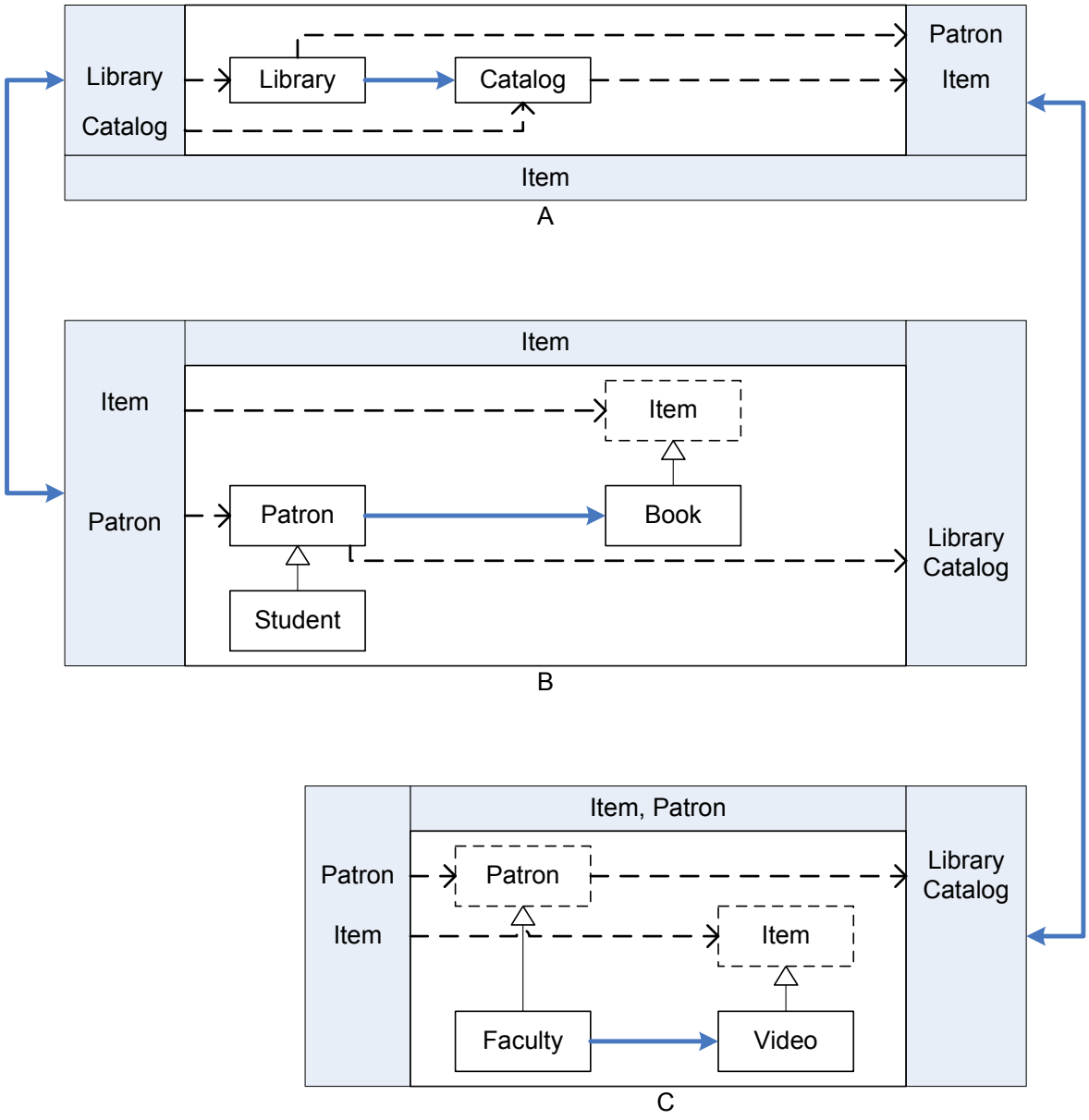


Figure 3.2: Expanded Hybrid Model of the Example Library Management System

containment hierarchy. In Figure 3.2, A, B and C are aggregate components. An aggregate component can be shown either collapsed or expanded. In Figure 3.2, all aggregate components are expanded, and their internal structures are visible. In Figure 1.2, all components are collapsed and labelled with the properties that are externally visible.

As discussed in the previous section, an aggregate component is derived from a container, and represents the collection of objects that can be instantiated from the concrete classes contained in the container. According to the physical location and the contribution to the objects represented by a given aggregate component, the classes of an object-oriented system fall into one of four categories:

- *Defined Classes*, visually represented as the solid boxes in the center of a component, are contained in the original container, and contribute to the static description of the objects the component represents. In Figure 3.2, Book is a defined class of component B, since Book is declared and defined in package B.

- *Ghost* classes, visually represented as the dashed boxes in the center of a component, contribute to the static description of the objects the component represents, but do not physically belong to the original container. The names of ghost classes are also listed on the top boundary of the component, as they reflect the cross-boundary inheritance the original container involves. In Figure 3.2, Item is a ghost class of both component B and C. It is originally declared in package A, and implemented in package B and C.

- *Exiled* classes, visually represented as the bottom boundary of a component, are contained in the original container, but make no contribution to the static description of the objects the component represents. In Figure 3.2, Item is an exiled resource of component A, since it is declared but not implemented in package A.

- *External* classes neither belong to the original container, nor contribute to the static description of the objects the component represents. In Figure 3.2, Library is an external class of component B, since it is declared and defined outside package B.

To sum it up, an aggregate component is composed of defined, ghost, and exiled classes. The defined and exiled classes of a component describes the scope of the container from which the component is created, while the ghost and defined classes of a component provide the complete static description for the objects the component represents.

3.2.3 Ports

Ports are the interfaces through which a component interacts with its environment. We distinguish two kinds of ports: inports and outports [72].

- An *inport*, visually represented as the left boundary of a component, is the interface through which the component provides resources to others. An inport represents the subset of the available resources that the component provides and are actually used by other components; resources that are provided by the component but not used by the system are not considered to be part of the inport list. In Figure 3.2, component C provides two type resources: Patron and Item.

- An *outport*, visually represented as the right boundary of a component, describes the resources that the component requires from others. In Figure 3.2, component C requires two type resources: Library, and Catalog.

3.2.4 Connectors

A connector [63, 72] specifies the interrelationship among two components. There are three types of connectors: inheritance, delegation [72], and assembly [72] connectors.

- An *inheritance* connector, visually represented as a solid line with an empty arrowhead, specifies the inheritance relation between two classes. It can exist within an aggregate component, and cannot cross component boundaries. In Figure 3.2, class Faculty inherits class Patron.

- A *delegation* connector, visually represented as a dotted arrow, links ports of an aggregate component and the ports of the components it contains. A delegation connector promotes the required interfaces and the provided interfaces of the contained

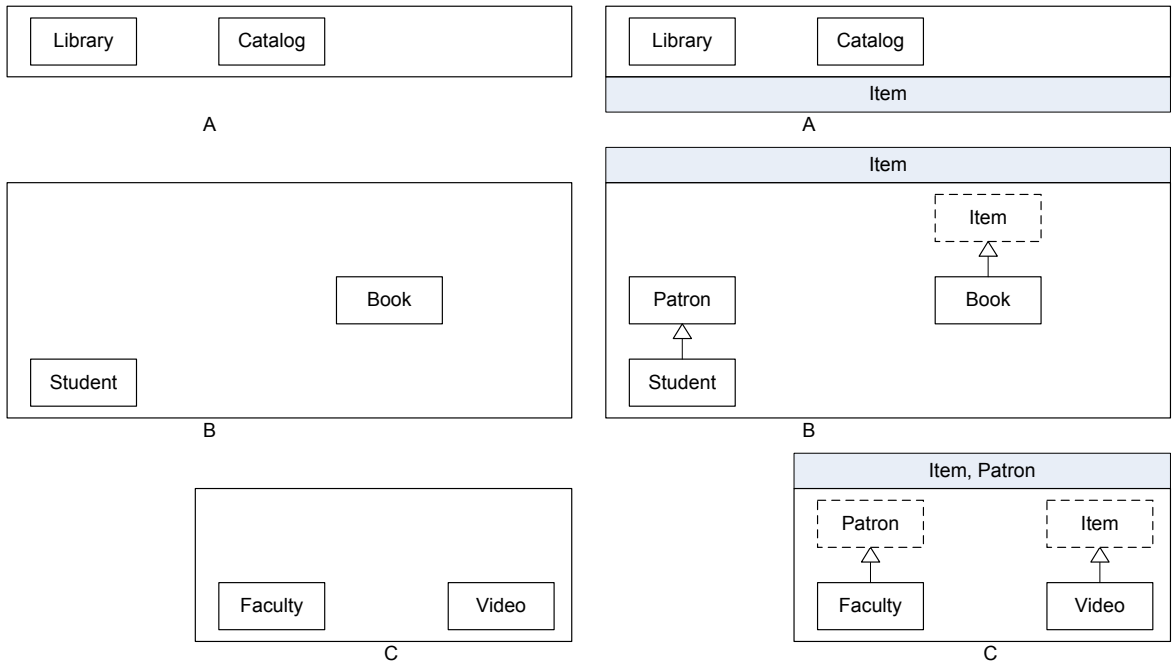


Figure 3.3: Constructing Hybrid Models

components to the corresponding interfaces of its container components. In Figure 3.2, component A provides resources that Library provides, and requires resources that Library requires.

- An *assembly* connector, visually represented as a solid arrow, specifies the client-server relationship between two components. The client component uses resources provided by the server component. For example, component A uses the resource Patron and Item provided by component B and C.

3.3 Constructing Hybrid Models

In this section, we use the example library management system to show how to create a Hybrid Model. We assume that we start with a collection of classes organized into a tree-like hierarchy of containers, such as Java packages or C++ namespaces. For example,

we may use the package containment hierarchy shown in Figure 2.4. The construction of a Hybrid Model for an object-oriented program consists of three steps.

1. All abstract classes, including Java-style interfaces, are initially removed from their containers. An abstract class is intended to be a superclass and cannot be instantiated. Conceptually, it contains partial blueprint of the objects that its subclasses represent, and should be understood along with its subclasses. For example, Item and Patron in Figure 2.4 are removed from components A and B, respectively.

2. For each concrete class in each container, all ancestors of that class are pulled into the container. An abstract class with multiple implementations in different containers will appear in each such container. As Figure 3.3 shows, Item is pulled into both components B and C.

3. Any dependencies between classes that are not in the same container become dependencies of their corresponding aggregate containers. That is, external usages and unfulfilled requirements of the parts become, respectively, usages and requirements of the whole. The interface of each aggregate container is calculated automatically from the dependencies of the contained elements. Figure 3.2 shows the Hybrid Model of the example program at the top-level abstraction.

A Hybrid Model can be extracted from any program written in an object-oriented programming language that uses static typing, such as Java or C++. Although those languages share many common features, there are also a number of differences between them. We now discuss some programming language features that may affect the construction of Hybrid Models.

- Global functions and global variables. Classes are the basic building blocks of object-oriented programs, but in some programming languages, such as C++, functions and variables can be declared to be globally scoped, outside of any class definition. When encountering a global function or a global variable, we first create a pseudo concrete class as if the function or variable were a member of the pseudo class. Then we perform the three steps to construct a Hybrid Model.

- Function and class templates. Function and class templates in C++ allows skeleton code that is parameterized by type to be used to create for different data types. At compile time, a C++ compiler generates the appropriate functions or classes for handling the individual data types. When encountering templates, we can rely on a C++ compiler to expand them before constructing a Hybrid Model; that is, we use the concrete (expanded) types in our models.

- Reflection. Reflection is a feature in Java and other programming languages that allows a running program to examine and possibly modify its runtime behavior. Hybrid Models are created from the static information available during the compilation process. Thus, it is difficult for a Hybrid Model to capture the dynamic dependencies introduced by reflection. One way to improve the accuracy of a Hybrid Model is to search for the code patterns that indicate the use of reflection, and estimate which classes or methods will be executed at run-time. However, it is impossible to recover all possible dynamic dependencies unless dynamic information is used.

- Pointers. Pointers is a powerful feature of the C++ language that allows a program to directly access objects or values that are not explicitly declared beforehand. It is difficult for naive static analysis to extract the dependencies involving pointers. More time-consuming techniques, such as points-to analysis, may be applied to improve the accuracy of Hybrid Models.

3.4 Customizing Hybrid Models

A Hybrid Model can be automatically generated from the source code, as long as the containment hierarchy is given. However, in some cases, we need additional information to make the Hybrid Model more accurately reflect the implementation.

3.4.1 Frameworks

Object-oriented frameworks have been identified by both academics and practitioners as having the potential to provide the benefits of large-scale software reuse [43, 85, 10]. A

framework is typically composed of a collection of inter-dependent classes that provide partial implementations of a set of related tasks, or a toolkit for creating complex system components, such as GUIs. Consequently, it is typical that many of the classes in a framework are abstract, since it is expected that programmers will extend and customize the parts provided by the framework. Those abstract classes and their interdependencies describe the software architecture for a family of applications in a given domain. When building an application, programmers reuse, extend, tailor frameworks to address special needs, and the semantics of the resulting system is usually tightly bound to the semantics of the underlying frameworks. Consequently, understanding such a system requires that maintainers comprehend the structure and behavior of the underlying frameworks as well.

There are three possible ways to create Hybrid Models for the applications that are built on object-oriented frameworks.

1. Import the entire source code of the frameworks as if it was a part of the application. This approach will pull in all the abstract classes defined in frameworks, as well as all the interdependencies between those abstract classes.

2. Import only the inheritance hierarchy of the frameworks. The resulting Hybrid Model contains the abstract classes defined in the frameworks, and captures the usage dependencies from the application to the frameworks. However, it ignores the interdependencies between the abstract classes pulled from the frameworks.

3. Ignore the underlying frameworks. The resulting Hybrid Model contains no information about the underlying frameworks.

The Hybrid Model resulting from Approach 1 contains the most complete information about the application, while Approach 3 leads to most concise Hybrid Model. The end users of Hybrid Models should choose from the three approaches according to their knowledge of the framework and the nature of the target application. If a maintainer has little knowledge of the frameworks or the application massively extends the frameworks, then Approach 1 or 2 is desired. If the maintainer has some knowledge of the underlying framework or knows that the information about the frameworks has little to do with the maintenance task at hand, then Approach 3 is preferred.

3.4.2 External Behavior

Reusability is one of the claimed benefits of object-oriented technology. Object-oriented systems are often designed to be reused by others, and built on library systems that are created for general reuse purpose. Thus, to understand an object-oriented system, we need to take the external links in consideration.

At the highest level of granularity, an object-oriented system can be viewed as a single aggregate component. It provides and requires services to other external applications. This information is not available in the source code, but can be specified by the end users. In the example library management system, shown in Figure 3.2, if class Book is expected to be reused by other applications, then component B should provide Book through its inport, even though components A and C do not require such a type resource.

3.5 Why Not UML Diagrams?

Hybrid Models might appear to be superficially similar to existing architecture description models, particularly UML component diagrams. One might ask: Why not use UML component diagrams instead?

UML is a general-purpose modeling language. It might be possible to customize a UML Component Diagram so that it can capture the information that a Hybrid Model represents. However, since the Hybrid Model is a special model for reverse engineering, it can represent the as-implemented architecture of an object-oriented system in a more concise and precise way. It allows us to build more efficient tools to automate the extraction, analysis, and presentation of software systems.

More importantly, Hybrid Models and UML component diagrams are different since they are designed for different purposes. UML component diagrams are typically used in forward engineering, especially component-based development [94], aiming to describe software architecture and promote software reuse. The main focus of UML component diagrams is to capture the functionalities of components, while ignoring their implementation details. Hybrid Models, on the other hand, are used in reverse engineering to represent the implementation of a design. The main goal of Hybrid Models is to help maintainers to identify what is implemented by providing information about how it is implemented. That

is, in Hybrid Models, the details are considered to be important. Because of the different purposes, Hybrid Models differ from UML Component Diagrams in key ways:

Difference in Components :

A UML component represents a modular unit, which is self-contained, independent, and substitutable. It may be created during early design, long before its internal structure is fully developed. Thus, a UML component diagram describes a design, and contains little information about how each component is implemented.

An aggregate component in Hybrid Models is created from bottom up. It summarizes the important properties of the fine-grained programming language constructs in support of design recovery. The ghost and exiled classes of an aggregate component indicate the inheritance relations that the component involves. They are part of implementation details, but are important to understand a system at a high level.

Difference in ports :

The ports of a UML component are typically associated with interfaces, or abstract classes, while the ports of an aggregate component can be associated with any named programming language constructs, such as classes, methods, and attributes.

Difference in connectors :

A connector in a UML component diagram is deliberately designed to represents a communication path between two components. A connector in a Hybrid Model, on the other hand, indicates a possible communication path between two groups of objects. As Hybrid Models are constructed based on static analysis, they may contain the connectors that are logically impossible depending on the logic of the source code.

3.6 Summary

In this chapter, we have defined the Hybrid Model for describing object-oriented software systems at a coarse level of granularity. It is aimed at explicitly capturing the essential

properties of classes as both namespaces and the types of objects at the package level. We describe the notation of the Hybrid Model, and present the approach to construct and customize Hybrid Models.

The usefulness of the Hybrid Model as a program model for software maintenance has been validated in following ways. We apply the Hybrid Model to assist program comprehension (Chapter 4). We perform architectural analysis on object-oriented systems over Hybrid Models (Chapter 5). We use Hybrid Models to explicitly model and analyze software change (Chapter 6).

Chapter 4

Applications in Program Comprehension

Summary A commonly used strategy to address the scalability challenge in object-oriented reverse engineering is to synthesize coarse-grained representations, such as package diagrams. However, the traditional coarse-grained representations are poorly suited to object-oriented program comprehension as they can be difficult to map to the domain object models, contain little real detail, and provide few clues to the design decisions made during development.

In this chapter, we apply Hybrid Models to support object-oriented program comprehension. Each aggregate component of a Hybrid Model represents a set of software objects, and contains the complete static description of the objects it represents. Thus, Hybrid Models allow maintainers to understand objects as independent units, and focus on their external properties and their interrelationships at different levels of granularity. We show the usefulness of the Hybrid Model to program comprehension by means of an exploratory case study.

Contributions. The contributions of this chapters are the following:

- The description of the comprehension process using Hybrid Models.
- The application of Hybrid Models in a real-world comprehension scenario.

Structure of the chapter. Section 4.1 introduces cognitive models, and existing approaches to support program comprehension. Section 4.2 analyzes the reasons why package diagrams are ill suited to object-oriented program comprehension. In section 4.3, we apply Hybrid Models to support both top-down and bottom-up comprehension strategy. Section 4.4 describes the tool used to create and manipulate Hybrid Models. Section 4.5 demonstrates a real-world comprehension scenario using Hybrid Models, and section 4.6 presents our conclusion.

4.1 Introduction

Program comprehension plays an important role in the process of software maintenance. It is the vital first step of most maintenance activities, such as correcting faults, adding new functionality, improving performance, and adapting to a new environment. Several studies indicate that more than 50 percent of the total maintenance effort is devoted to understanding the software to be modified [12, 21].

Research into software comprehension models suggests that programmers attempt to understand code using bottom-up, top-down, or integrated comprehension strategies. Pennington *et al.* observed that programmers build their mental models from bottom up [78]. They collect small-sized program units, such as sequences, loops and conditional patterns. Then they mentally "chunk" those structures into more generalized structures, until an overall understanding of the program is attained. Several researchers observed that programmers evolve their mental models in a top-down manner [8, 89]. The comprehension process begins with a primary "hypothesis" about program functions. The primary hypothesis is progressively refined into more specific and less functional subsidiary hypotheses until a hierarchical structure is constructed. During the comprehension process, programmers scan source code or other related artifacts, looking for beacons which indicate the occurrence of certain structures or operations. The discovery of beacons help to verify existing hypotheses, form new hypotheses, and further refine the subsidiary hypotheses. Von Mayrhauser and Vans observed that comprehension of large scale programs involves both top-down and bottom-up activities [101].

It is commonly accepted that a tool to aid program comprehension should support both

top-down and bottom-up comprehension strategy. To achieve this goal, current reverse engineering tools often assist maintainers in building abstractions through chunking from lower level program constructs, creating a hierarchy of abstractions, and exploring the hierarchy in a top-down fashion.

In object-oriented reverse engineering, package diagrams are commonly used to support the exploration at higher levels of abstraction than objects and classes. Many existing tools offer to generate package diagrams by dividing classes into packages, which act as coarse-grained proxies for their contained classes [74, 80, 84, 99]. While grouping classes into packages provides better readability of classes and their interrelations, it harms the comprehensibility of objects as independent units.

In this chapter, we utilize Hybrid Models to represent object-oriented systems at different levels of granularity. Instead of grouping programming language classes, we aggregate the complete static description of software objects, so that each coarse-grained entity of the Hybrid Model represents a set of objects. At a low level of abstraction, software objects can be understood as independent units, while at a higher level, each coarse-grained entity can be understood as a whole and be mapped to real-world objects. In addition, Hybrid Models serve as a kind of palette that allows users to mix the relationships that maintainers are interested in, and interpret them at different levels of granularity.

4.2 A Motivational Example

In this section, we use the example library management system to show why package diagrams may not satisfy the comprehension needs at a high level. Figure 2.4 depicts the class diagram of the example system, in which the method invocation relationship between classes is modelled as a stereotyped dependency. This diagram is similar to what most reverse engineering tools are capable of generating to model the system as a whole; a package diagram is created by dividing the eight classes into three distinct packages. Using the package diagram, as shown in Figure 2.6, maintainers may face the following issues during program comprehension.

First, it may not be possible for maintainers to understand — that is, create a coherent mental model of — one package independently of the others. In this example, none of the

three packages can be understood solely based on the code they contain. Item in package A is an abstract class. It is implemented by its subclasses contained in package B and C. When studying package A, maintainers can only guess the behavior of class Item based on its method signatures. On the other hand, when studying package B or C, maintainers must investigate the internal details of package A, since both packages inherit attributes or methods from class Item, and contains partial blueprint of the objects they represents. If maintainers limit their investigation in only one package, they could misunderstand the package and make ill-advised modifications to the code.

Second, the external properties of a package may not reflect the external properties of the set of objects that its contained classes represent. A package is composed of a set of classes, and thus its external properties are the properties that the contained classes export to outside packages. In this example, class Item is part of the external properties of package A, even though it is only partially implemented.

Third, the interrelationship between packages may not capture the interrelationship between objects contained in the packages. Class interrelationship, such as *calls*, must be interpreted in the context of class hierarchy, as implicit dependencies may be derived from the ones that are explicitly defined in source code. For example, in the class diagram shown in Figure 2.4, an object of Catalog could send messages to an object of Video. However, there is no communication path from package A to package C.

Due to the above issues, it is difficult for maintainers to use the package diagram to derive the design intentions, such as how the system is decomposed, how system functionalities are distributed among the packages, and what is the responsibility of each package. Therefore, package diagrams are of limited usefulness to developers seeking to acquire a high-level understanding of an object-oriented system.

4.3 Program Comprehension using Hybrid Models

With respect to comprehension, the major limitation of a package diagram is that the full definition of a class may be spread across different packages; the semantics of any given object may not be understandable by examining only the package of the defining class. If we view a class as a collection of objects with common structure and behaviors, we note

this view does not scale up to the next level. That is, a package is not a collection of objects, but a collection of partially defined types. The interrelationships among packages represent the compilation dependencies among program language constructs, but cannot be interpreted as the relations among objects or classes.

We use Hybrid Models to address the issues mentioned above. In Hybrid Models, each coarse-grained entity includes the complete static descriptions of the objects that are defined and implemented within it. Thus, an aggregate component, like a class, can be interpreted as the set of objects it represents. An interdependency between two aggregate components arises when there is a possible relation between the objects they represent.

We apply Hybrid Models to support both bottom-up and top-down comprehension strategies.

4.3.1 Supporting Bottom-up Comprehension

The main activities in the process of bottom-up comprehension are building abstractions from lower level program constructs. A tool to support bottom-up comprehension must provide such abstraction mechanisms [91].

Hybrid Models support bottom-up comprehension by aggregating classes into aggregate components. Each component summarizes the important properties of a group of classes, which in turn helps to answer questions that are commonly asked about the group, such as

- What classes does the group inherit from other groups?
- Which classes does the group expect others to extend?
- What are the responsibilities or the functionalities that the group provides?
- What collaboration does the group requires from other?

the Hybrid Model of an object-oriented system can be automatically generated as long as a containment hierarchy is given. To support bottom-up comprehension strategy, we allow the maintainers to create their own abstractions. Currently, we support three different approaches to aggregate classes:

1. Aggregate all the classes that physically belong to a given package, and name the resulting component after the package.

2. Aggregate all inheritance descendants of a given class, and name the resulting component after the given classes.

3. Select an arbitrary group of class, and name the resulting component to reflect their meaning.

4.3.2 Supporting Top-down Comprehension

Top-down comprehension involves two main activities: forming hypotheses based on the maintainers' previous experience, and reading the code in a depth-first manner to verify or reject these hypotheses. A tool that supports top-down comprehension may provide representations of a system at various levels of abstraction, and allow maintainers to explore the system in a top-down fashion [91].

Not only does a Hybrid Model captures the architecture of an object-oriented system at various levels of abstraction, it also enables a top-down divide-and-conquer exploration of the object-oriented system. At a high level of abstraction, a maintainer can focus on the external properties of aggregate components, and the connectors between them without worrying about the internal implementation. At a low level of abstraction, the maintainer can focus on one aggregate component to acquire deeper knowledge.

Overviews

With a Hybrid Model, maintainers are able to focus on the external properties of a component at a higher level of granularity. For example, component C in Figure 3.2 contains four classes. Among them, Item and Patron are known to other components. Video and Faculty provide the implementation for the two resources, but they are at a low data abstraction level and are not of great interest at a coarse-grained level. Hence, component C can be understood as a whole, and known as a service provider for type resources Item and Patron, while its internal details, Video and Faculty, are hidden.

With a Hybrid Model, maintainers are able to study all possible relations between objects in different components. As each component contains the complete blueprints of the objects they represent, the connectors between components reflect both explicit dependencies, which are explicitly defined in source code, and implicit dependencies, which are derived from explicit dependencies through inheritance. Therefore, the presence of a connector indicates a possible relation between two components, while the absence of a connector between two components indicates they are not directly connected. For example, Figure 3.2 shows that there are bidirectional communication path between component A and component B.

Context for a Detailed View

With a Hybrid Model, maintainers are able to study one component at a time. Each component represents a set of objects, and contains the complete static description of those objects. Therefore, the structure and the behavior of those objects can be understood solely based on the code contained in the component, along with the resources that component requires from outside components. For example, according to the Hybrid Model shown in Figure 3.2, the boundary of component C provides the necessary context to understand the class Video and Faculty. Therefore, to understand the component, maintainers can limit their investigation within that component.

4.4 Tool Support

We have implemented a tool, called the Hybrid Model Extractor. It provides an interactive environment in which maintainers can manipulate the containment hierarchy of an object-oriented system. Hybrid Model Extractor automatically create a Hybrid Model based on an input containment hierarchy. The resulting Hybrid Model is visualized using Graphviz [31].

A user may choose from primitive class interrelations, e.g. *calls*, *instantiates*, *aggregation*, etc., or their combinations, e.g. union, intersection, difference, etc., so that he can study various relations individually or in groups. Object-oriented program comprehension requires both structural and behavioral information. From a structural viewpoint, maintainers may focus on *aggregation* and *composition*, while from a behavioral viewpoint,

they may focus on *calls* and *instantiates*. Different relations generate different views of the Hybrid Model, but share the containment relation among components. We consider those views of the Hybrid Model as a set of layered maps. The base map is composed of components and the inheritance connectors among class components. Ports, assembly connectors and delegate connectors forms an add-on map, which is determined by the class relation under consideration. For example, Figure 3.2 shows *calls* add-on map. As the base map remains unchange, the Hybrid Model severs a platform to compare or combine different relations.

4.5 Case Studies

In this section, we present an exploratory case study to show how Hybrid Models can be used in a realistic software comprehension scenario. The subject system is LSEdit, a Java system currently under development in Software Architecture Group at the University of Waterloo. It is a part of Swagkit [93], a reverse engineering toolkit for extracting, abstracting, and exploring software architectures. LSEdit is an interactive visualization tool designed to enable users to explore and edit software landscapes in TA format [38].

LSEdit version 7.1.25 consists of total 348 classes, including 5 Java interfaces, 9 abstract classes, and 334 concrete classes. All of the classes are organized in a single package. It is impractical for a maintainer to comprehend the entire system at once, so we employ a divide-and-conquer strategy. We have chosen not to use an automated or semi-automated approach, such as software clustering [100, 104], to create the system model. Instead, we will create it manually to show how Hybrid Models can help maintainers to perform several tasks: to chunk fine-grained entities into a higher-level structure, to form meaningful hypotheses at a high-level abstraction, to confirm or reject those hypotheses using low-level information, and to derive design rationales during the program comprehension process.

4.5.1 Chunking

Initially, we knew little about the source code. We had to read through Java files, and group logically related classes into coarse-grained entities. Based on our knowledge of object-oriented paradigm and Java programming language, we knew that classes may connect

with each other through a variety of relations, such as *inheritance*, *association*, *composition*, *instantiates*, etc. With a Hybrid Model, we are able to choose relations, and perform analysis on one perspective of the system at a time.

Step 1. We started with *inheritance*, which is often the semantic backbone of an object-oriented system. The descendants of the same class often describe the variants for an abstract domain object. Therefore, it is reasonable to group the classes related through inheritance.

The base map is particularly suitable for this purpose as it only contains the inheritance among classes. Among the total 348 classes, 192 classes are not involved in any inheritance relation, 114 classes are in hierarchical trees, and other 42 classes are in hierarchical graphs due to the presence of multiple interfaces. We grouped each hierarchy tree into a component, and named it after the root of the tree. For example, the class ToolBarButton was grouped with its 18 subclasses to form a component called 'ToolBarButton'. This component can be cross-referred to a general toolbar button. For those classes with multiple parents, we subjectively assigned them with one of their ancestors according to the naming convention. For example, EntityInstance waas grouped with LandscapeObject instead of DiagramCoordinates, as its name is similar to EntityClass and RelationInstance, the descendants of LandscapeObject.

Step 2. We also examined the *inner-outer-class* relation. In Java, an inner class is a class nested in another class. An inner class is often tightly coupled with its outer class as it has access to the outer's private members. Therefore, it makes sense to group inner classes with their outer class, and map the resulting component to the domain objects that the outer class models.

We studied the *inner-outer-class* add-on map, gathered Java inner classes with their outer classes, and named the groups after the outer classes. For example, LegendBox has 17 inner classes, 2 of which are types of graphic components while the other 15 implement GUI event listeners to handle the events that occur on those graphic components. The inner classes are not useful on their own, but together with their outer class, they can be cross-referred to the 'Legend' page on the right side of LSEdit

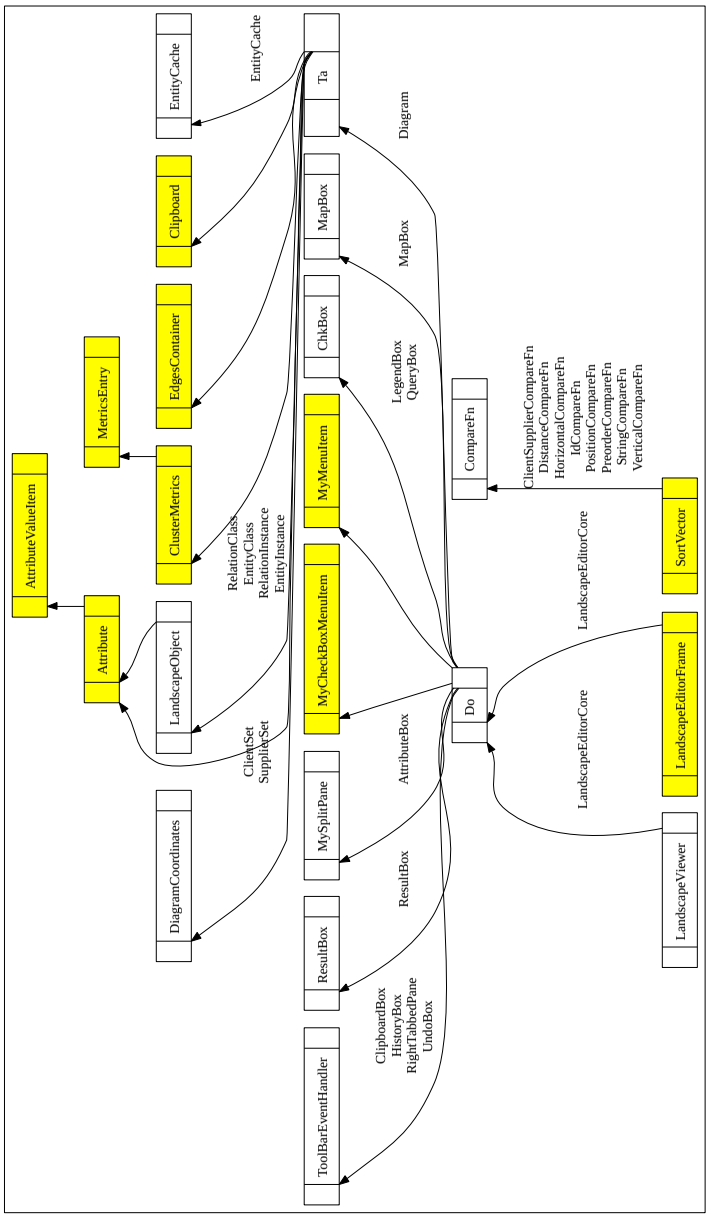


Figure 4.1: A partial view of *composition* add-on map.
Yellow boxes represent class-components, and are labelled with class names. White boxes represent aggregate components, and are labelled with user-defined names. Connectors to container-components are labelled with associated resources.

user interface. After the above two steps, we reduced the number of top-level entities to 97.

Step 3. After *inheritance*, *aggregation* and *composition* are the most important dependencies in traditional structural representations of object-oriented systems. An aggregation specifies a whole-part relationship. A composition is a special aggregation where the lifetime of the part is controlled by the whole. The group of the parts and the whole can be mapped to the complicated real-world object that the whole models.

In the context of reverse engineering, an *aggregation* can be roughly interpreted as the phenomenon that a class (whole) has an attribute whose type is the aggregated classes (part), and a composition is an aggregation in which the whole instantiates the part. Based on the above consideration, we calculated *composition*, which is the intersection of *aggregation* and *instantiates*, and created an add-on map. Figure 4.1 shows the most complicated part of the *composition* add-on map.

From Figure 4.1, we were able to identify three groups. The left bottom shows that LandscapeEditorFrame is composed of LandscapeEditorCore, whose family members, including ancestors, descendants, and inner classes, are composed MyMenuItem, AttributeBox, LegendBox, MapBox, etc. The names of those involved classes reminded us of the user interface of LSEdit. Each name corresponds to a graphic component on the screen. Therefore, it is natural to group them together to form an aggregate component representing the GUI part of the system. The classes on the right top of the figure, starting with Ta, can be grouped together because the composition relation among them is consistent with the structure of TA files in real world — a TA file specifies a typed graph (Ta) which includes the types (EntityClass, RelationClass) of nodes and edges as well as the attributes (Attribute) of those nodes (EntityInstance) and edges (RelationInstance). Finally, the right bottom of the figure shows the composition relationship among utility classes.

In addition, we reviewed other relationships, such as *instantiates*, *aggregation*, *refers*, *calls*, *static-method-invocation*, etc. As we analyzed one relation at a time, we found that the clustering result from previous steps may seem inappropriate when viewed from a

different perspective. For example, if EntityInstance is separated from LandscapeObject but is often referred as an object that provides the responsibilities of LandscapeObject, then unexpected coupling to the component that contains EntityInstance will be revealed when *calls* is examined. In such a case, we adjusted the clustering, and checked other relations iteratively.

4.5.2 Constructing Hypotheses

Once we had acquired some knowledge of the code, such as by studying Figure 4.1, we considered the system within its problem domain, and formed hypotheses about how the system's design can be decomposed. One well-known way to describe a system in an object-oriented manner is to use Class-Responsibility-Collaborator (CRC) cards [3]. A CRC card is composed of three parts: a class name, responsibilities and collaborators. Responsibilities are what the class knows and does, and collaborators are those classes that the class needed to fulfill its responsibilities. CRC cards were original introduced to teach object-oriented thinking, and later became a modelling technique that is often applied to identify classes and their interactions at the early stage of object-oriented analysis and design. Here we use CRC cards to describe our mental image of the system.

Figure 4.2 depicts our conceptual model in CRC notation. A typical application with a user interface often follows the MVC architectural pattern [9], in which the user interface is isolated from the underlying business logic, so that the data and the visual appearance of the application can be modified independently. After studying Figure 4.1, we surmised that the LSEdit system may indeed implement the MVC architectural pattern. Thus, we further conjectured that LSEdit could be decomposed into three main parts: the graph user interface (GUI), the typed graph specified in a TA file (TA), and some utility classes (UTIL).

- GUI consists of a set of smaller graphic components. It renders graphics, handles user interaction events, and dispatches some user commands to TA. It may require some common algorithms or methods from UTIL.
- TA understands the data structure of the typed graphs specified in TA files. It provides means to modify, load and store graphs. It may also require the collaboration

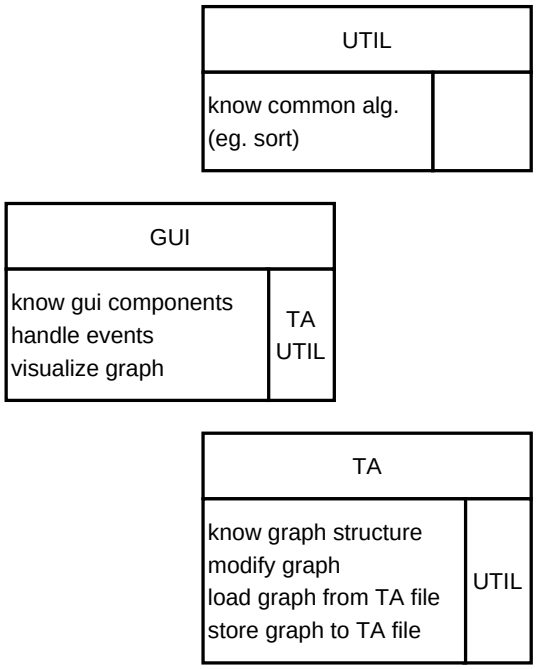


Figure 4.2: Conceptual Model of LSEdit in CRC Notation

from UTIL.

- UTIL encapsulates common algorithms or methods required from the other two components.

4.5.3 Confirm/Reject Hypotheses

In order to confirm or reject the top-level hypothesis, we need to identify the components that correspond to the three domain objects, and analyze the interaction among them. Hence, we divided all classes of the system into three regions: 223 classes for the component GUI, 96 classes for the component TA, and 29 classes for the component UTIL. Then we created the *calls* add-on map as shown in Figure 4.3.

In Hybrid Models, the responsibilities of a class component can be derived from the resources that are declared or defined in it, and the responsibilities of an aggregate component can be interpreted as the union of the responsibilities that its internal components have. The inports of components indicate the responsibilities that they reveal to the outside. Therefore, Figure 4.3 give us a peek into the responsibilities of the top-level components.

Comparing the conceptual and concrete model of LSEdit, we found that the inports of the top-level components roughly match their responsibilities in the conceptual model. The inport of the component TA includes most classes that correspond to constructs of a TA file, such as Ta, Attribute, EntityClass, etc. This indicates that the component TA has the responsibilities to maintain TA files. From the inport of the component GUI, we are able to identify some GUI related classes, such as ResultBox and ToolBarButton, the classes that deal with user input event, such as EditModelHandler, and the classes related to graph rendering, such as LandscapeLayouter. The inport of the component UTIL includes 3 classes that encapsulate common algorithms. After checking the *static-method-invocation* add-on map, we found that most classes in the component UTIL contain only static methods and are not meant to be instantiated.

The assembly connectors in Figure 4.3 describe the interaction among components, and can be interpreted as the collaboration among domain objects. All of the collaboration in the conceptual model can be identified from the concrete model. However, there are some unexpected collaborations. For example, the component UTIL requires resources from both

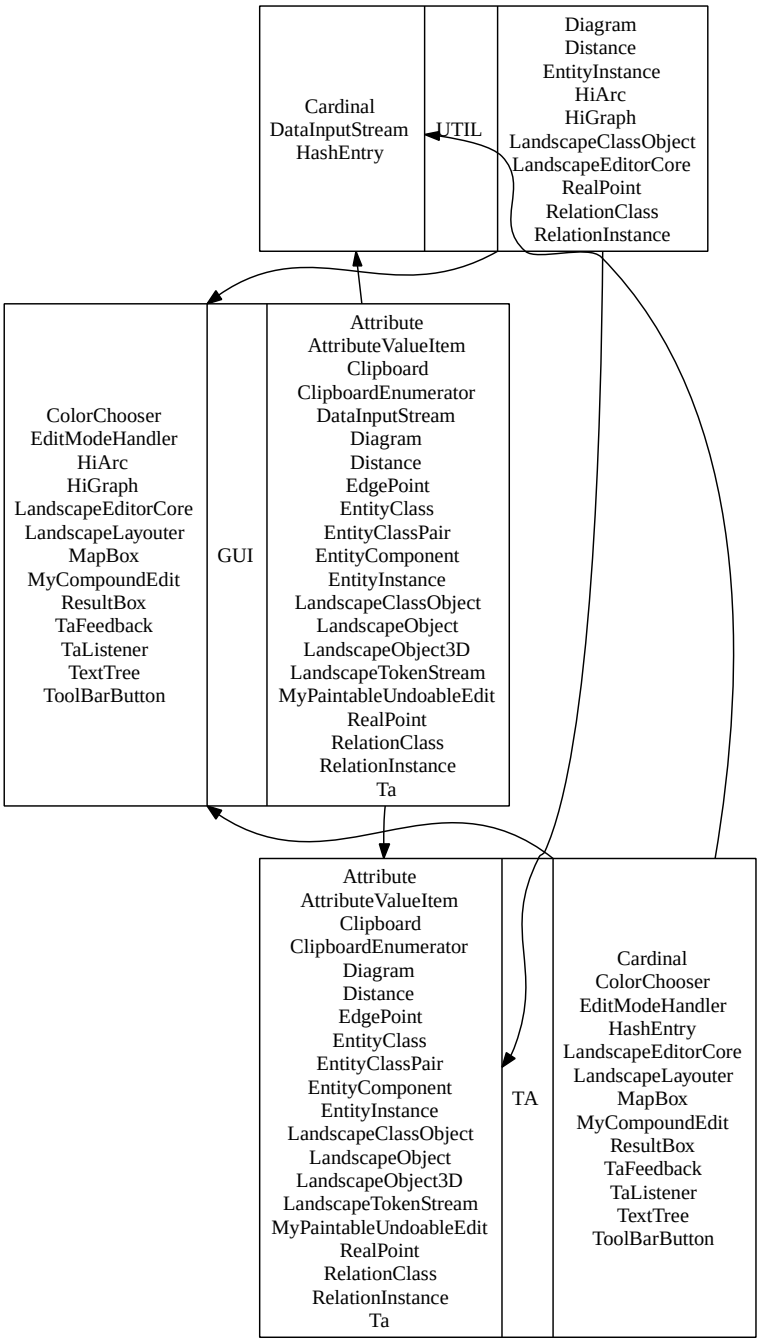


Figure 4.3: Concrete Model of LSEdit (*calls* Add-on Map)

components GUI and TA. We examined the internal structure and behavior of the component UTIL, and found that some utility classes, such as Util and SortVector, take objects as parameters and retrieve run-time data stored in those objects. Such collaboration is necessary and should be added to our conceptual model.

The biggest surprise is that the component TA requires many resources from the component GUI, especially some graphic components. This can be caused by two reasons: either the partitioning is not reasonable, or some responsibilities and collaborations are missing from the conceptual model. We studied the component TA. After examining the low-level representation, we found that the classes that model the typed graphs from TA files are not pure data. The Java class Diagram can be cross-referred to the visualized graph in LSEdit, and EntityInstance and RelationInstance correspond to nodes and edges of the visualized graph respectively. They not only keep the structure of the graph, but also are responsible for rendering the graph. Therefore, both responsibilities and collaborators of the component TA in our conceptual model should be synchronized with the concrete model. If a maintainer decides to re-architect the LSEdit system using Model-view-controller (MVC) architectural pattern, he may consider of isolating GUI related responsibilities from the component TA and moving them to the component GUI.

4.5.4 Derive Design Rationale

A Hybrid Model organizes the system in a hierarchical structure. Components at different levels of granularity can be mapped to domain objects or submodules. Therefore, navigating along the hierarchical abstraction enable maintainers to derive the rationale behind system decomposition and responsibility assignment. For example, LSEdit system is decomposed into three parts, GUI, TA and UTIL. The decomposition indicates designers' intention to separation user interface from application data. GUI is further decomposed into small graphic components according to the structure of user interface arrangement. TA is decomposed into a group of classes that model the constructs of real-world TA files.

In a good design, an object or a module reveals only the interface that clients need to interact with it. With Hybrid Models, it is possible for maintainers to deduce important design decisions about program logic encapsulation made during software development. The inport of a component in the Hybrid Model shows the resources that the component

reveals to the outside through a particular relationship. In a *calls* add-on map, the inports can be interpreted as the service that the component provides to others. By comparing the revealed functionalities with the responsibilities that a component has, maintainers are able to uncover the design rationale about program logic encapsulation. For instance, the component GUI in Figure 4.3 reveals 13 out of the 223 classes it contains. This indicates that most responsibilities this component has are not pertinent to the use of the component and are hidden from the rest of the system. The internal details of TA and UTIL further show that most classes know GUI through LandscapeEditorCore, the core of the user interface, and only a small portion of classes in TA rely on other GUI elements. If maintainers plan to further reduce the coupling between GUI and other components, they may wish to investigate how the classes exposed by GUI are used by the other two components.

A Hybrid Model also helps to derive the purpose of inheritance. An inheritance relationship can be introduced to reuse implementation, specialize behaviors, or establish a contract. Without knowing the context where the inheritance is used, maintainers can not determine the rationale behind such inheritance. The Hybrid Model, which integrates *inheritance* and *calls*, is able to provide such a context. For example, 59 classes in LSEdit system accomplish layout related responsibilities, and 11 of them are descendant of LandscapLayouter. The rest of the system interacts with this region only through LandscapLayouter. This indicates that the LandscapLayouter serves as a contract about how the region is used. For another example, Diagram is a descendant of DiagramCoordinates, Ta, TemporalTa, EditableTa and UndoableTa. Diagram is often accessed directly, and occasionally accessed as an object of DiagramCoordinates or Ta. However, it is never used as an object of TemporalTa, EditableTa or UndoableTa. Therefore, the main purpose of this inheritance is code reuse.

4.5.5 Case Study Summary

The case study presented above illustrates how Hybrid Models can aid program comprehension at coarse-grained level. With Hybrid Models, maintainers are able to map components extracted from source code to objects or subsystems in the problem domain at different levels of granularity. When a component is mapped to a subsystem, the resources attached to its inport can be interpreted as the functionalities of the subsystem. When it is cross-

referred to a complicated domain object, the resources attached to its inport indicate the responsibilities expected from other domain objects. Therefore, maintainers can think in a problem domain using an appropriate Hybrid Model.

In addition, the integration problem can be mitigated. Hybrid Models serve as a kind of palette that allows maintainers to mix the relationships that they are interested in. As structural and behavioral properties of an object-oriented system can be integrated and visualized in one single view, maintainers do not have to search for the required information from other perspectives of programs or constantly shift between different views.

However, the Hybrid Model approach has limitations. First, it is based on static analysis, and so suffers from the usual conservatism about relation information. Low-level data and control dependency analysis may help narrow the set of potential targets of polymorphic calls, and reduce the impossible relations. Second, the construction of a Hybrid Model may introduce a large number of ghost classes when a system has a deep class hierarchy and many instances of cross-package inheritance. To our experience, however, this case is very rare. Inheritance, especially implementation inheritance, often leads to tight coupling between the subclass and the superclass. They are often grouped in the same package as they are logically related, and should be understood together.

4.6 Summary

In this chapter, we have applied Hybrid Models to support the comprehension of object-oriented systems. Maintainers who attain high-level understanding of an object-oriented system can benefit from the proposed Hybrid Model. It allows maintainers to apply a divide-and-conquer comprehension strategy to deal with the complexity of large systems. At a low level of abstraction, maintainers can study one component at a time, and understand its composing software objects as independent units. At a high level, maintainers can focus on the external properties of components and their interrelationships. In addition, the Hybrid Models may help maintainers to identify domain objects at coarse-grained levels, and may provide clues to the important design decision made during development. An exploratory case study has been performed to show the usefulness of Hybrid Models to program comprehension in a realistic comprehension scenario.

Chapter 5

High-level Dependency Analysis

Summary Uncovering, modelling, and understanding architectural level dependencies of software systems is a key task for software maintainers. However, current dependency analysis techniques for object-oriented software are targeted at the class or method level; this is because most dependencies — such as *instantiates*, *references,* and *calls* — must be interpreted in the context of one or more class hierarchies.

Hybrid Models capture inheritance relations and all possible usage dependencies of object-oriented software systems at the package level of granularity. In this chapter, we propose a set of dependency analysis methods based on the Hybrid Models. We present an exploratory case study of the Apache Ant build system; we show how the dependency analysis using Hybrid Models can help maintainers capture external properties of packages, and better understand the nature of their interdependencies.

Contributions. The contributions of this chapters are the following:

- The dependency analysis approach using Hybrid Models.
- The list of architectural patterns that help understand the design at a coarse-grained level of abstraction.

Structure of the chapter. This chapter is organized as follows. Section 5.1 briefly introduces the importance of dependency analysis, and reviews the existing approaches of dependency analysis at a coarse-grained level. Section 5.2, Section 5.3, and Section 5.4

describe the dependency analysis methods over Hybrid Models. Section 5.5 presents the visualization support for the dependency analysis. In section 5.6, we apply these methods on an exploratory case study that show how maintainers can benefit from the dependency analysis approach. Finally, section 5.7 summarizes our work.

5.1 Introduction

Dependency analysis is a key activity in program comprehension and software maintenance. To a maintainer, a software system is usually modelled abstractly as a collection of entities with dependencies among them. Extracting, analyzing, and modelling these dependencies is of key importance in acquiring in-depth understanding of the structure and behavior of the software system. Many maintenance activities — such as impact analysis, code reuse, and remodularization — rely heavily on having a deep understanding of dependencies that exist between software components. Therefore, it is essential for maintainers to be able to accurately and confidently identify, analyze, and model the dependencies within a software system.

In practice, dependency analysis for object-oriented systems is often performed primarily at the class or method level of abstraction; this is because usage dependencies — such as *instantiates*, *references*, and *calls* — must be interpreted in the context of one or more class hierarchies [7]. However, dependency analysis at a fine-grained level is often unable to deal with large complex systems.

A commonly used strategy to address the scalability problem is to partition the set of all classes into packages, lift both inheritance relations and usage dependencies between classes to become dependencies between packages, and then analyze package dependencies. Sangal and Waldman use the Dependency Structure Matrix (DSM) to present package dependencies and help maintainers to identify unexpected dependencies [84]. JDepend [40] and NDepend [68] use package dependencies to calculate design quality metrics, such as afferent and efferent coupling [57]. The existing approaches ignore the difference between inheritance relations and usage dependencies at the package level, which make it difficult to analyze the design properties of key object-oriented concepts, such as abstraction, encapsulation, inheritance, polymorphism, which are vital to the quality of an object-oriented

design.

In this chapter, we propose a set of methods to analyze the dependencies of object-oriented systems over Hybrid Models. As discussed in the Chapter 3, a Hybrid Model explicitly captures the cross-package inheritance relations, as well as all possible usage dependencies between objects at the package level. It provides a basis for system-level dependency analysis, which is essential to reconstruct software architecture, provide overviews of software systems, and facilitate the maintenance activities at the architecture level. Our approach is to analyze high-level dependencies from two perspectives: the external properties of components, and the characteristics of connectors.

5.2 Component Analysis

We now describe some patterns that involve a single aggregate component. These patterns are based on the numbers and types of resources that are associated with the internal structure, the inport, and the output of an aggregate component. It is possible that an aggregate component may satisfy multiple patterns at the same time.

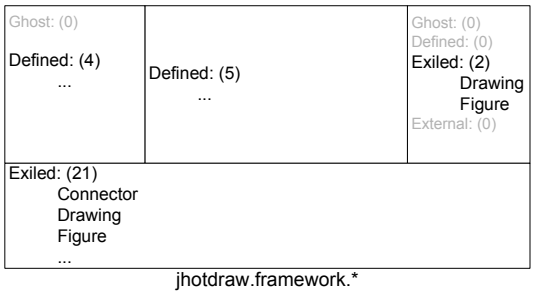
5.2.1 Internal Structure and Cross-Package Inheritance

The ghost classes of an aggregate component represent the classes that the component inherits from others, and the exiled classes of an aggregate component represent the classes that the component expects to be extended by others. Thus, the number of ghost classes and the number of exiled classes reveal the cross-package inheritance that the original package of an aggregate component involves, and provide clues for the role the component plays in a software system.

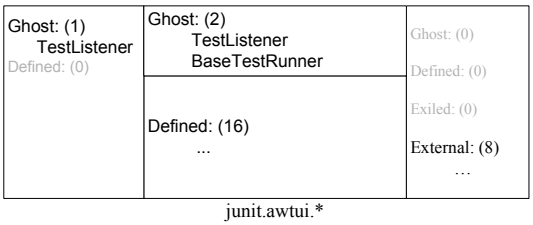
We calculate three metrics, shown in Table 5.1, for each aggregate component of a Hybrid Model. According to those metrics, we define four categories of aggregate components: **interface**, **implementation**, **mixed**, and **self-implemented**.

Interface Aggregate Component α**:** $NOG(\alpha) = 0 \wedge NOX(\alpha) > 0$

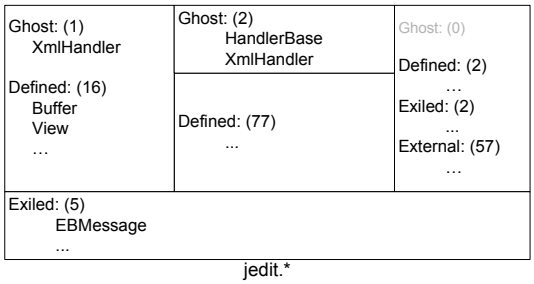
An aggregate component that contains exiled classes but no ghost classes is called an **interface** component. Such a component declares a set of classes, but provides



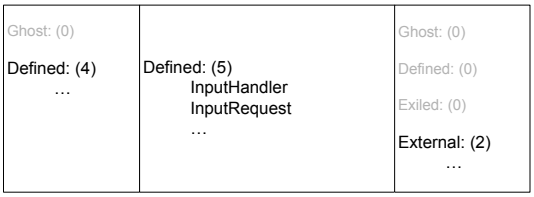
(a)



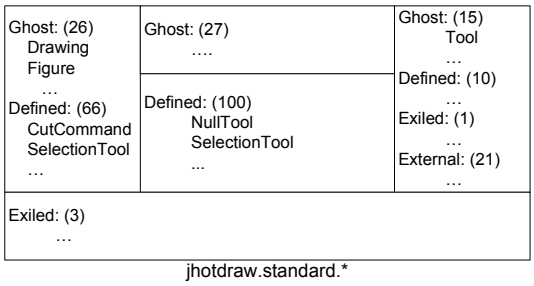
(b)



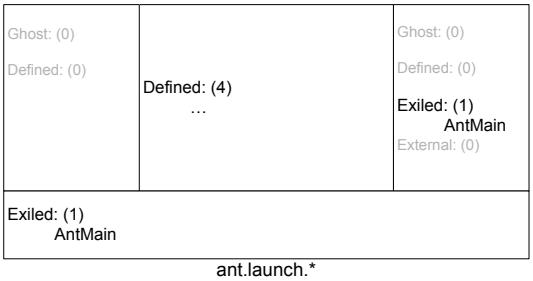
(c)



(d)



(e)



(f)

Figure 5.1: Example Aggregate Components

Name	Description
$NOG(\alpha)$	# of the ghost classes in aggregate component α
$NOD(\alpha)$	# of the defined classes in aggregate component α
$NOX(\alpha)$	# of the exiled classes in aggregate component α

Table 5.1: A list of aggregate component metrics used in dependency analysis

no implementations for them. Those exiled classes are usually deliberately designed in order to allow a group of possible behaviors. For example, in Figure 5.1(a), 21 out of 26 classes declared in jhotdraw.framework.* are exiled classes. All of them are Java interfaces. This component defines the framework for the JHotDraw program [42].

Implementation Aggregate Component α: $NOG(\alpha) > 0 \wedge NOX(\alpha) = 0$

An aggregate component that contains ghost classes but no exile classes is called an **implementation** component. It occurs when the classes within the original package inherit from classes declared outside the component. Such a component relies on the definition of its ghost classes. For example, junit.awtui.* in Figure 5.1(b) has two ghost classes. It implements a listener for test progress, and reuses the code in BaseTestRunner via inheritance.

Mixed Aggregate Component α: $NOG(\alpha) > 0 \wedge NOX(\alpha) > 0$

An aggregate component that contains both ghost and exiled classes is called a **mixed** component. Mixed components often have complicated structure. Like an interface component, a mixed component declares a group of classes, but provides only incomplete description of their behaviors. At the same time, it implements or reuses the classes that are defined in other components. For example, jedit.*, shown in Figure 5.1(c), is a mixed component. It contains the main logic of the JEdit system [41], and is coupled with many other components of the system.

Self-implemented Aggregate Component α: $NOG(\alpha) = 0 \wedge NOX(\alpha) = 0$

A **self-implemented** component has neither ghost nor exiled classes. It declares classes and provides complete description for their behaviors. A utility compo-

nent is often self-implemented. For example, ant.input.* in Figure 5.1(d) is a self-implemented component that provides resources to handle user inputs.

5.2.2 Inports and Data Abstraction

Inheritance is a key modelling tool in the object-oriented paradigm. An inheritance relation can be introduced for a variety of reasons, such as reusing implementation, specializing behaviors, or establishing contracts. Consequently, it is important for maintainers to understand the purpose of inheritance. The inport of an aggregate component reflects how a component is used by others. It provides the context where inheritance is used, and helps to derive the design rationales behind inheritance.

Both ghost and defined classes of an aggregate component may provide resources through the inport of the component. We count the number of resources associated to an inport as shown in Figure 5.2. Based on the composition of the inports, a component can be classified as **directly used**, **indirectly used**, or a combination of the two.

Name	Description
NOPT (α)	# of type resources provided by component α
NOPTG(α)	# of type resources provided by the ghost classes of component α
NOPTD(α)	# of type resources provided by the defined classes of component α

Table 5.2: A list of inport metrics used in dependency analysis

Directly-used Aggregate Component α: $NOPTG(\alpha) = 0 \wedge NOPTD(\alpha) > 0$

A **directly-used** component exports only resources provided by its defined classes. Those resources are known to other components, and thus are important at the coarse-grained level. A utility component, such as ant.input.* in Figure 5.1(d), is directly used by others.

Indirectly-used Aggregate Component α: $NOPTG(\alpha) > 0 \wedge NOPTD(\alpha) = 0$

An **indirectly-used** component exports only resources provided by its ghost classes. Those resources are usually designed to model abstract concepts in the real world.

An indirectly-used component is known to other components as the implementation of some abstract concepts. However, the classes that implement those concepts are at a low data abstraction level, and are typically of low interest at a coarse-grained level. For example, junit.awtui.* in Figure 5.1(b), is known to others as the component that provide the service of TestListener, while its implementation is not exported by the component.

Mixed-use Aggregate Component α**:** $NOPTG(\alpha) > 0 \wedge NOPTD(\alpha) > 0$

An inport revealing both ghost and defined resources often results from mixed design intentions. For example, jedit.* in Figure 5.1(c) is known to others as an abstract concept, XmlHandler, instead of a particular implementation of XML processing handler. At the same time, it also provides some defined resources, such as Buffer and View.

In practice, we have found that most aggregate components are mixed-use components. To further investigate the design intention about how a component is used, we compare the number of the resources provided by the ghost classes of a component to the number of all resources provided by the component.

$$R_{indirect}(\alpha) \quad = \quad \frac{NOPTG(\alpha)}{NOPT(\alpha)} \tag{5.1}$$

We note that $R_{indirect}$ is a numeric value between 0 and 1. If the rate of a component is close to 1, then the component is likely designed to be used indirectly.

5.2.3 Inports and Modularity

In a good design, an object or a module reveals only the interface needed to interact with it. Therefore, the size of an inport can sometime provide important clues to the design decisions about program logic encapsulation made during development [75]. According to the number of exported resources, a component can have an **empty**, **narrow** or **wide** inport.

Empty Inport: $NOPT(\alpha) = 0$

An **empty** inport does not export any resources. This pattern occurs when a component is unused, or is the starting point of the system, or accessed through other means, such as Java reflection or pointer "tricks" in C++. In the case of ant.launch.* in Figure 5.1(f), its empty inport indicates that it is the starting point of the system.

Narrow Inport: $NOPT(\alpha) \ll (NOG(\alpha) + NOD(\alpha))$

A **narrow** inport of a component exports a small portion of the resources the component contains, while keeps most resources hidden from the outside. A narrow inport implies the designers' intention to hide complicated implementation details, such as a class serves as an interface to a large, complicated component within a system. For example, junit.awtui.* in Figure 5.1(b) encapsulates 18 classes, but it exports only 1 resource.

Wide Inport: $NOPT(\alpha) \sim (NOG(\alpha) + NOD(\alpha))$

A **wide** inport of an aggregate component exports many of its defined resources to other components. It often occurs when the component acts as a library, providing general-purpose utilities that are used throughout the rest of the system. For example, ant.input.* in Figure 5.1(d) exports four of the five resources it defines; this component provides I/O related functionality to the rest of the system. jhotdraw.standard.* in Figure 5.1(e) also has a wide inport; it provides a standard implementation of the resources that are originally defined in jhotdraw.framework.

5.2.4 Outports and Reuse

The outport of an aggregate component reveals the services that the component requires from others. In a Hybrid Model, a class can belong to multiple aggregate components. Thus, an aggregate component may require resources that are already provided by its ghost, defined, or exiled classes. We classify the required resources and count their numbers, as shown in Table 5.3.

There are three patterns that describe the components regarding their outports: **self-sufficient**, **open-for-extension**, and **open-for-variation** components.

Name	Description
NORT (α)	# of the required type resources of component α
NORTG(α)	# of the required type resources of component α that are also provided by its ghost classes of
NORTD(α)	# of the required type resources of component α that are also provided by its defined classes of component α
NORTX(α)	# of the required type resources of component α that are also provided by its exiled classes of component α

Table 5.3: A list of outport metrics used in dependency analysis

Self-sufficient Aggregate Component α: $NORTG(\alpha)+NORTD(\alpha)+NORTX(\alpha) = 0$

A **self-sufficient** component requires few, if any, resources that are external to it. That is, the resources that the component defines are fully implemented within it. For example, ant.input.* in Figure 5.1(d) is self-sufficient.

Open-for-extension Aggregate Component α: $NORTX(\alpha) > 0$

An **open-for-extension** component requires exiled resources of the component. This pattern indicates that the required resources are "open for extension" [60], and could have an unlimited collection of possible behaviors. For example, jhotdraw.framework.* in Figure 5.1(a) requires the exiled resources Drawing and Figure, both of which have dozens of implementations in other components of JHotdraw system.

Open-for-variation Aggregate Component α: $NORTG(\alpha) + NORTD(\alpha) > 0$

An **open-for-variation** component requires ghost or defined resources of the component. The required resources have a number of possible behaviors. Some are defined in the components, while others are implemented externally. This pattern may exist if each derivative is tightly coupled with different classes. For example, jhotdraw.standard.* in Figure 5.1(e) is open for variation of resource Tool. jhotdraw.standard.* provides some standard tools, such as NullTool, a tool to do nothing, and SelectionTool, a tool to select figures. These tools are logically different from the additional tools defined in other components, such as text tool, zoom tool, etc., and

are tightly coupled with other classes in jhotdraw.standard.*.

We use $R_{OpenForVariantion}$ value to measure the extends to which a component is open for variation.

$$R_{OpenForVariantion}(\alpha) = \frac{NORTG(\alpha) + NORTD(\alpha)}{NOTR(\alpha)} \quad (5.2)$$

A self-sufficient component can be replaced within the system by another component that conforms to the specification of its exported resources, and be reused in another project or for another purpose. An open-for-extension or an open-for-variation component, on the other hand, cannot be reused on its own.

5.3 Assembly Connector Analysis

An assembly connector between two aggregate components exists if the classes in one component uses the resources provided by the classes in the other. It also indicates a possible communication path between the objects represented by the two components. Any changes to the server component might lead to changes in the client component. Therefore, analysis of the directions, the types, and the strength of assembly connectors between two components can help to understand how tightly two components are coupled, and what causes the dependency between them.

5.3.1 Types of Assembly Connectors

When a Hybrid Model is derived from a package diagram, classes are regrouped into aggregate components. As a result, a resource shared between the client and sever component may physically belong the package the client component is derived from (DIC), or the package the sever component is derived from (DIS), or neither (DIT). To investigate the nature of usage dependencies between components, we count each types of resources associated with an assembly connector.

Based on the types of the associated resources, an assembly connector (α, β) can be classified into four types:

Rely-on-Behavior (RB): $\#DIS(\alpha, \beta) = 0 \wedge \#DIC(\alpha, \beta) = 0 \wedge \#DIT(\alpha, \beta) > 0$

In a Rely-on-Behavior (RB) assembly connector, all associated resources are DIT resources. A RB connector links to an inport with ghost resources, because the associated resources are not originally declared in the server component, but the server component provides at least one possible behavior for those resources.

Rely-on-Declaration (RD): $\#DIS(\alpha, \beta) > 0 \wedge \#DIC(\alpha, \beta) = 0$

A Rely-on-Declaration (RD) assembly connector is associated with DIS resources and optional DIT resources. A RD connector links to an inport with defined resources, because the associated resources that are originally declared and implemented in the server component.

Provide-Structure (PS): $\#DIS(\alpha, \beta) = 0 \wedge \#DIC(\alpha, \beta) > 0$

A Provide-Structure (PS) assembly connector is associated with DIC resources and optional DIT connectors. A PS connector links an outport with defined or exiled resources to an inport with ghost resources, since the associated resources are originally declared in the client components and implemented in the server component. In this pattern, the server component either reuses the code from the client, or implements the contacts specified by the client.

Provide-Structure-Rely-on-Declaration (PSRD): $\#DIS(\alpha, \beta) > 0 \wedge \#DIC(\alpha, \beta) > 0$

A Provide-Structure-Rely-on-Declaration (PSRD) assembly connector is associated with both DIS resources and DIC resources. Some of the shared resources are originally declared in the client component, while others are originally declared in the server component.

5.3.2 Strength of Assembly Connectors

Assembly connectors in a Hybrid Model show the presence of usage dependencies between components. The common mechanisms that constitute usage dependencies between classes include: one method invokes another; one class is the type of a method's parameter, local variable, or return value; and one class is related to another by an aggregation. These

mechanisms also constitute interaction coupling and component coupling [7, 19]. Although inheritance relations in a Hybrid Model are hidden within components, a Hybrid Model of a system makes evident most coupling dependencies between components.

We measure the strength of an assembly connector using the number of its associated DIS, DIC and DIT resources.

$$SOAC(\alpha, \beta) \quad = \quad \#DIS(\alpha, \beta) + \#DIC(\alpha, \beta) + \#DIT(\alpha, \beta) \tag{5.3}$$

This number reflects how many type resources in one component are used by the other. The strengthes and the directions of the assembly connectors between two components give maintainers an idea of how tightly the two components are coupled. Generally speaking, two components are more tightly coupled if there is a bidirectional assembly connector and the numbers of associated resources are high.

5.3.3 Connectors and Package Dependencies

Classes of object-oriented programs are often organized into packages. Both inheritance relations and usage dependencies between classes contribute to the dependencies between their packages. Package dependencies provide a convenient way for maintainers to check for compilation and deployment dependencies at a high level of abstraction. It is generally agreed that a good object-oriented design should minimize the dependencies between packages, and try to avoid dependency cycles [24, 58].

Unlike a package diagram, a Hybrid Model does not directly capture dependencies between original packages since classes are regrouped into aggregate components during construction. However, most package dependencies can be easily derived from the assembly connectors. Table 5.4 shows how the assembly connectors between two components can be mapped to the package dependencies between their corresponding containers.

A rely-on-behavior assembly connector indicates no package dependencies between the original packages, because the associated resources are originally declared in a third component. A rely-on-declaration assembly connector indicates a possible package dependency in the same direction; the client component uses resources that originally defined in the corresponding package of the server component.

connector $A \rightarrow B$	connector $A \leftarrow B$	package dependency
RB		
RD		$P_A \rightarrow P_B$
PS		$P_A \leftarrow P_B$
RB	RB	
RD	RB	$P_A \rightarrow P_B$
PS	RB	$P_A \leftarrow P_B$
RD	PS	$P_A \rightarrow P_B$
RD	RD	$P_A \leftrightarrow P_B$
PS	PS	$P_A \leftrightarrow P_B$
PSRD	$*$	$P_A \leftrightarrow P_B$

Table 5.4: The relationship between assembly connectors and package dependencies. A and B are aggregate components. P_A and P_B are the original packages.

A provide-structure assembly connector implies that designer's intention to apply the "Dependency Inversion Principle" [58]. The original package of the client component contains classes at a high abstraction level, and is designed to be more stable than the corresponding package of the server component. A provide-structure assembly connector is often accompanied by a rely-on-declaration connector in the reverse direction. This is desirable in an object-oriented design, as it allows bidirectional control flow but does not cause a compilation dependency cycle between two packages.

If two components are linked with a provide-structure-rely-on-declaration connector, bidirectional rely-on-declaration connectors, or bidirectional provide-structure connectors, then there is a possible compilation dependency cycle between the two corresponding packages. To break down the dependency cycle, maintainers may consider applying "Extract Interface" [25] and "Move Class" [23] refactoring techniques.

Name	Description
SOAC (α, β)	Strength of assembly connector between component α and *beta*
SODC (α, β)	Strength of delegation connector between component α and *beta*

Table 5.5: A list of connector metrics used in dependency analysis

5.4 Delegation Connector Analysis

A delegation connector promotes the ports of the sub-components to their container component. It is associated with a collection of resources shared by the ports of the container and containee component. We calculate the strength of a delegation connector as the number of the resources associated with the connector.

The strength of a delegation connector between two aggregate components reveals how the containee component contributes to the externally visible properties of the container component. Examining all delegation connectors of a given container component, we are able to answer questions, such as

- Which containee components contributes the most to the externally visible properties of the container component?
- Which component has no contribution to the externally visible properties of the container component?
- Do the containee components equally contribute to the properties of their container, or is there a dominant contributor?

5.5 Visualization Support

We apply polymetrics visualization technique [51] to highlight the key properties of Hybrid Models. The goal of visualization support is to help maintainers to identify patterns listed in the previous sections, and gain insights of the design at a high level.

As Figure 5.2 depicts, the height of a component is determined by the size of its ghost classes, defined classes, and exiled classes, while the width of a component is determined by

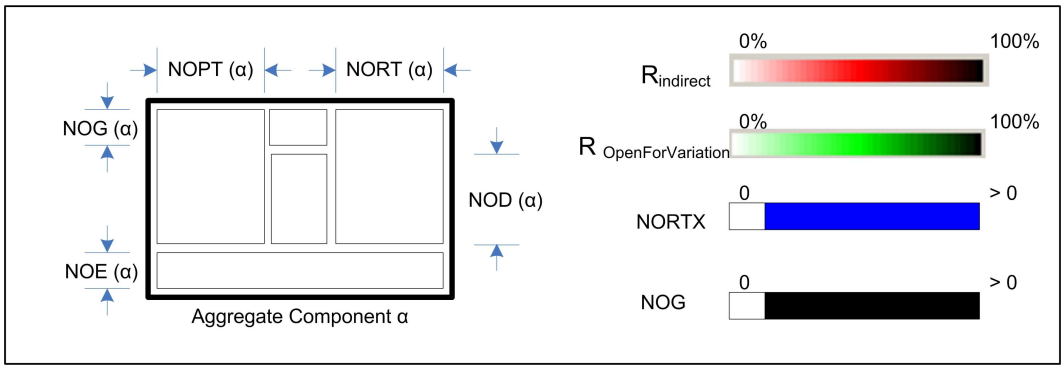


Figure 5.2: Visualization Support for Dependency Analysis

the size of its ports. The color of the parts of an aggregate component are determined by the important metrics of the component, which help identify patterns involving a component.

In addition, we use the color shade of edges to indicate the strength of a delegate connector, and use different styles of edges to represent the different types of assembly connectors.

5.6 Case Study

In this section, we present an exploratory case study to show how maintainers can benefit from dependency analysis at the architectural level. Our choice of case study was Apache Ant [22], a Java-based build tool. It was selected for the case study because it is a medium-sized object-oriented system that is in wide use. Apache Ant version 1.6.5 consists of approximately 170,000 lines of code, 70 packages and 1014 classes, including 64 Java interfaces, 62 abstract classes, and 888 concrete classes.

5.6.1 A Big Picture of Apache Ant

In this case study, we used a prototype tool that we built to extract static information from Java class files, construct Hybrid Models based on the package containment hierarchy, and automatically create visualizations of the Hybrid Models using GraphViz [31].

Figure 5.3 depicts the Hybrid Model of the Ant system in a hierarchy structure. The

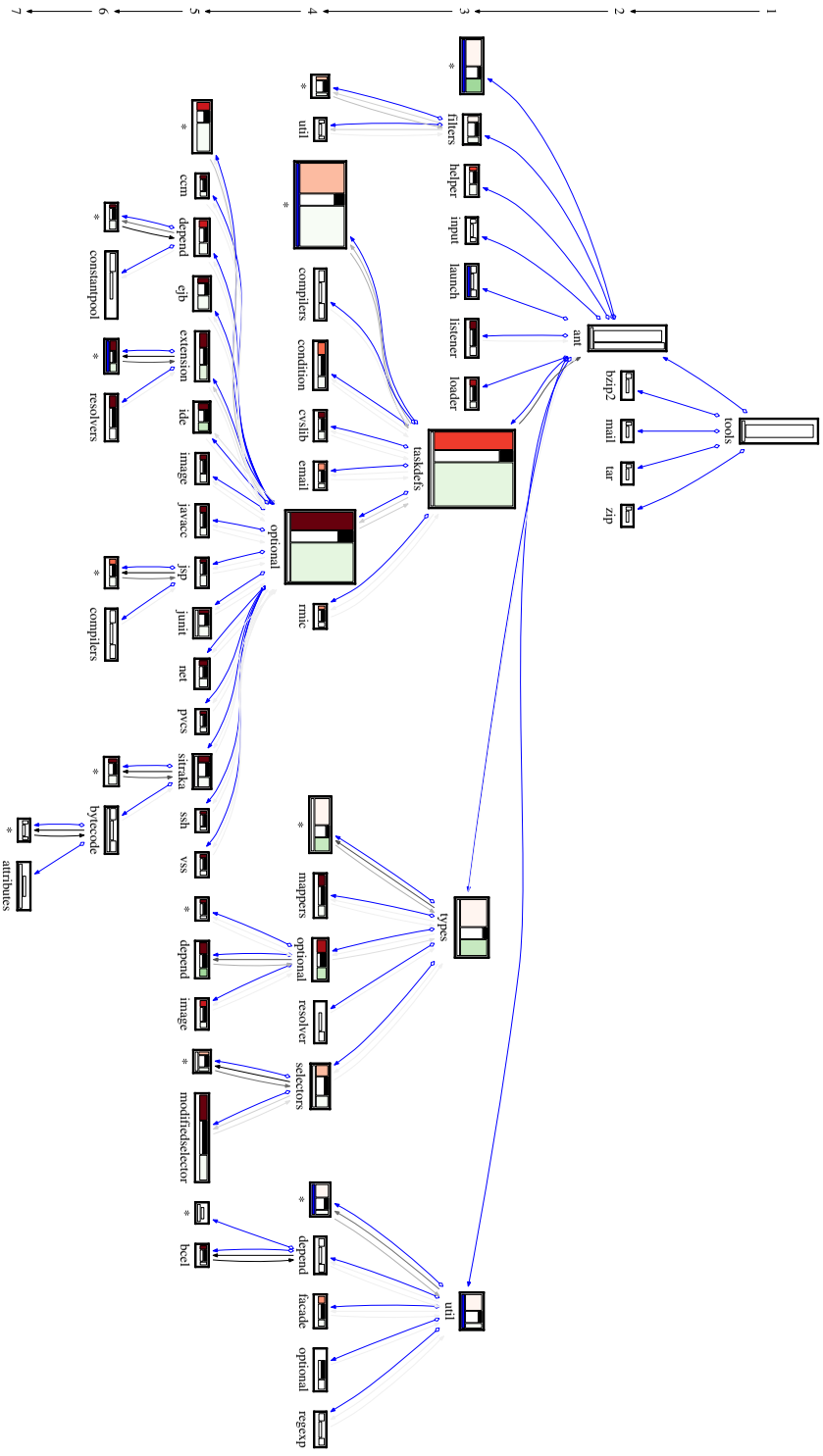


Figure 5.3: The Hybrid Model of Apache Ant 1.6.5.

color shades of delegation connectors are determined by their strength. This figure provides a big picture of the Ant system. With the big picture, we can detect most patterns of individual aggregate components. According the color of the inports, we found that component optional at the fourth level is used indirectly by others. It is composed a group of sub-components, which are in similar shape. Most of them are also used indirectly by others, and equally contribute to the external visible properties of component optional. Two components at the third level, types and util, are both composed of a set of sub-components in different shapes, and both have one dominant contributor to their external properties.

Figure 5.3 shows no assembly connectors, but we can study assembly connectors by focusing our attention on a selected level of granularity. The third-level Hybrid Model of Apache Ant is composed of 10 components and 45 assembly connectors between them. During the construction of the Hybrid Model, 55 abstract classes (41 distinct classes) are duplicated in 8 aggregate components. This indicates that the Ant system has many instances of cross-package inheritance. Some ghost resources, such as BuildListener and EnumeratedAttribute, specify the services that their components provide. Others, such as AbstractSelectorContainer and AbstractAnalyzer, are not exported by their components. Thus, it is very likely that their descendants inherit those classes for the purpose of code reuse.

A total 178 type resources are shared among the 10 third-level aggregate components. 5 of them provide more than 80% of those shared resources, and compose the backbone of the Ant system. Figure 5.4 depicts the interrelation among the five components. Component ant.* appears to be the framework of the system, because it contains more exiled resources than any other components. Moreover, almost half of the resources it requires are originally defined in its corresponding package. ant.taskdefs is an implementation component. It exports more ghost resources than defined resources. Thus, its corresponding package defines classes at a low data abstraction level. ant.types, ant.util, and ant.filters can be utility components because they all have wide inports and export about half of their defined resources. 20 assembly connectors, including 8 PSRD connectors, among them show that the five components are tightly coupled, and there are many dependency cycles between their corresponding packages.

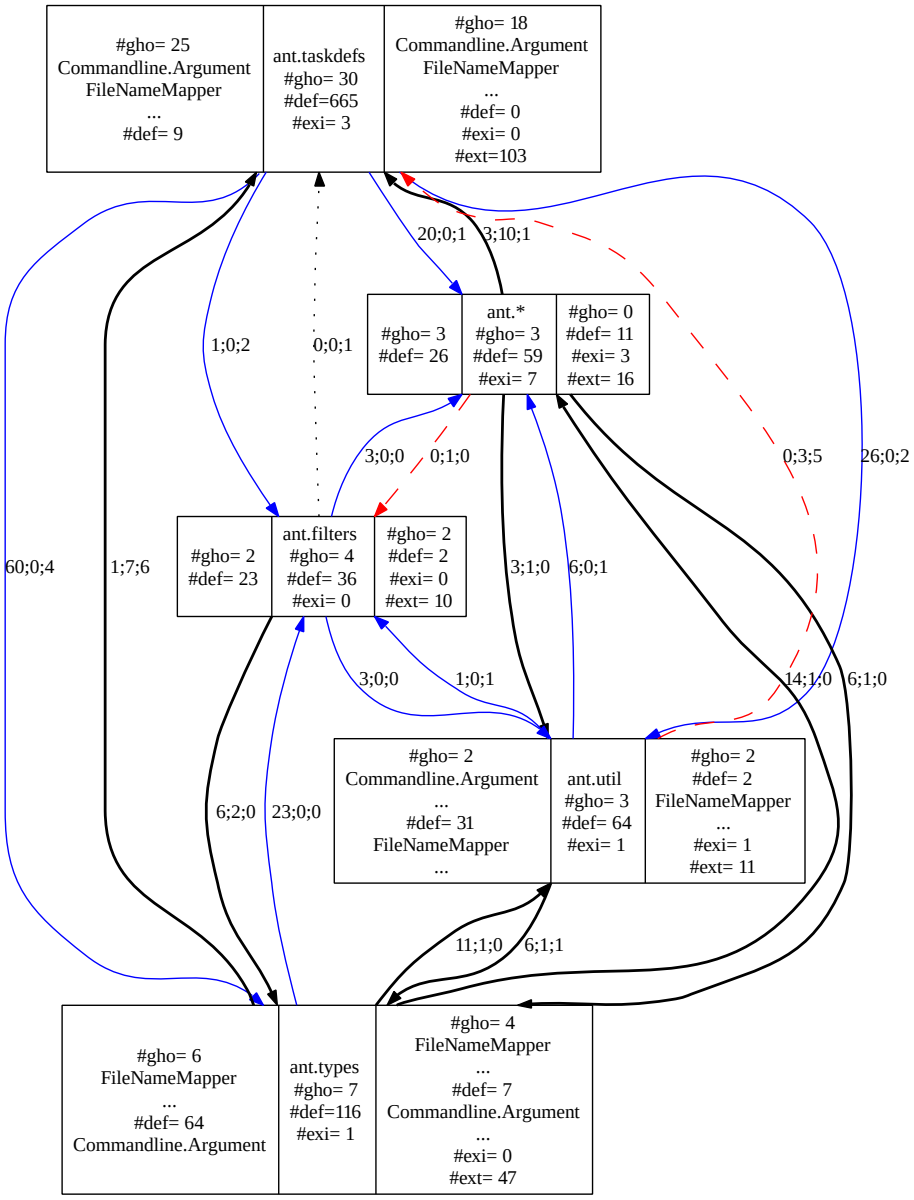


Figure 5.4: Assembly Connectors among 5 third-level Components.
Bold solid lines represent PSRD connectors; Blue solid lines represent RD connectors;
Red dashed lines represent PS connectors. Black dotted lines represent RB connectors.
Each connector is labelled with strength(#DIS; #DIC; #DIT).

The remaining five third-level components, which are not shown in Figure 5.4, are either small or have thin inports. In a Hybrid Model, the inport of a component indicates the responsibilities that it reveals to the outside through usage dependencies. Therefore, we are able to peek into the responsibilities of the five components by reviewing only a few classes associated with their inports.

5.6.2 Refactoring Opportunities

Dependency analysis over Hybrid Models not only helps to identify architecturally significant information, such as the external properties of groups of objects and the collaboration between them, but also can help to identify the design intention about program logic encapsulation and data abstraction. Some design decisions may become inappropriate as software evolves. In this case study, we found at least three types of potential design problems.

Intimate Collaboration

Components collaborate with each other by providing the resources required by others. A resource may have multiple servers and multiple clients. However, it is undesirable that a component provides a resource to its clients, and requires the same resource from its servers. It indicates that the resource has multiple implementations in different components, and those implementations requires intimate collaboration from one another. As the result, their containing components are tightly coupled, and the modularity of the system is poor.

For example, Figure 5.4 shows that ant.taskdefs and ant.types are open for variation of the same resource, Argument. ant.util also provides an implementation of Argument. After examining the internal structure of ant.util, we found that the subcomponent ant.util.facade contains a subclass of Argument, which is further inherited by some classes defined in ant.taskdefs. This subcomponent is not used by any other subcomponents within ant.util. It is likely that programmers intend to reuse the code from Argument without changing the existing code. At the same time, it is also likely that the developers wanted to maximize the code reuse by putting common code in the subclass of Argument in ant.util. As a result, the possible behaviors of Argument are described in three different packages, maintainers

may have to examine several levels of class hierarchy to get a complete description of an object of Argument. "Replace Inheritance with Delegation" [25] refactoring technique can be applied to reuse code of Argument via composition instead of via inheritance [26].

The three components also provide and require resource FileNameMapper. After examining the internal structure of ant.util, we found that there is an implementation of Composite design pattern [26]. FileNameMapper is the base class, and ant.util contains the composite classes and most leaf classes. ant.types and ant.taskdefs both have a couple of leaf classes. Maintainers need to determine whether it is necessary to define similar objects in different packages. "Replace Inheritance with Delegation" [25] and "Move Class" [23] refactoring techniques can be applied to regroup classes.

Conflicting design intentions

An indirectly-used inport reflects the designers' intention to allow a component to hide implementations of abstract concepts, so that the component can be extended without changing other components. If a maintainer allows the component to be used by other components directly, the original design intention is violated. Such modification may lead to architectural decay. Based on this assumption, we searched for any components whose inport exports more resources provided by its ghost classes than the resources provided by its defined classes. In this case study, we found a possible conflicted design intention in ant.taskdefs.

Component ant.taskdefs in Figure 5.4 is likely designed to be used indirectly. It is a giant component with relatively thin port. Most of its exported resources are ghost resources. Figure 5.5 shows that half of its incoming dependencies are either RB or PS connectors; two PSRD connectors from ant.types and ant.* are associated with more DIC resources than DIS resources. Therefore, ant.taskdefs are often known as a provider of abstract concepts. In fact, it was always used indirectly until version 1.5.

Refactoring techniques [25] can be applied to turn ant.taskdef back into an indirectly-used component. The 9 defined resource that ant.taskdefs provides are used by four different components, likely in different usage scenarios. ant.listener.* and ant.loader.* use resources, which are originally only used within ant.taskdefs. Those resources can be split from the ant.taskdefs and moved to a utility packages, e.g. ant.util. Newly added class Redirec-

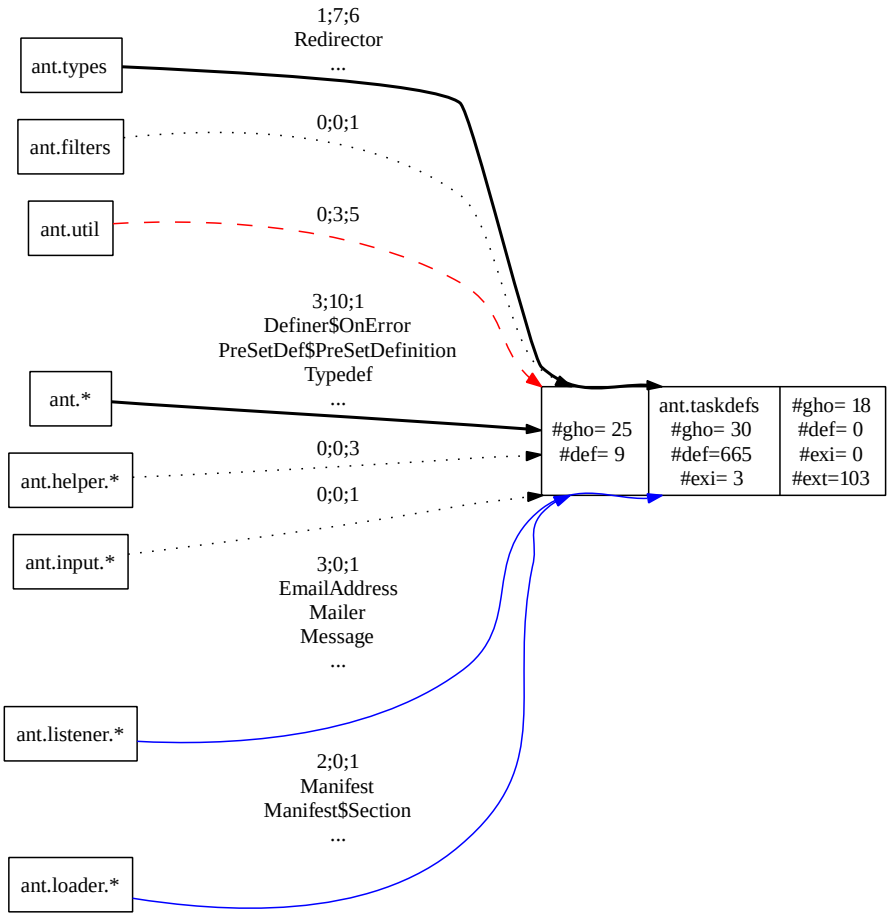


Figure 5.5: Incoming Assembly Connectors of ant.taskdefs.

torElement in ant.types uses Redirector in ant.taskdefs, which leads to a dependency cycle between two packages. To break the package dependency cycles, maintainers may apply "Extract Interface" refactoring technique [25] on the exported resources, separate their implementation from their interface, and let both packages depends on the interface. The same approach can be applied to change the PSRD connector from ant.* and ant.taskdefs into a PS connector.

Fat Connectors

An assembly connector specifies a client-server relationship between two components. A strong assembly connector indicates tight coupling between the components. Reducing the strength of an assembly connector may reduce the coupling between the original packages.

ant.filters as shown in Figure 5.4, collaborates with four other components. Compared to other incoming connectors of ant.filters, the connector from ant.types is much stronger than the others. It also contribute a large portion of the required resources of ant.types.

We examine the internal structure of component ant.types, and found that only one class, FilterChain, uses the 21 resources provided by ant.filters. FilterChain is packaged in ant.types mainly because it inherits DataType as most of the other classes in ant.types. Other than inheritance, FilterChain has few dependencies with other classes in ant.types. Therefore, it is possible to reduce the strength of the connector between the two components by simply moving FilterChain into ant.filters. Maintainers may also considering using "Replace Inheritance with Delegation" and "Extract Class" [25] refactoring techniques to further decouple the two components and allow the implementations of the same abstract concept remain in the same package.

5.6.3 Summary of Case Study

Hybrid Models provide a promising approach to visualize and analyze object-oriented systems at coarse-grained levels of abstraction. A Hybrid Model captures all possible usage dependencies between components. As cross-package inheritance relationships are removed, we not longer need to consider how inheritance affects the interpretation of usage dependencies. More importantly, when we focus on a particular region of a system, there is no need to think about the implicit dependencies that exist through classes outside the region.

Dependency analysis over Hybrid Models reveals the external properties of a component, such as the number and types of resources that the component provides to and requires from others. We can derive the role that a component plays in the system based on its external properties. Especially, when a component has a thin inport, we are able to learn its responsibilities, and better understand the component as a whole.

Hybrid Models can also help maintainers to recover original design intentions of the sys-

tem, including data abstraction, program logic encapsulation. Furthermore, Hybrid Models can help to identify potential design problems — such as tight coupling and compilation dependency cycles — and suggest possible solutions to the problems.

5.7 Summary

In this chapter, we apply Hybrid Models to analyze dependencies at coarse-grained level of abstraction. Our approach focus on two perspectives: the external properties of component, and the characteristics of connectors. The exploratory case study of Apache Ant system shows that the dependency analysis results can help maintainers capture the external properties of coarse-grained entities and better understand the nature of their interdependencies.

Chapter 6

Architectural Change Analysis

Summary As an object-oriented system evolves, its architecture tends to drift away from the original design. Knowledge of how the system has changed at coarse-grained levels is key to understanding the de facto architecture, as it helps to identify potential architectural decay and can provide guidance for further maintenance activities. However, current studies of object-oriented software changes are mostly targeted at the class or method level.

In this chapter, we propose a new approach to modelling the evolution of object-oriented software changes at coarse-grained levels. We take snapshots of an object-oriented system, represent each version of the system as a Hybrid Model, and detect software changes at coarse-grained level by comparing two Hybrid Models. Based on this approach, we further identify a collection of change patterns, which help interpret how a system changes at the architectural level. Finally, we present an exploratory case study to show how our approach can help maintainers capture and better comprehend architectural evolution of object-oriented software systems.

Contributions. The contributions of this chapters are the following:

- The evolutionary analysis approach based on Hybrid Models.
- The list of architectural change patterns involving individual components and multiple components.

Structure of the chapter. The remainder of this chapter is organized as follows. Section 6.1 shows the importance of evolutionary analysis. Section 6.2, Section 6.3, and

Section 6.4 elaborate our approach of evolution analysis. Section 6.5 presents the visualization support for the evolutionary analysis. In Section 6.6, we apply the evolution analysis on an exploratory case study. Section 6.7 discusses current researches related to our work, and Section 6.8 summarizes what we have done.

6.1 Introduction

Change is a measure of success in the world of software. As users grow familiar with a system, they often conceive of new features that can be added and new kinds of problems that can be attacked. Successful systems will respond positively to these pressures to change, with the addition of new features and support for using the system in new environments to solve new problems. However, change processes themselves tend to be incremental rather than revolutionary; over time, as changes accrue, the de facto architecture tends to drift away from the original design goals and architectural plans. In the absence of careful re-architecting, the design of the evolving system becomes brittle and resistent to further change [45, 52, 76]. Maintenance activities become more difficult, time consuming, and risky.

Explicitly modelling the changes that have occurred to a software system — at different granularities and from different points of view — provides valuable information to the system maintainer who needs to understand exactly why a system's design is the way it is and what strategies may work best to effect future change. In the last decade, more and more attention has been focused on uncovering evolution change from source code or historical data [16, 18, 29, 108]. The goal of our work is to study evolutionary information of object-oriented software systems as they also suffer from high maintenance costs, and may benefit from this kind of historical modelling and analysis.

Most current research on object-oriented software evolution is targeted at the class level of abstraction, which is natural as classes are the basic building blocks of object-oriented programs. However, such an approach does not scale well to the system level due to the large volume of information involved. A complex object-oriented system typically consists of hundreds of classes, which in turn may exhibit a high degree of interdependence among them. Furthermore, while considering one large system is hard enough, comparing multiple

versions of a system exacerbates the scaling problems by an order of magnitude.

One way of managing complexity is to model and analyze evolution at a coarse-grained level, such as the package level. However, in languages such as Java and C++, a package or namespace construct is simply a container of classes and has little or no semantics; a package does not exhibit the important semantic properties of its containing classes as types of objects, and a package diagram is unable to capture inheritance and usage dependency between classes at a coarse-grained level.

Hybrid Models capture the important properties of classes, including inheritance and usage dependencies, at a coarse-grained level of abstraction. They not only present overviews of object-oriented systems, but can also provide a basis for evolution analysis at a selected level of granularity. In this chapter, we apply Hybrid Models to uncover and analyze evolutionary change at the system level. The evolution analysis is achieved by comparing the Hybrid Models of adjacent versions. With Hybrid Models, we were able not only to gain an overall picture of software evolution, but also to investigate the detailed structural change in a selected scope at a preferred level of granularity. We have applied our approach in an exploratory case study, and have identified a collection of change patterns that help interpret how a system changes at the architecture level.

In the next three sections, we identify and analyze the changes in individual components, as well as the changes that involves multiple components.

6.2 Change in Aggregate Components

Our objective of investigating change in an individual component is to capture the externally visible change of its corresponding package. In a Hybrid Model, the properties of a package are summarized into the defined, ghost and exiled classes, and the inport and outport of its corresponding component. Therefore, to gain an overview of package-level evolution, we consider the additions and removals in those five parts of the component:

- $\Delta_{Defined}$: The number of defined classes that have been added or removed serve as a measure of the growth of the containing package.
- Δ_{Ghost}: If the set of ghost classes of a component has changed, then cross-package

inheritance relations have also changed.

- Δ_{Exiled}: Exiled classes are effectively the set of abstract concepts declared in a component, but implemented elsewhere; consequently, a change in this set means that the high-level design of the system has changed.

- Δ_{inport}: A component provides services to others through its inport; As a result, a change in the inport likely entails a change in the component's responsibilities, possibly through a refactoring of the high-level design. In Dig and Johnson's study on frameworks, most incidents of "breaking" an API were found to result from refactoring activities [18].

- $\Delta_{outport}$: An aggregate component requires services from others through its outport. Adding a resource to or removing a resource from the outport indicates that the component requires different services, i.e., that the implementation details have changed.

A usage dependency, such as **calls**, between two classes indicates the possible relations between the objects they represent. Since an object of a class is also polymorphically an object of the ancestors of the class, usage dependencies must be interpreted in the context of a class hierarchy. In Hybrid Models, such interpretation is reflected in terms of the composition of ports. Thus, we also study how the port composition changes.

- Change of inport composition.

  An aggregate component may receive **direct** and **indirect** requests. A **direct** request is sent to an object of a defined class, while an **indirect** request is sent to an object of an ghost class. Increasing the services provided by its defined classes indicates more defined classes are directly known to other packages. Increasing the services provided by the ghost classes shows that more defined classes of the component is hidden, and known to others as the implementation of some abstract concepts.

- Change of outport composition.

  A component may send requests to an object of its exiled class. This reveals that the component is "open for extension" [60], and the requests are the contract that

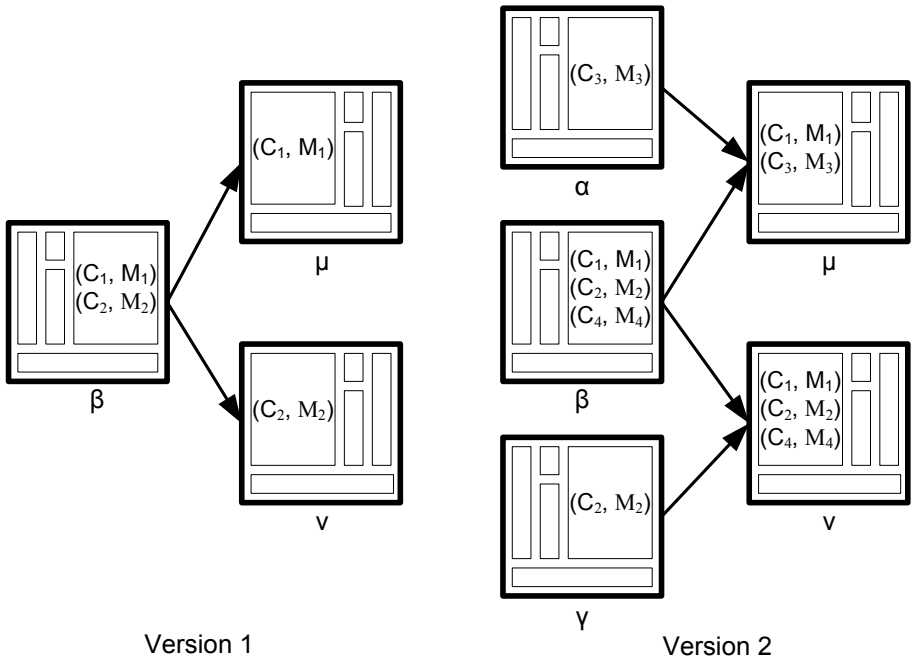


Figure 6.1: Change in Assembly Connectors.

specify the obligations of its server components. Change, especially the removal, of an **open-for-extension** request implies that the interactions between the component and its service providers are changed.

A component may also send requests to an object of a ghost or defined class of the component. The required service has a number of possible behaviors. Some are defined in the component, while others are implemented externally. The addition of **open-for-variation** requests indicates the increasing coupling between variations of implementations for the same abstract concept.

6.3 Change in Assembly Connectors

The previous section describes the approach to identify the externally visible change of individual components. To understand how change in one component affects or is affected by another, we must investigate the change of the assembly connectors between them.

An assembly connector specifies a client-server relationship between the components at the same level of granularity. The assembly connector between the client component μ and the server component ν is associated with the resources shared by the outport of μ, and the inport of ν.

$$ac(\mu, \nu) = outport(\mu) \cap inport(\nu) \tag{6.1}$$

An assembly connector is changed if there is any addition or removal of the associated resources. Change in the assembly connector between client α and server μ can be characterized by the following equation:

$$\begin{aligned} \Delta_{ac}(\alpha, \mu) &= \Delta_{outport}(\alpha) \cap \Delta_{inport}(\mu) && (6.2) \\ &\cup \Delta_{outport}(\alpha) \cap \Xi_{inport}(\mu) && (6.3) \\ &\cup \Xi_{outport}(\alpha) \cap \Delta_{inport}(\mu) && (6.4) \end{aligned}$$

6.3.1 Co-change between the client and server component

As Equation 6.2 depicts, an assembly connector changes if a resource is added to or removed from both the outport of the client component and the inport of the server component. This indicates either the change in the server component leads to the change in the client component, or the server component changes in order to meet new requirements of the client component. For example, in Figure 6.1, component α requires a new resource, (C_3, M_3), and component μ provides such a new resource. Thus, $\Delta_{ac}(\alpha, \mu) = \{(C_3, M_3)\}$.

To further investigate how the client and server component affect each other, we normalize the number of co-changed resources with the change size of the client outport and the server inport, respectively.

$$r_{afferent}(\alpha, \mu) = \frac{\mid \Delta_{outport}(\alpha) \cap \Delta_{inport}(\mu) \mid}{\mid \Delta_{inport}(\mu) \mid} \tag{6.5}$$

$r_{afferent}(\alpha, \mu)$ represents how much change in the inport of the server component μ is demanded by the client component α. Comparing $r_{afferent}$ of all incoming assembly connectors of a server component, we are able to assess why its responsibility has changed.

In Figure 6.1, $r_{afferent}(\alpha, \mu) = 1$ and $r_{afferent}(\beta, \mu) = 0$, thus new responsibility is added to component μ solely for the needs of component α.

$$r_{efferent}(\alpha, \mu) = \frac{\mid \Delta_{outport}(\alpha) \cap \Delta_{inport}(\mu) \mid}{\mid \Delta_{outport}(\alpha) \mid} \tag{6.6}$$

$r_{efferent}(\alpha, \mu)$ indicates how much change in the outport of the client component α is caused by the change from the server component μ. Comparing $r_{efferent}$ of all outgoing assembly connectors of a client component, we can evaluate how its change depends on the change in other components. In Figure 6.1, $r_{efferent}(\beta, \mu) = 0$ and $r_{efferent}(\beta, \nu) = 1$, thus only component ν contributes to the change of component β.

6.3.2 Reuse Resources

Equation 6.3 indicates that an assembly connector changes if a resource is added to or removed from the outport of the client component, but remains unchanged in the inport of the server component.

The client component requires a resource that the server component provided in the previous version, or the client component no longer requires the resources, while the server component still provides such a resource to other components. For example, $\Delta_{ac}(\gamma, \nu) = \{(C_2, M_2)\}$, since component γ in version 2 reuses an existing resource, (C_2, M_2), provided by component ν.

6.3.3 Re-implement Resources

Equation 6.4 indicates that an assembly connector changes if a resource is added to or removed from the inport of the server component, but remains unchanged in the outport of the client component.

The server component provides an implementation for the resource that was required in the previous version, or the server component no longer provide the resource, but other components still provide the same resources. For example, component ν provides the resource, (C_1, M_1) to component β, which already used the resource in version 1.

6.3.4 Summary

When an assembly connector changes due to the co-change between components, it is likely that the involved resources become or are no longer significant at the package level. When an assembly connector changes for the purpose of reuse or re-implementation, the involved resource are significant in both versions of the software system.

6.4 Change in Delegation Connectors

A delegation connector promotes the ports of the sub-components to their container component. Thus, the delegation connector between component α and its containing component β is associated with the resources shared by their corresponding ports.

$$\begin{aligned} dc_{inport}(\alpha, \beta) &= inport(\alpha) \cap inport(\beta) \\ dc_{outport}(\alpha, \beta) &= outport(\alpha) \cap outport(\beta) \end{aligned}$$

Additions and removals of resources from delegate connectors reveal how fine-grained change contributes to the change at a coarse level of granularity. The analysis on the delegation connectors promoting inports is same as the analysis on the delegation connectors promoting outports. Therefore, we discuss only the analysis on the delegation connectors that connect the inports of components.

Suppose component α contains component μ, the change of the delegation connector between the two can be divided into three parts.

$$\begin{aligned} \Delta_{dc-inpot}(\alpha, \mu) &= \Delta_{inport}(\alpha) \cap \Delta_{inport}(\mu) && (6.7) \\ &\cup \Delta_{inport}(\alpha) \cap \Xi_{inport}(\mu) && (6.8) \\ &\cup \Xi_{inport}(\alpha) \cap \Delta_{inport}(\mu) && (6.9) \end{aligned}$$

6.4.1 Internal change leads to external change

As Equation 6.7 shows, a delegation connector changes if the involved resources are added to or removed from both the ports of the container component and the ports of the containee

component. For example, in Figure 6.2, component μ provides a new resource (C_3, M_3), which is promoted by its container component α.

To further investigate how the fine-grained components contribute to the externally visible change of their container component, we normalize the number of exposed internal changes with the change size of the container inport and the containee inport, respectively.

$$r_{coarse-in}(\alpha, \mu) = \frac{\mid \Delta_{inport}(\alpha) \cap \Delta_{inport}(\mu) \mid}{\mid \Delta_{inport}(\alpha) \mid} \tag{6.10}$$

$r_{coarse-in}(\alpha, \mu)$ represents how much of the externally visible change of component α at the coarse-grained level is contributed by the component μ at the more fine-grained level. Comparing $r_{coarse-in}$ of the delegation connectors from all containee components, we are able to answer questions, such as: Which component contributes the most to the externally visible change of its container component? Do the sub-components equally contribute to the change of their container, or is there a dominant contributor?

$$r_{fine-in}(\alpha, \mu) = \frac{\mid \Delta_{inport}(\alpha) \cap \Delta_{inport}(\mu) \mid}{\mid \Delta_{inport}(\mu) \mid} \tag{6.11}$$

$r_{fine-in}(\alpha, \mu)$ represents how much the change in component μ at the fine-grained level contributes to the externally visible change of component α. Examining $r_{fine-in}$ for all containee components, we are able to learn whether the majority of change at the fine-grained level is externally visible at the coarse-grained level.

6.4.2 Exposing or hiding internal resources

Equation 6.8 indicates that a delegation connector changes if the container component exports or hides the resource that is significant at the finer-grained level. For example, in Figure 6.2, resource (C_2, M_2), which was significant at the fine-grained level in version 1, becomes visible at the coarse-grained level in version 2.

6.4.3 Reusing or re-implementing external resources

Equation 6.9 indicates that a delegation connector changes if the port of the containee component changes but the port of container component remains same. In this case,

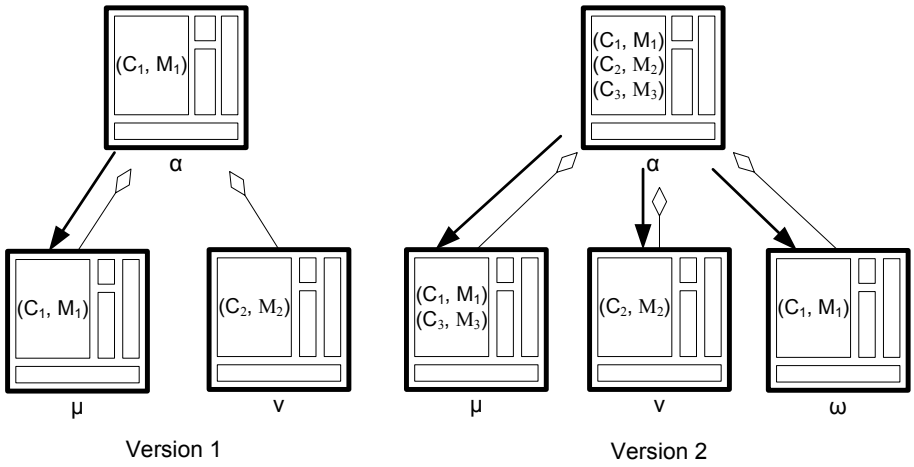


Figure 6.2: Change in Delegation Connectors.

resources are added to or removed from the ports of the containee component, while those resources are significant at the coarse-grained level in both versions. This happens when at least one of the siblings of the containee components provides or requires the same resources. For example, in Figure 6.2, new component ω in version 2 provides resource (C_1, M_1), which was already exported by component α in version 1.

6.5 Visualization Support

In previous sections, we list some of the possible change patterns involving individual components, assembly and delegation connectors. The analysis on an aggregate component captures the externally visible change of the selected scope; the analysis on assembly connectors show how the components at the same level of granularity affect each other; the analysis on delegation connectors describes how the internal change of a scope affects its external properties at a coarser-grained level.

A system may experience various combinations of the change mentioned above. As modification to a system accumulates to a certain degree, the implementation may drift away from the original design. For example, the component, which was intended to respond only to indirect requests, may over time receive more and more direct requests. Or the

inport of a component may grow much quickly than the component so that more and more its defined classes become externally visible. Or two unrelated components start to collaborate with each other.

We could have defined a set of heuristics and thresholds to detect and report significant changes at the architecture level. However, we believe that it is subjective to determine whether a change leads to architecture drift or decay, and many other factors should also be taken into consideration, such as the current architecture, the history of the target system, and the prediction of future change.

Therefore, we choose to use visualization techniques to help maintainers more quickly apprehend the change at coarse-grained levels, and let them decide whether the design intention has changed and what maintenance actions might be appropriate.

Our toolkit can produce three automated evolutionary views of a target system:

1. A *snapshot view* presents the Hybrid Model that is reverse engineered from an object-oriented system at a particular point in its history. The aggregate components in the Hybrid Model are organized in a tree structure, which is consistent with the package containment hierarchy. For example, Figure 5.3 is the snapshot of Ant version 1.6.5.

2. A *comparison view* presents the difference between two selected Hybrid Models. For example, Figure 6.5 shows the partial difference between Ant version 1.4 and 1.5.

3. An *evolution matrix* organizes snapshots of the selected aggregate components in a matrix. It provides an overview of how a collection of aggregate components change over time. For example, Figure 6.4 shows the history of the packages contained in package *ant*.

The properties of a Hybrid Model that are key to our evolution analysis are also visualized in the three views. As shown in Figure 6.3, the height of a component is determined by the change size of its ghost classes, defined classes, and exiled classes, while the width of a component is determined by the change size of its ports. The color of each part of an aggregate component indicates whether the change is mainly addition or removal.

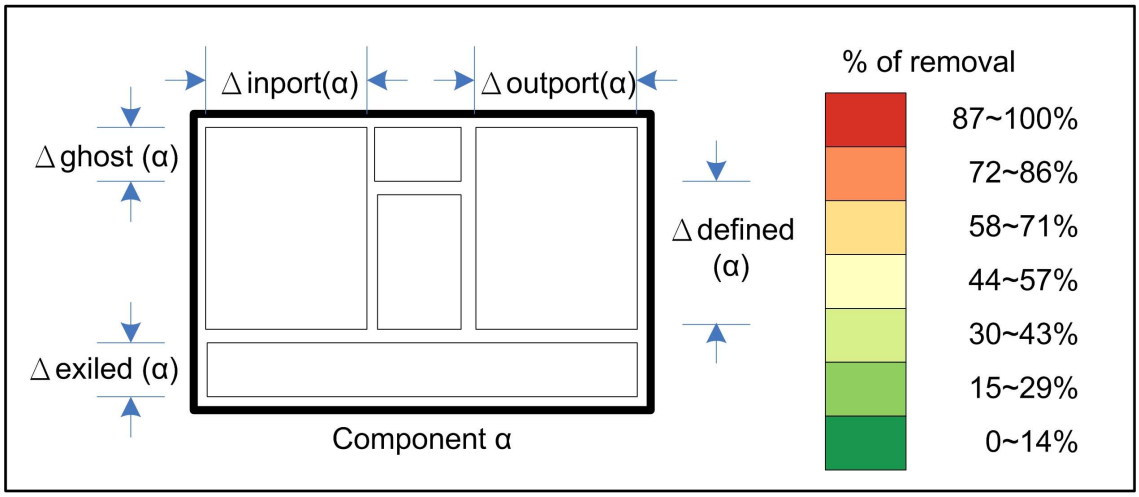


Figure 6.3: Visualization Support for Evolutionary Analysis

6.6 Case Study: The Evolution of Apache Ant

To evaluate the change analysis approach explained in the previous sections, we conducted an exploratory case study on a Java-based build tool, called Apache Ant [22]. It was selected for the case study because it is a medium-sized object-oriented system, which has more than 7 years of history and is still under development, and also because we are familiar with it through our study on its architecture (Chapter 5).

Apache Ant, originally a part of the Apache Tomcat [95] project, became an indepen-

Version	Release Date	# package	# classes
1.1	19 Jul 2000	6	116
1.2	24 Oct 2000	17	224
1.3	3 Mar 2001	25	323
1.4	3 Sep 2001	35	507
1.5	10 Jul 2002	56	731
1.6	18 Dec 2003	67	943
1.7	19 Dec 2006	71	1158

Table 6.1: The size and release date of the studied versions.

dent project in July 2000. It has been used as a general-purpose build tool in many projects since then. In the first release of Ant as a stand-alone project, there were 6 packages and 116 classes, while in the latest release, there were 71 packages and more than a thousand classes. In this case study, we analyzed 7 major releases, whose release dates, and the number of packages and classes are listed in Table 6.1.

The focus of this case study was on studying how Apache Ant evolved over time. We apply Hybrid Models to visualize the changes, and to seek answers to the following questions:

- How did the externally visible properties of a package evolve?
- How did the internal structure of a complex package evolve?
- What was the relationship between the external and internal change of a package?
- What was the evolutionary influence among sibling packages?

6.6.1 How has the package *tools.ant* evolved?

We were particularly interested in the evolution of package *tools.ant* as it contains over 95% classes of the whole system, and has a complex structure. We extracted Hybrid Models for all studied releases, and created an evolution matrix. As Figure 6.4 shows, package *tools.ant* has changed a lot since version 1.1. New packages were introduced in each major release. All of the original packages still exist in version 1.7, but in different sizes, shapes, and colors. Compared to their first appearance in the system, small packages, especially those introduced in the later versions of the system, are relative stable, while the four packages with the longest history have changed most.

In the remainder of this section, we discuss our detailed observations about the history of Apache Ant. We pay particular attention to possible indicators of architectural drift.

Increasing number of ghost classes

The number of ghost classes at this level of granularity has steadily increased, as shown in Figure 6.4. There are only two ghost classes in version 1.1, and both belonged to package

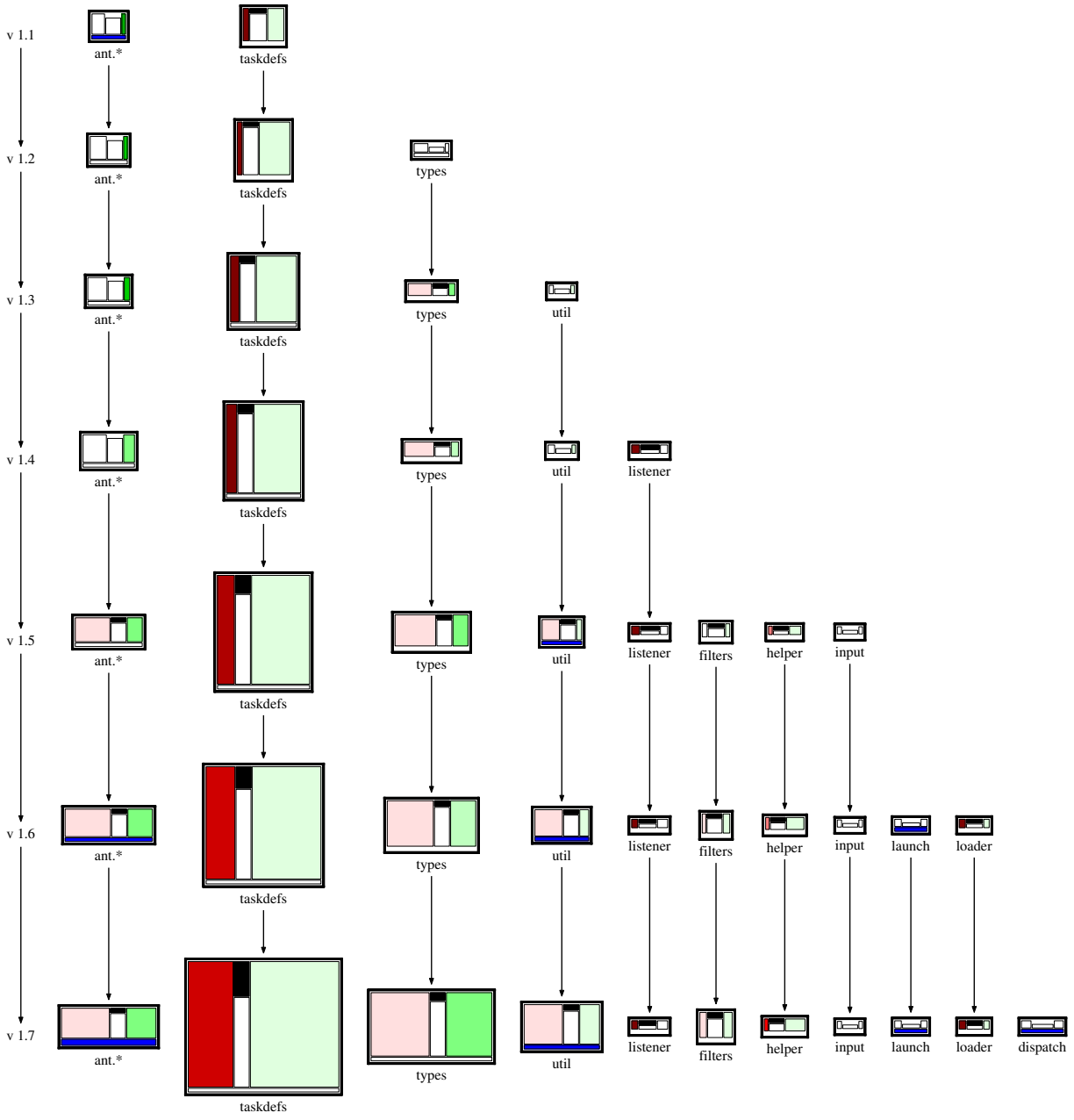


Figure 6.4: The history of pkg. *tools.ant* is displayed in an Evolution Matrix.

taskdefs. In version 1.7, 63 classes have become ghost classes of one or more of the 7 packages at this level.

Several packages acquired ghost classes when they were first introduced to the system. For example, package *filters* implemented an abstract class that was new to the system, while the other three packages implemented the classes that already existed in the previous versions: package *listener* inherited class `BuildListener`, package *helper* inherited class `ProjectHelper`, and package *loader* inherited class `AntClassLoader`. The class and package names suggest that each one of the three package was used to group some classes that implement a common abstract concept. The dark colored inports of the three components, as shown in Figure 6.4, further confirmed that they were mostly used indirectly through their ghost classes, and known to their clients as the implementation for those ghost classes, while their internal classes were hidden from others. It is not surprising that these packages, which were initially designed to be at a low data abstraction level, continuously have ghost classes over their lifetime. However, when a package with no ghost classes starts to inherit from other packages, it may be an indicator of architectural drift. In the versions of Ant that we studied, we found three such cases, two of which are related.

Ghost classes first appeared in package *types* in version 1.3, when the package reused an existing class, `DirectoryScanner`, from package *ant.**. In version 1.5, package *ant.** acquired a ghost class, `SelectorScanner`, which was declared in package *type*, and is the superclass of class `DirectoryScanner`. We consider it likely that developers intended to introduce a more general concept without causing much change to the existing high-level design. As a result, a class in the middle of a class hierarchy tree was separated from the other family members. Furthermore, in version 1.6, package *ant.** was involved in another cross-package inheritance relationship, which was also due to the presence of class `DirectoryScanner`. History shows that the number of ghost classes in package *types* continually increased since version 1.3, while package *ant.** had only a few in the more recent version. Thus, we consider that it would be reasonable for maintainers to apply "Move Class" refactoring technique [25], and move class `DirectoryScanner` from package *ant.** to package *type*.

The third case took place in version 1.5, when package *util* acquired a ghost class named `Argument`, which is an inner class from package *types*. The same class also appeared as

a new ghost class in package *taskdefs*. After examining the internal structure of package *util*, we found that one of its subcomponents contains a subclass of `Argument`, which is in turn inherited by some classes defined in package *taskdefs*. This subcomponent is not used by any other subcomponents within package *util*. It is likely that programmers intend to reuse the code from class `Argument` without changing the existing code. At the same time, it is also likely that the developers wanted to maximize the code reuse by putting common code in package *util*. As a result, the possible behaviors of `Argument` are described in three different packages.

The three cases described above not only indicate the design change in three individual packages, but also reveal the trends that there were increasing number of the classes that were implemented in multiple components. Using package *util* as an example, by the time of version 1.7, it had 7 ghost classes, 5 of which were implemented in more than two components.

Removal of Exiled Classes

Since version 1.5, a number of exiled classes have been introduced to serve as contracts between components. For example, class `TimeoutObserver` was introduced to package *util* in version 1.5 as an update interface to receive the signal from the class `WatchDog`. The two classes are a part of an instance of the observer design pattern [26].

However, it is unusual to remove an abstract concept that was significant at package levels; we noted only one such case in the studied period, and it resulted from package splitting. In version 1.1, package *ant.** had an exiled class, `EnumeratedAttribute`, which was extended by package *taskdefs*. The exiled class disappeared in the next version. At the same time, the new package *types* had a class with the same class name. It is possible that class `EnumeratedAttribute` was moved. After examining the other removed classes in package *ant.**, we found that class `Path` were also moved from package *ant.** to *types*. This confirmed that some classes were split from package *ant.** to form a new package.

Expansion of Ports

One noticeable architectural change in Figure 6.4 is that the ports of most packages became wider and wider, and the color of some ports changed too.

The width of package *ant.** grew much faster than its height. Compared to package *ant.** in version 1.1, the package in version 1.7 responded to 4 times more message types, while the number of its defined classes increased by only 50%. This indicates that one or more classes of the package have accumulated a number of responsibilities that are significant at the package level. Thus, applying change to those classes become more likely to affect other packages.

The growth of package *taskdefs* can be divided into two stages. Before version 1.5, it grew much faster than any other packages, while its inport had little change. This is not surprising since package *taskdefs* was originally designed to extend the abstract concepts from package *ant.**. However, since version 1.5, not only did its inport expand rapidly, the color of its inport also became lighter. This indicates that it started to receive more and more direct requests from other packages. It is likely that package *taskdefs* increasingly exported its own responsibilities besides the abstract concepts it implements.

When the packages *types* and *util* were first introduced to the system, both were small. Later both grew into complex packages with wider ports. The color of their ports shows that they received both indirect and direct requests, and were continuously open for variation. This indicates as the system has grown, there have been an increasing number of concepts implemented in multiple packages, and the coupling between the implementations increased as well.

Change of Assembly Connectors

We also created comparison views to study the difference between the Hybrid Models of adjacent versions of Ant. There are at least 3 similarities shared by those views.

First, we observe that package *taskdefs* appears to be the driving force of the evolution of Ant: it is the biggest subpackage of the top-level package *tools.ant*, it grew much faster than any of its siblings, and it continually demanded changed resources from its siblings. In Figure 6.5, package *taskdefs* has dark outgoing arrows to five siblings , which indicates that package *taskdefs* depended on the change in its siblings. As the arrows are weighted by $r_{afferent}$ of assembly connectors, the dark color indicates most of the inport change of the five packages were contributed to the changes in package *taskdefs*. It is likely that package *taskdefs* demanded new services as it grew, and its sibling packages changed in

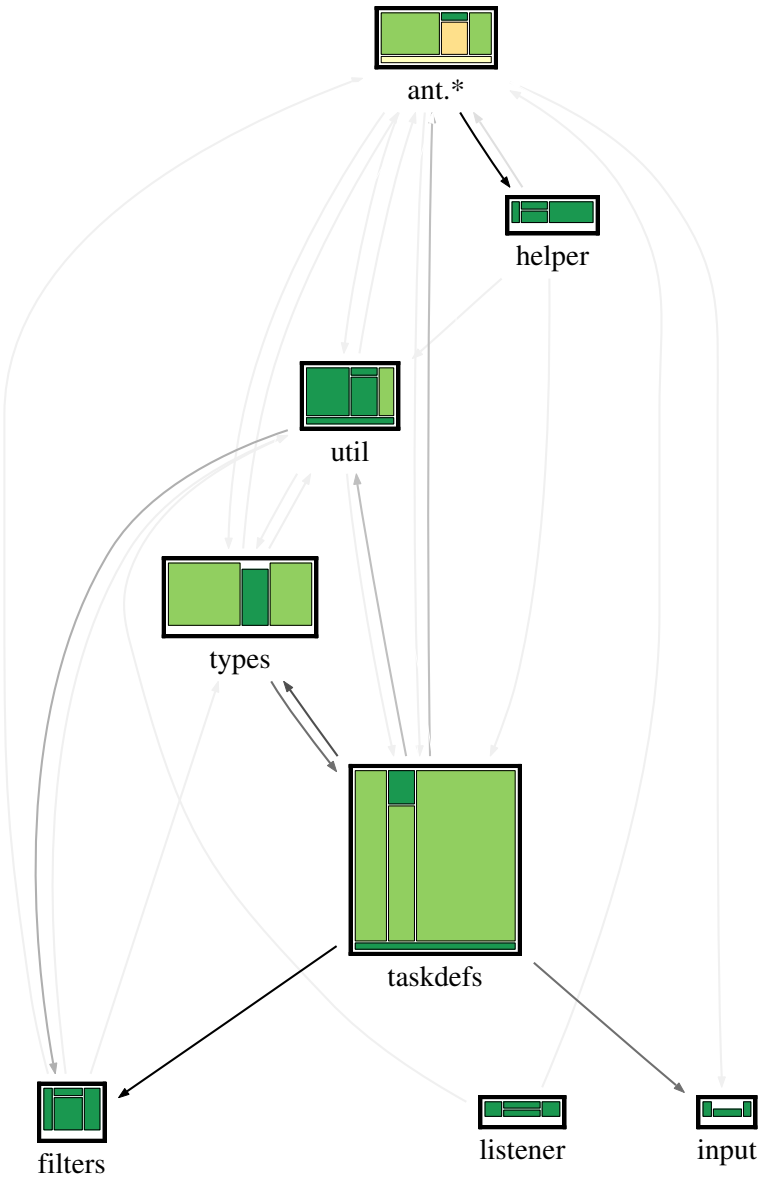


Figure 6.5: Assembly connector change in pkg. *tools.ant* caused by co-change between components. The color shades of connectors are determined by their $r_{afferent}$ values.

order to meet its new requirements.

Second, package *ant.** provides some services that are used and reused by its siblings. Figure 6.6 shows that some resources package *ant.** provided in version 1.4 were reused by its sibling in version 1.5. Change to those services may affect a number of other packages.

Third, there are increasing number of classes that were implemented in multiple packages. Figure 6.7 shows that in version 1.5, both package *util* and *listener* provided new implementation for the classes that were important in version 1.4 at the package level.

6.6.2 Evolution at the Finer-grained Level

As Ant system has evolved, its packages *util*, *taskdefs*, and *types* have all become significantly more complex and have acquired several subpackages. However, their evolutionary histories are quite different.

Most internal changes in package *util* are externally visible. Figure 6.8 depicts the comparsion view of package *util*. The dark color of the delegation connectors indicates that the change at the sub-package level is also visible at the package level. Package *util*, as its package name suggested, was initially designed to be a utility package, providing services shared by other packages. It is composed of several sub-packages with few dependencies among them. Those sub-packages evolves independently.

Package *taskdefs* continuously provided implementations for the existing abstract concepts. As Figure 6.9 depicts, all sub-packages of *taskdefs* in version 1.5 implemented some existing classes that were significant at the coarse-grained level. Package *taskdefs* was designed as an implementation package, which contains many ghost classes, such as class `Task`, `EnumeratedAttribute`, etc. Many of them were implemented in more than one sub-package of *taskdefs*. Therefore, when a new class or a new package is added, the resources it provides or requires may have already been important at the coarse-grained level.

Many internal changes in package *type* are limited within the package. As Figure 6.10 demonstrates, all three sub-packages of *type* have added ghost classes, and two of them have added exiled classes, but none of them are visible at the coarse-grained level. In addition, the delegator connector between package *types* and *types.** are darker than others, which indicates that most externally visible changes of package *types* are contributed by package *types.**.

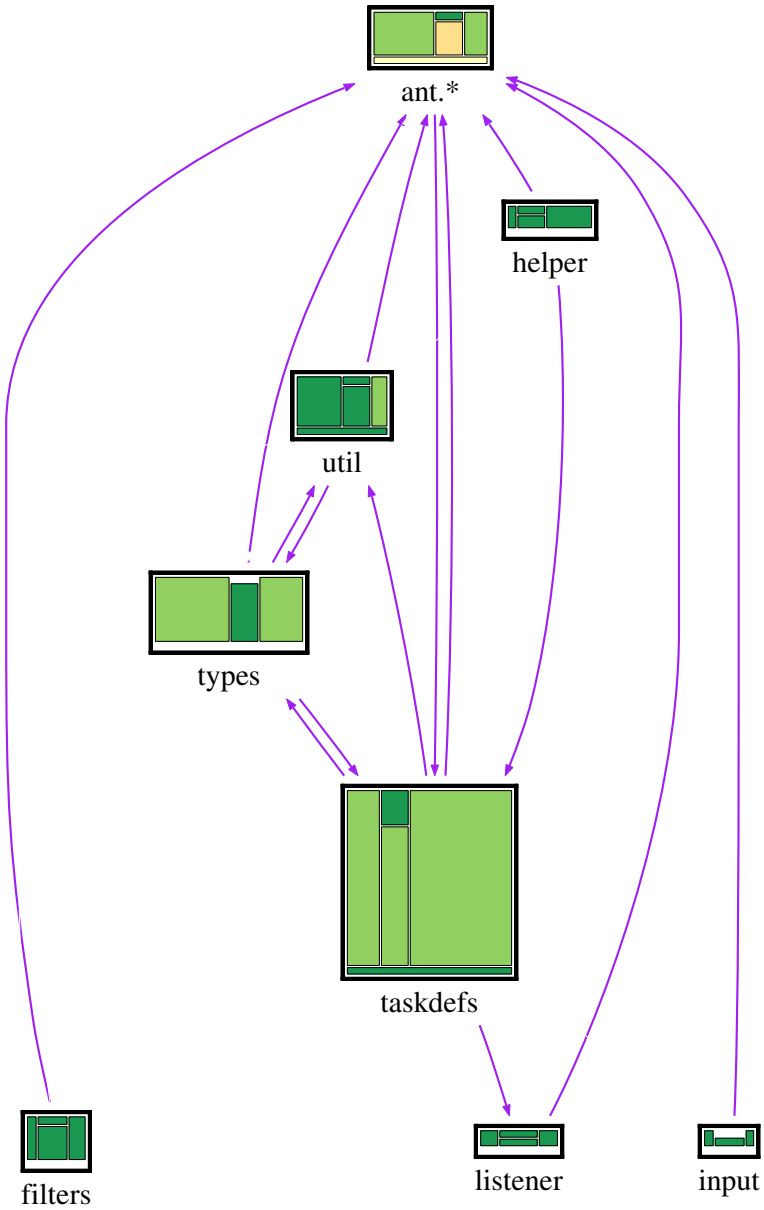


Figure 6.6: Assembly connector change in pkg. *tools.ant* for reuse purpose.

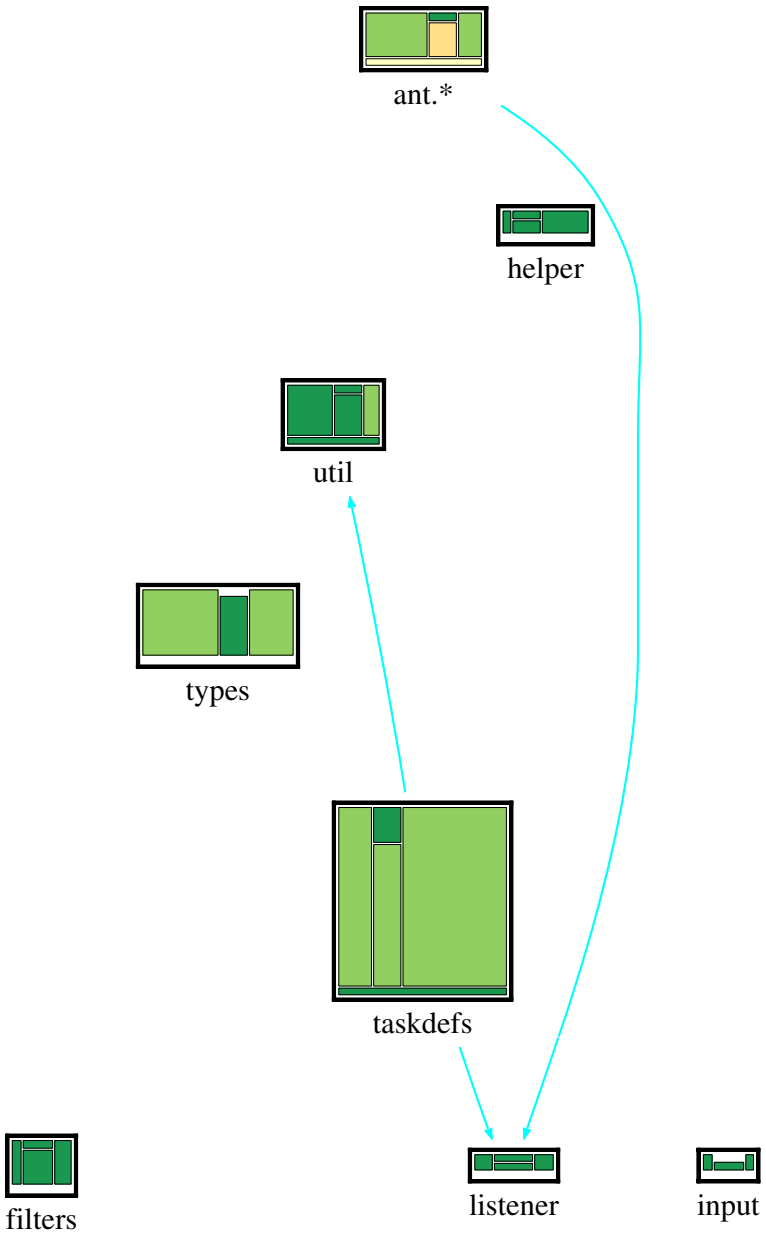


Figure 6.7: Assembly connector change in pkg. *tools.ant* for reimplementation purpose.

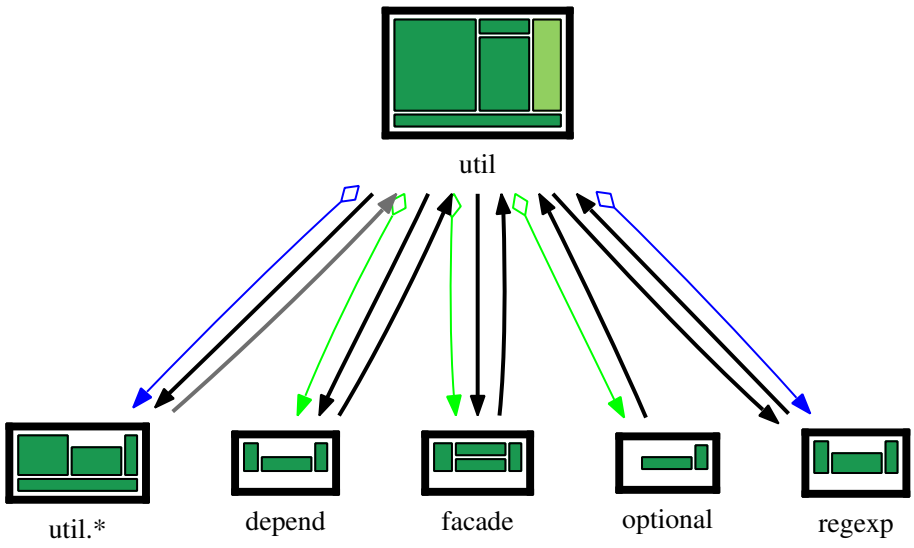


Figure 6.8: Delegation connector change in pkg. *util* caused by the co-change between components. The color shades are determined by $r_{fine-in}$ and $r_{fine-out}$ values.

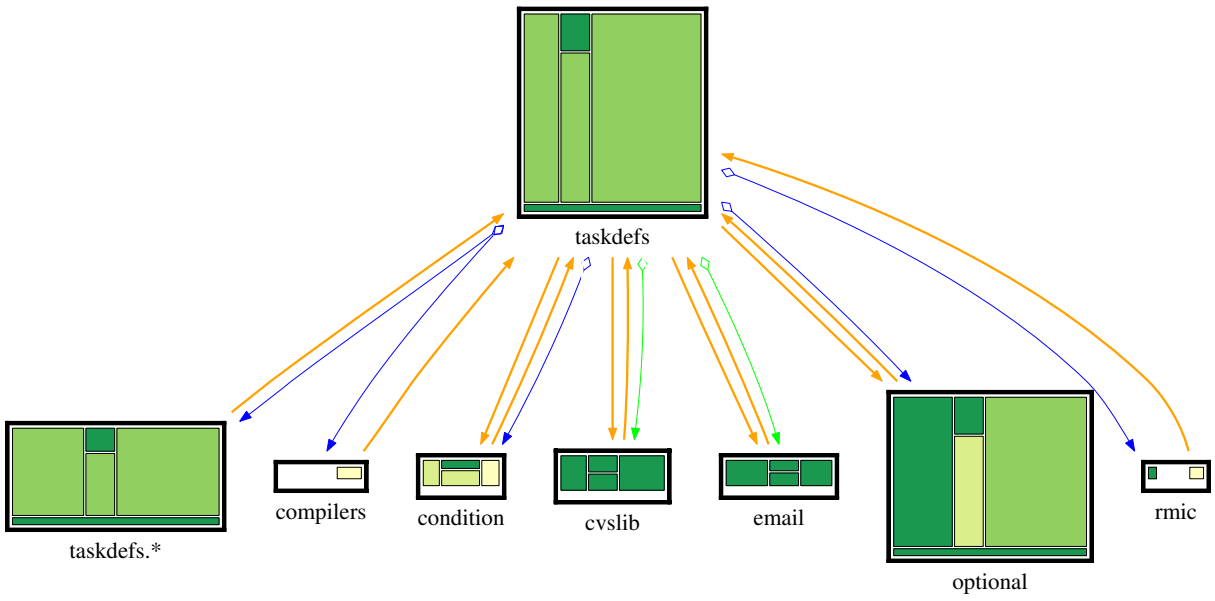


Figure 6.9: Delegation connector change in pkg. *taskdefs* for the reuse and reimplementation purpose.

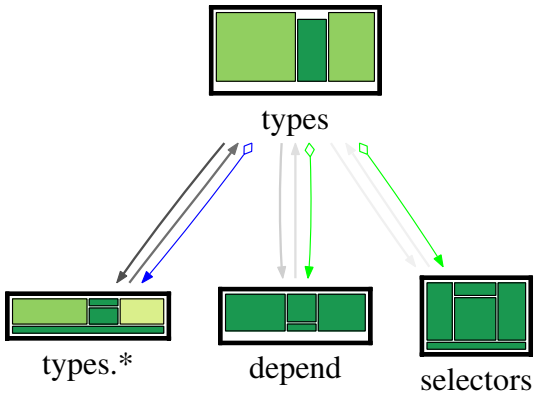


Figure 6.10: Delegation connector change in pkg. *types* caused by the co-change between components. The color shades are determined by $r_{coarse-in}$ and $r_{coarse-out}$ values.

6.6.3 Discussion

In the proposed evolution analysis, we take snapshots of a target system along its life time, and then analyze the difference between successive snapshots. With Hybrid Models, we can not only gain an overview of how a software system evolves over time, but also analyze the difference between two versions of a software system at the selected level of granularity.

However, this approach is sensitive to the choice of interval between snapshots. If the two snapshots are too close in time, the difference between them may not be significant at the selected level of granularity. A lot of effort is required to recover and analyze Hybrid Models, while little evolutionary information is revealed. If two snapshots are too far apart in time, then a lot of important design may be missed. Therefore, it is important to find a balance between accuracy and efficiency.

From our experience, the proposed approach is most effective when 50% to 80% of classes in the previous snapshot exists in the next one. In this case study, we created a Hybrid Model for each major release of Ant system, as we found that there is little package-level change in minor releases. Other systems may not share the same properties.

6.7 Related Work

Our work on architecture evolution analysis builds on prior work in two primary areas: change pattern detection and evolutionary visualization.

6.7.1 Change Pattern Detection

Godfrey and Zou employ origin analysis to detect structural change in procedural code [28]. They emphasize the analysis on call relationships, and classify the detected change into renaming, moving, splitting, and merging.

Xing and Stroulia presented a technique to recover co-evolution patterns among classes of an evolving software system [108]. They first detect and classify structural change of individual classes, and then apply association rules to distinguish three co-evolution patterns among classes.

Change of object-oriented systems is often interpreted in terms of refactorings. A number of refactoring detection approaches have been presented in past research. Gőrg and Weißgerber identify refactorings from the structural change that took place in the same CVS transaction [29]. Demeyer et al. detect possible refactoring activities by studying metrics change over successive versions of a software system [16]. Dig et al. apply a hashing technique to identify similar pairs of entities, and then perform an expensive semantic analysis to refine the initial candidates for refactorings [18]. Although they use different approaches and heuristics for refactorings, all of them can only recover the change patterns that they intend to identify.

6.7.2 Evolutionary Visualization

Current techniques for visualizing software evolution rely heavily on software metrics to produce condensed views.

The work most closely associated with our research is Lanza et al.'s use of polymetrics to visualize the history of classes [51]. Their method produces an evolution matrix, in which each cell represents a snapshot of a class, and the dimensions of the cell are determined by the metrics values of the class. They focus on the change of individual classes, while we

emphasize on the change of packages and their interrelationships.

Wu et al. also use matrices, called Evolution Spectrographs, to visualize the evolutionary measurements computed on subsequent versions [107]. They develop special coloring techniques to represent one particular property, e.g., fanin and fanout, of a target system at a selected level of granularity.

Pinzger et al. uses a Kiviat Diagram to graphically represent multiple metrics values of a source code entity (module, class, etc.), and their changes across several releases [79]. The coupling dependencies between source code entities are visualized as the edges between Kiviat Diagrams.

Rysselberghe et al. reconstruct evolution processes of existing software systems by exploiting clone detection techniques [83]. They identify adding, deleting and moving method in the program with aid of dotplots visualization.

Current research on object-oriented software evolution either relies on quantitative analysis, or is focused on semantic analysis at the class level of abstraction. In contrast to this state of art, our approach identifies change patterns at coarse-grained levels of abstraction.

6.8 Summary

In this chapter, we have presented an approach for studying the evolution of large, object-oriented software systems at a coarse level of granularity. We take snapshots of an object-oriented systems, represent each version of the system as a Hybrid Model, and detect software change at coarse-grained level by comparing two Hybrid Models. In our case study of the Apache Ant system, we show how Hybrid Models can help us to gain an overview of how the system evolved over time, identify possible architectural drift, and interpret detailed structural change in a selected scope at a preferred level of granularity.

Chapter 7

Conclusion

In this chapter, we summarize the contributions made in this thesis, discuss the benefits of our approach, and suggest future research directions.

7.1 Contributions

The system responsibilities of object-oriented software are achieved by a collection of collaborating objects. Classes provide a means to describe how objects behave during run-time. A reverse engineering tool, to support the comprehension and maintenance of object-oriented software systems, must capture both the collaboration between objects, and the dependencies between classes. However, the conventional coarse-grained representations, such as package diagrams, fail to treat both classes and objects as independent units due to the gap between objects and their classes.

The main contribution of this thesis is the creation of the Hybrid Model, which explicitly captures cross-package inheritance relations, as well as all possible dependencies between objects at a coarse-grained level of abstraction. Not only does a Hybrid Model provide an abstract and meaningful representation of an object-oriented system, it also provides a foundation for various kinds of analyses at the architectural level.

To investigate the efficacy of using Hybrid Models to reduce the complexity of a reverse engineering process, we applied the Hybrid Model in three different reverse engineering contexts: program comprehension, architectural dependency analysis, and architectural

change analysis.

1. Program comprehension.

   We applied the Hybrid Models to support both top-down and bottom-up comprehension strategies.

   To meet the comprehension needs of a bottom-up strategy, we use the Hybrid Model to assist maintainers in building abstractions by grouping classes into aggregate components. The boundary of an aggregate component summarizes the key properties of a group of classes as not only namespaces, but also types of objects. Thus, maintainers may form a coherent mental model of the group of classes.

   To meet the comprehension needs of a top-down strategy, we use the Hybrid Model to assist maintainers in creating a hierarchy of abstractions, and exploring the hierarchy in a divide-and-conquer manner. At a high level of abstraction, maintainers can focus on the external properties of aggregate components and the connectors between them without worrying about their internal implementation. At a low level of abstraction, the boundary of an aggregate component provide necessary context for the understanding of more detailed information.

   We conducted a case study to demonstrate that the Hybrid Models can be used in a real-world comprehension scenario.

2. Architectural dependency analysis.

   We presented and discussed a dependency analysis approach to help maintainers capture external properties of packages, and better understand the nature of their interdependencies. In this approach, we use the Hybrid Model to explicitly capture the cross-package inheritance relations, as well as all possible usage dependencies between objects at the package level.

   Based on the Hybrid Model, we presented a collection of architectural patterns involving aggregate components and connectors. Those patterns help analyze the design properties of key object-oriented concepts, such as data abstraction, encapsulation, and inheritance, which are vital to the quality of an object-oriented design.

3. Architectural change analysis.

   We presented and discussed an evolutionary analysis approach to uncover and analyze structural change of object-oriented systems at the architectural level. In this approach, we applied the Hybrid Model to explicitly modelling object-oriented software changes at coarse-grained levels. With the Hybrid Model, we were able not only to gain an overall picture of software evolution, but also to investigate the detailed structural change in a selected scope at a preferred level of granularity. We presented a collection of change patterns involving individual components and multiple components, and designed visualization techniques to support the identification of those patterns.

7.2 Future Work

This sections proposes future research directions based on the results of this thesis.

7.2.1 Improving the Accuracy of Hybrid Model

The Hybrid Model is extracted from source code based on static analysis, and so suffers from the usual conservatism about relation information. Low-level data and control dependency analysis may help narrow the set of potential targets of polymorphic calls, and reduce the impossible relations.

Dynamic execution traces have been proven to be useful in aiding object-oriented program comprehension. Currently, information of a program's behavior contained in a Hybrid Model is independent of inputs and the runtime environment. However, it is possible to slicing Hybrid Models using dynamic information. Since a assembly connector in a Hybrid Model represents a possible communication path between groups of objects, we believe that the Hybrid Model can be used to show execution trace at a high level of abstraction. Thus, it provides a platform to combine and compare static and dynamic information.

7.2.2 Visualization Support

The Hybrid Model is a graphic representation of object-oriented software systems. Thus, it is important to provide efficient tools to visualizing and exploring the Hybrid Models. Currently, we use Graphviz to perform the layout and rending [31]. However, it does not meet all of our needs for the navigation and exploration of Hybrid Models. For example, it is difficult to manipulate multiple Graphviz windows to support the navigation among views at different levels of abstraction. In addition, we integrate Graphviz with our toolkits by creating input and output files. There is little realtime collaboration between tools. The future work includes developing a special visualizer for the exploration of the Hybrid Models.

7.2.3 Architecture of A Product Family

Both architectural dependency analysis and change analysis presented in this thesis aim to understand the nature of a single target system. Future work includes performing case studies on the applications that belong to one product family or from the same problem domain, e.g., UML modelling tools. We hope to find similar architectural patterns and general trends in evolution.

Appendix A

Toolkit

In this chapter, we present a toolkit, Hybrid Model Toolkit, that we developed to extract, analyze, and present Hybrid Models. Figure A.1 depicts the overall architecture of the Hybrid Model Toolkit. Our toolkit is composed of the following parts:

Repository stores the information extracted from the code history. There are two level repository: Class-level Repository and Hybrid Model Repository.

- Class-level Repository stores the information at the class-level of abstraction. Its data structure is shown in Figure A.2. Currently, we store several primitive class interrelations, including *calls*, *instantiates*, *localVariable*, *fieldType*, *paramType*, *returnType*, etc.
- Hybrid Model Repository stores the extract Hybrid Models and the results of various analysis performed on Hybrid Models. Figure A.3 depicts the data structure of the Hybrid Model repository.

Extractor extracts the class-level information from the code history, and store the facts in the class-level repository. Currently, we support the extraction of facts from Java class files.

Hybrid Model Extractor provides an interactive environment, in which users can modify containment hierarchy of an object-oriented system. It can automatically generate

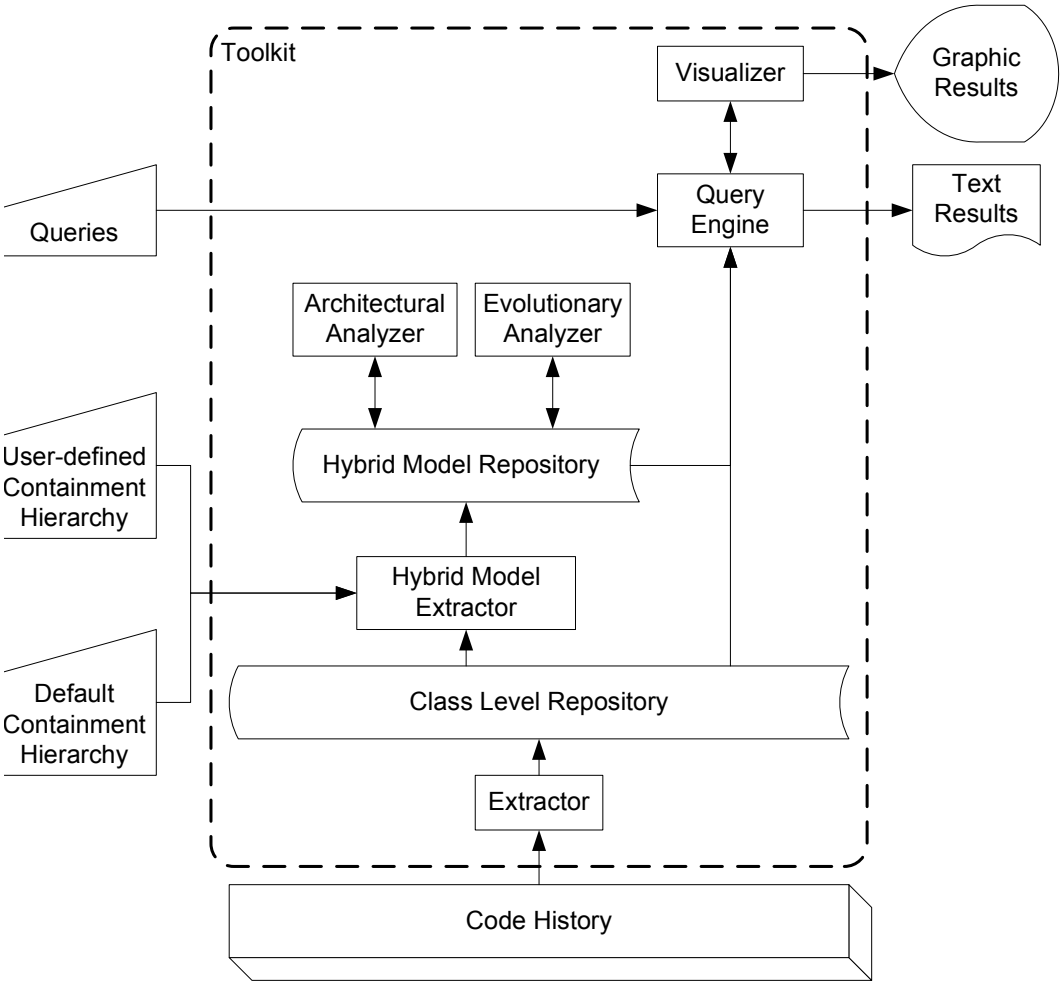


Figure A.1: The Architecture of Hybrid Model Toolkit

the Hybrid Model of an object-oriented software systems using a given containment hierarchy.

Analyzers performs analysis on Hybrid Models.

The repository of Hybrid Models provides a foundation for further analysis at the architectural level. The current design include two analyzers:

- Architectural Analyzer performs the architectural dependency analysis presented in Chapter 5. The analysis results are stored in the Hybrid Model repository in terms of the properties of model elements, as shown in Figure A.4. Most analysis results are numeric, and can be visualized using polymetrics visualization technique [51].
- Evolutionary Analyzer implements the architectural change analysis presented in Chapter 6. It automates the extraction and comparison of Hybrid Models, and store the difference between in terms of new Hybrid Models.

Query Engine provides an interactive environment, in which users can ask questions about the properties of Hybrid Models, and choose preferred the scope and granularity of subsequent visualization.

Visualizer produces various views of the Hybrid Model, which is presented using Graphviz [31].

The goal of the visualizer is to help maintainers to intuitively grasp the architecture or architectural change at coarse grained level. We apply polymetrics visualization technique to highlight the key properties of Hybrid Models. For example, the height of a component is determined by the size or the change size of its ghost classes, defined classes, and exiled classes, while the width of a component is determined by the size or the change size of its ports.

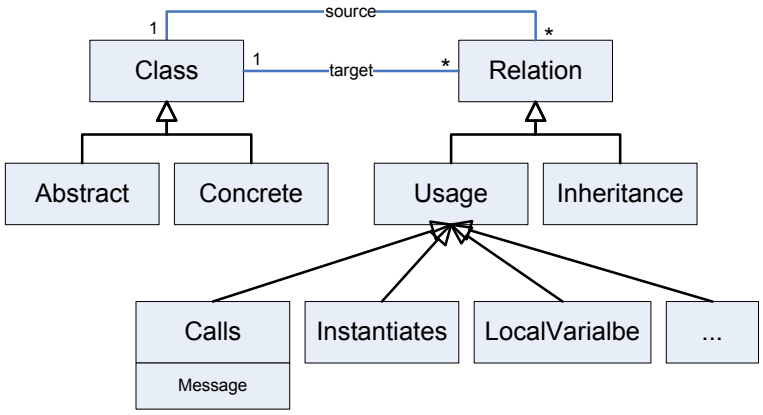


Figure A.2: Data Structure of Class Level Repository

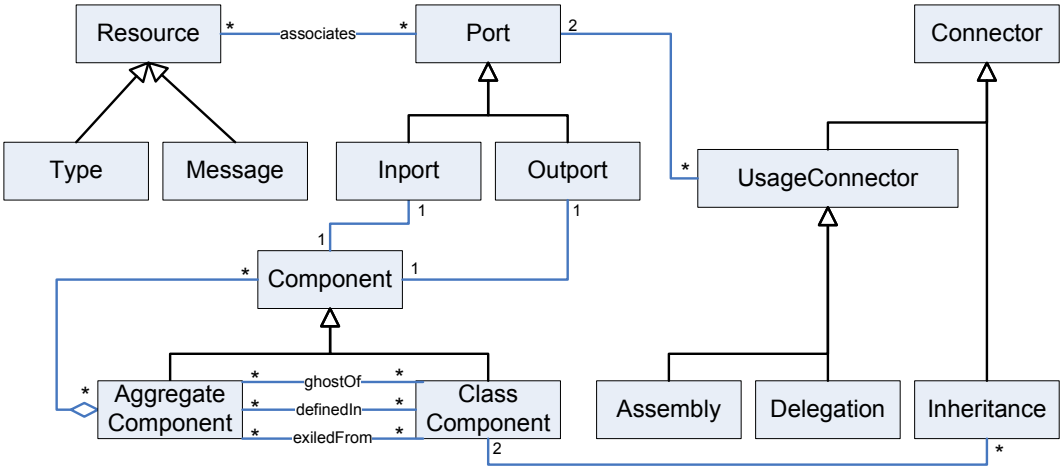


Figure A.3: Data Structure of Hybrid Model Repository

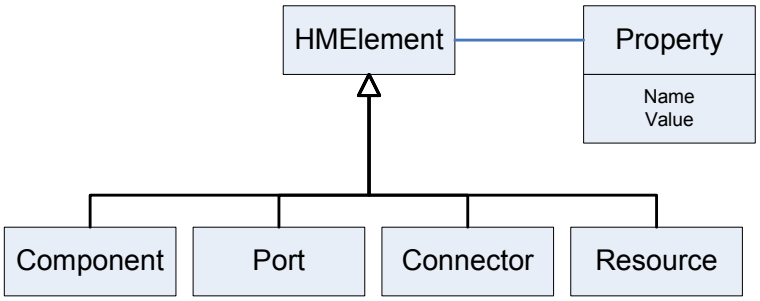


Figure A.4: Analysis results are stored in the Hybrid Model Repository

Bibliography

[1] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

[2] Giuliano Antoniol, Roberto Fiutem, and Luca Cristoforetti. Design pattern recovery in object-oriented software. In *Proceedings of the 6th International Workshop on Program Comprehension*, pages 153–160, 1998.

[3] Kent Beck and Ward Cunningham. A laboratory for teaching object oriented thinking. In *OOPSLA '89: Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, pages 1–6, New York, NY, USA, 1989. ACM Press.

[4] Pam Binns, Matt Englehart, Mike Jackson, and Steve Vestal. Domain-specific software architectures for guidance, navigation and control. *International Journal of Software Engineering and Knowledge Engineering*, 6(2):201–227, June 1996.

[5] Cornrad Bock. UML 2 composition model. *Journal of Object Technology*, 3(10):47–73, November-December 2004.

[6] Grady Booch. *Object-oriented analysis and design with applications (2nd ed.)*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[7] Lionel C. Briand, John W. Daly, and Jürgen K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, January/February 1999.

[8] Ruven Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543–554, 1983.

[9] Steve Burbeck. Applications programming in Smalltalk-80: How to use Model-View-Controller (MVC). URL: http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html, 1987.

[10] Roy H. Campbell, Nayeem Islam, David Raila, and Peter Madany. Designing and implementing choices: an object-oriented system in C++. *Communications ACM*, 36(9):117–126, 1993.

[11] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[12] Linore Cleveland. A program understanding support environment. *IBM System Journal*, 28(2):324–344, 1989.

[13] Peter Coad and Edward Yourdon. *Object-Oriented Analysis*. Yourdon Press, Upper Saddle River, NJ, USA, 1991.

[14] CPPX. Open source C++ fact extractor. URL: http://swag.uwaterloo.ca/ cppx, 2002.

[15] Dennis de Champeaux. Object-oriented analysis and top-down software development. In *ECOOP '91: Proceedings of the European Conference on Object-Oriented Programming*, pages 360–376, London, UK, 1991. Springer-Verlag.

[16] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA-00)*, pages 166–177, Minneapolis, MN, USA, Oct. 2000. ACM.

[17] Frank DeRemer and Hans Kron. Programming-in-the large versus programming-in-the-small. In *Proceedings of the International Conference on Reliable Software*, pages 114–121, New York, NY, USA, 1975. ACM Press.

[18] Danny Dig and Ralph Johnson. The role of refactorings in API evolution. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM-05)*, pages 389–398, Budapest, Hungary, Sept. 2005. IEEE.

[19] Johann Eder, Gerti Kappel, and Michael Schrefl. Coupling and cohesion in object-oriented systems. Technical report, University of Klagenfurt, 1992.

[20] Alexander Egyed and Phillipe B. Kruchten. Rose/Architect: a tool to visualize architecture. In *Proceedings of 32nd Annual Hawaii Conference on Systems Sciences*, 1999.

[21] R. K. Fjeldstad and W. T. Hamlen. Application program maintenance study - reports to our respondents. In Girish Parikh and Nicholas Zvegintzov, editors, *Tutorial of Software Maintenance*, pages 13–27. IEEE Computer Society Press, 1983.

[22] The Apache Software Foundation. The Apache Ant project. URL: http://ant.apache.org/.

[23] Martin Fowler. URL: http://www.refactoring.com/catalog/index.html.

[24] Martin Fowler. Reducing coupling. *IEEE Software*, 18(4):102–104, 2001.

[25] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1 edition, 1999.

[26] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional, 1995.

[27] David Garlan. Software architecture: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 91–101, New York, NY, USA, 2000. ACM.

[28] Michael W. Godfrey and Lijie Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, 2005.

[29] Carsten Görg and Peter Weißgerber. Detecting and visualizing refactorings from software archives. In *Proceedings of the 13th International Workshop on Program Comprehension (IWPC-05)*, pages 205–214, St. Louis, MO, USA, May 2005. IEEE.

[30] Michael M. Gorlick and Rami R. Razouk. Using weaves for software construction and analysis. In *ICSE '91: Proceedings of the 13th International Conference on Software Engineering*, pages 23–34, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[31] Graphviz. URL: http://www.graphviz.org/.

[32] John Grundy. Software architecture modelling, analysis and implementation with softarch. In *HICSS '01: Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)-Volume 9*, page 9051, Washington, DC, USA, 2001. IEEE Computer Society.

[33] Yann-Gaël Guéhéneuc, Rémi Douence, and Narendra Jussien. No java without caffeine: A tool for dynamic analysis of java programs. In *ASE '02: Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, pages 117–126, Washington, DC, USA, 2002. IEEE Computer Society.

[34] Yann-Gaël Guéhéneuc and Narendra Jussien. Using explanations for design patterns identification. In *Proceedings of the 1st IJCAI Workshop on Modelling and Solving Problems with Constrains*, pages 57–64, 2001.

[35] Ivan Herman, Guy Melançon, and M. Scott Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43, 2000.

[36] Dirk Heuzeroth, Thomas Holl, Gustav Högström, and Welf Löwe. Automatic design pattern detection. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC'03)*, pages 94–104, 2003.

[37] Charles Antony Richard Hoare. Communicating sequential processes. *Communications ACM*, 21(8):666–677, 1978.

[38] Richard C. Holt. An introduction to TA: The tuple-attribute language. URL: http://www.swag.uwaterloo.ca/pbs/papers/ta.html, 1997.

[39] Geir Magne Høydalsvik and Guttorm Sindre. On the purpose of object-oriented analysis. *SIGPLAN Notices*, 28(10):240–255, 1993.

[40] JDepend. URL: http://clarkware.com/software/JDepend.html.

[41] JEdit. URL: http://www.jedit.org/.

[42] JHotDraw. URL: http://www.jhotdraw.org/.

[43] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Object-Oriented Programming*, 1(2), 1988.

[44] Hermann Kaindl. Difficulties in the transition from oo analysis to design. *IEEE Software*, 16(5):94–102, 1999.

[45] Steven Klusener, Ralf Lämmel, and Chris Verhoef. Architectural modifications to deployed software. *Science of Computer Programming*, 54:143–211, 2005.

[46] Jürgen Koenemann and Scott P. Robertson. Expert problem solving strategies for program comprehension. In *Proceedings of the SIGCHI COnference on Human factors in Computing Systems: Reaching through Technology*, pages 125–130, 1991.

[47] Ralf Kollmann and Martin Gogolla. Capturing dynamic program behavior with UML collaboration diagrams. In *CSMR '01: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, pages 58–67, Washington, DC, USA, 2001. IEEE Computer Society.

[48] Christian Krämer and Lutz Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Proceedings of the 3rd Working Conference on Reverse Engineering*, pages 208–215, 1996.

[49] Bruno Laguë and Michel Dagenais. An analysis framework for understanding layered software architectures. In *IWPC '98: Proceedings of the 6th International Workshop*

on Program Comprehension, page 37, Washington, DC, USA, 1998. IEEE Computer Society.

[50] Danny B. Lange and Yuichi Nakamura. Object-oriented program tracing and visualization. *Computer*, 30(5):63–70, 1997.

[51] Michele Lanza. *Object-oriented Reverse Engineering: Coarse-grained, Fine-grained and Evolutionary Software Visualization*. PhD thesis, University of Bern, May 2003.

[52] Meir M. Lehman, Juan F. Ramil, Paul D. Wernick, Dewayne E. Perry, and Wladyslaw M. Turski. Metrics and laws of software evolution - the nineties view. In *METRICS'97: Proceedings of the 4th International Symposium on Software Metrics*, page 20, Washington, DC, USA, 1997. IEEE Computer Society.

[53] Bennet P. Lientz, E. Burton Swanson, and Gerry E. Tompkins. Characteristics of application software maintenance. *Communications of the ACM*, 21(6):466–471, 1978.

[54] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, 1995.

[55] David C. Luckham, James Vera, Doug Bryan, Larry Augustin, and Frank Belz. Partial orderings of event sets and their application to prototyping concurrent, timed systems. *Journal of Systems and Software*, 21(3):253–265, 1993.

[56] Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. In *SIGSOFT '96: Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 3–14, New York, NY, USA, 1996. ACM Press.

[57] Robert C. Martin. OO design quality metrics: An analysis of dependencies. URL: http://www.objectmentor.com/publications/oodmetrc.pdf, 1994.

[58] Robert C. Martin. The dependency inversion principle. *The C++ Report*, 8(6):61–66, 1996.

[59] Robert C. Martin. Granularity. C++ Report, 1996.

[60] Robert C. Martin. The open-closed principle. In *More C++ gems*, pages 97–112. Cambridge University Press, New York, NY, USA, 2000.

[61] Nenad Medvidovic and David S. Rosenblum. Assessing the suitability of a standard design method for modelling software architectures. In *Proceedings of the First Working IFIP 52 Conference on Software Architecture (WICSA1)*, pages 161–182, San Antonio, TX, USA, 1999.

[62] Nenad Medvidovic, David S. Rosenblum, and Richard N. Taylor. A language and environment for architecture-based software development and evolution. In *ICSE '99: Proceedings of the 21st International Conference on Software Engineering*, pages 44–53, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[63] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[64] George A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*, 63:81–97, 1956.

[65] Joaquin Miller and Rebecca Wirfs-Brock. How can a subsystem be both a package and a classifier? In *UML'99: Proceedings of the Second International Conference on the Unified Modelling Language*, pages 584–597. IEEE Computer Society Press, 1999.

[66] Hausi A. Müller, Mehmet A. Orgun, Scott R. Tilley, and James S. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5(4):181–204, December 1993.

[67] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: bridging the gap between source and high-level models. In *SIGSOFT '95: Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 18–28, New York, NY, USA, 1995. ACM Press.

[68] NDepend. URL: http://www.ndepend.com/.

[69] Ulrich Nickel, Jörg Niere, and Albert Zündorf. The FUJABA environment. In *International Conference on Software Engineering*, pages 742–745, 2000.

[70] Jörg Niere, Wilhelm Schäfer, Jörg P. Wadsack, Lothar Wendehals, and Jim Welsh. Towards pattern-based design recovery. In *Proceedings of the 24th International Conference on Software Engineering*, pages 338–348, 2002.

[71] John T. Nosek and Prashant Palvia. Software maintenance management: changes in the last decade. *Journal of Software Maintenance*, 2(3):157–174, 1990.

[72] OMG. Unified Modelling Language: Superstructure (Version 2.0). http://www.omg.org, 7 2005.

[73] OMG. Architecture-driven modernization. URL: http://adm.omg.org/, 7 2008.

[74] Omondo EclipseUML. URL: http://www.omondo.com/.

[75] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[76] David Lorge Parnas. Software aging. In *ICSE '94: Proceedings of the 16th International Conference on Software Engineering*, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[77] Wim De Pauw, Erik Jensen, Nick Mitchell, Gary Sevitsky, John M. Vlissides, and Jeaha Yang. Visualizing the execution of java programs. In *Revised Lectures on Software Visualization, International Seminar*, pages 151–162, London, UK, 2002. Springer-Verlag.

[78] Nancy Pennington. Comprehension strategies in programming. In *Empirical Studies of Programmers: Second Workshop*, pages 100–112. Ablex Publishing Corporation, 1987.

[79] Martin Pinzger, Harald Gall, Michael Fischer, and Michele Lanza. Visualizing multiple evolution metrics. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software Visualization*, pages 67–75, New York, NY, USA, 2005. ACM.

[80] Claudio Riva and Jordi Vidal Rodriguez. Combining static and dynamic views for architecture reconstruction. In *CSMR '02: Proceedings of the 6th European Conference on Software Maintenance and Reengineering*, pages 47–56, Washington, DC, USA, 2002. IEEE Computer Society.

[81] IBM Rational Rose. URL: http://www.ibm.com/software/rationa.

[82] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modelling Language Reference Manual*. Addison-Wesley Professional, December 1998.

[83] Filip Van Rysselberghe and Serge Demeyer. Reconstruction of successful software evolution using clone detection. In *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, page 126, Washington, DC, USA, 2003. IEEE Computer Society.

[84] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. In *OOPSLA '05: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and applications*, pages 167–176, New York, NY, USA, 2005. ACM Press.

[85] Hans Albrecht Schmid. Creating the architecture of a manufacturing framework by design patterns. *SIGPLAN Notices*, 30(10):370–384, 1995.

[86] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4):314–335, 1995.

[87] Susan Elliott Sim, Charles L. A. Clarke, Richard C. Holt, and Anthony M. Cox. Browsing and searching software architectures. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 381–390, Washington, DC, USA, 1999. IEEE Computer Society.

[88] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. An examination of software engineering work practices. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, pages 21–35, 1997.

[89] Elliot Soloway, Beth Adelson, and Kate Ehrlich. Knowledge and processes in the comprehension of computer programs. In R. Glaser M. Chi and M. Farr, editors, *The Nature of Expertise*, pages 129–152. 1988.

[90] J. Mike Spivey. *The Z notation: a reference manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[91] Margaret-Anne D. Storey, F. David Fracchia, and Hausi A. Müller. Cognitive design elements to support the construction of a mental model during software visualization. In *IWPC '97: Proceedings of the 5th International Workshop on Program Comprehension (IWPC '97)*, page 17, Washington, DC, USA, 1997. IEEE Computer Society.

[92] Andrew Sutton and Jonathan I. Maletic. Mappings for accurately reverse engineering UML class models from C++. In *WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering*, pages 175–184, Washington, DC, USA, 2005. IEEE Computer Society.

[93] SwagKit. URL: http://www.swag.uwaterloo.ca/.

[94] Clemens Szyperski. *Component software: beyond object-oriented programming*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.

[95] Apache Tomcat. URL: http://tomcat.apache.org/.

[96] Borland Together. URL: http://www.borland.com/together/.

[97] Paolo Tonella. Concept analysis for module restructuring. *IEEE Transactions on Software Engineering*, 27(4):351–363, 2001.

[98] Paolo Tonella and Alessandra Potrich. Reverse engineering of the UML class diagram from C++ code in presence of weakly typed containers. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 376–385, 2001.

[99] Paolo Tonella and Alessandra Potrich. *Reverse Engineering of Object Oriented Code*. Springer, 2005.

[100] Vassilios Tzerpos and Richard C. Holt. ACDC: An algorithm for comprehension-driven clustering. In *WCRE '00: Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, pages 258–267, Washington, DC, USA, 2000. IEEE Computer Society.

[101] Anneliese von Mayrhauser and A. Marie Vans. Program understanding: Models and experiments. In Marvin V. Zelkowitz, editor, *Advances in Computers*, pages 1–38. Academic Press, 1995.

[102] Robert J. Walker, Gail C. Murphy, Bjørn Freeman-Benson, Darin Wright, Darin Swanson, and Jeremy Isaak. Visualizing dynamic software system information through high-level models. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 271–283, New York, NY, USA, 1998. ACM Press.

[103] Jos B. Warmer and Kleppe Anneke G. *The Object Constraint Language: Precise Modelling With UML*. Addison-Wesley Professional, October 1998.

[104] Theo A. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *WCRE'97: Proceedings of the Fourth Working Conference on Reverse Engineering*, pages 33–43, Washington, DC, USA, 1997. IEEE Computer Society.

[105] Norman Wilde, Paul Matthews, and Ross Huitt. Maintaining object-oriented software. *IEEE Software*, 10(1):75–80, 1993.

[106] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-oriented Software*. Prentice Hall, Englewood Cliffs, N.J., USA, 1990.

[107] Jingwei Wu, Richard C. Holt, and Ahmed E. Hassan. Exploring software evolution using spectrographs. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE-04)*, pages 80–89, Delft, Netherlands, Nov. 2004. IEEE.

[108] Zhenchang Xing and Eleni Stroulia. Understanding the evolution and co-evolution of classes in object-oriented systems. *International Journal of Software Engineering and Knowledge Engineering*, 16(1):23–52, 2006.