System-level Usage Dependency Analysis of Object-Oriented Systems

Xinyi Dong and Michael W. Godfrey
Software Architecture Group (SWAG)
David R. Cheriton School of Computer Science
University of Waterloo
{xdong, migod}@uwaterloo.ca

Abstract

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.*

In this paper, we propose an approach, called the High-level Object Dependency Graph (HODG), that captures all possible usage dependencies among coarse-grained entities. Based on the new model, we further propose a set of dependency analysis methods. Finally, we present an exploratory case study using HODGs — supported by an automated analysis tool — of the Apache Ant build system; we show how HODG analysis can help maintainers capture external properties of coarse-grained entities, and better understand the nature of their interdependencies.

1 Introduction

An object-oriented system is composed of a collection of communicating objects that cooperate with one another to achieve some desired goals. Similar objects form classes, which provide the static description of the properties and behaviors that their instances will have. Therefore, extracting, analyzing, and modelling classes/objects and their relationships is of key importance in acquiring in-depth understanding of object-oriented software systems. However, when dealing with complex object-oriented systems, maintainers can easily be overwhelmed by the large number of classes/objects and the high degree of interdependencies among them.

A commonly used strategy to address the scalability problem is to partition the set of all classes/objects into coarse-grained container entities and then analyze their interrelationships. Most reverse engineering tools create package diagrams by grouping classes into packages. Sangal and Waldman use the Dependency Structure Matrix (DSM) to present package dependencies and help maintainers to identify unexpected dependencies [20]. JDepend [9] and NDepend [17] use package dependencies to calculate design quality metrics, such as afferent and efferent coupling [13]. While grouping classes into packages may improve understandability of classes and their interrelations, at the same time it also hinders the comprehensibility of objects as independent units because the static description of an object can end up being distributed across multiple coarse-grained entities due to inheritance. AVID[22] and Program Explorer [12] supports the visualization of the dynamic execution trace at different levels of granularity by grouping objects. However, runtime information must be mapped to source code, which programmers can manipulate directly.

In this paper, we present a new model, called the High-level Object Dependency Graph (HODG), to represent object-oriented systems at a high level of abstraction. Instead of grouping programming language classes, we aggregate the complete static description of software objects so that each coarse-grained entity represents a set of objects. A dependency between coarse-grained entities indicates a possible relationship between the objects they represent. We further propose a set of methods to analyze object-oriented system over the new model. Our analysis results can help maintainers to answer the questions that are commonly asked at high levels of abstraction, such as:

- What are the externally visible properties of a coarse-grained entity? What are the responsibilities or the functionalities that the entity provides? What collaborations does the entity require?
- Is there a dependency between two coarse-grained entities? Are the two entities tightly coupled? What causes the dependency?
- How is a service shared among entities? Which entities

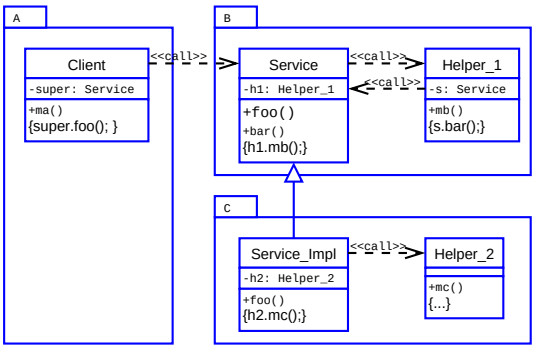

Figure 1. The Class Diagram of An Example Program. Method invocations are viewed as stereotyped dependencies.

provide the service and which entities use the service?

The remainder of this paper is organized as follows. The next section introduces the HODG approach. Section 3 describes the dependency analysis methods over the new model. In section 4, we apply these methods in an exploratory case study that show how maintainers can benefit from the dependency analysis approach. Finally, section 5 summarizes our work and suggests future research directions.

2 High-level Object Dependency Graphs

The relationships between classes can be roughly divided into two categories: inheritance and usage dependencies. Usage dependencies, including *instantiates* and *calls*, must be interpreted in the context of a 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 1, an object of `Client` could send messages to an object of `Service_Impl`, and there could be interactions between an object of `Service_Impl` and an object of `Helper_1`. These implicit *calls* dependencies, as well as the explicit *calls* dependency from `Service_Impl` to `Helper_2`, provide a complete high level description of how an object of `Service_Impl` interacts with other objects. If `Service` and `Service_Impl` are separated into different packages, neither package capture the complete behaviors of the objects instantiated from `Service_Impl`.

To address the problem, we propose a new kind of model, called the High-level Object Dependency Graph (HODG). The key difference between a HODG and a package diagram is that each coarse-grained entity in a HODG includes the complete static descriptions of the objects that are defined and implemented within it. Coarse-grained entities are connected by all possible usage dependencies, while inheritance relations are eliminated at the coarse-grained level. 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. The construction of the HODG 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, `Service` in Figure 1 is removed from `B`.

2. For each concrete class in each container, all ancestors[1] of that class are pulled into the container. For example, `Service` is pulled into `C`. We note that the containment relation in an HODG is not a tree, as ancestor classes can belong to more than one container; in UML terminology, the containers in an HODG are *aggregates* rather than *compositions* of their parts. An abstract class with multiple implementations in different containers will appear in each such container.

3. Any usage dependencies between classes that are not in the same container become dependencies of their respective 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 2 shows the HODG of the example program at the top-level abstraction.

In the next section, we present the notation of HODG in order to facilitate the discussion on dependency analysis in the following sections.

2.1 Notation

Resources, components, ports, and connectors are the four key elements of a HODG.

2.1.1 Resource

We define a resource [2] 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, `bar` is a method resource, and `Service` is a type resource. In this paper, we show only type resources, and consider variable and method resources to be parts of type resources.

[1] We focus on the application classes within the system under consideration, and ignore any inheritance relations to library classes, such as `java.lang.Object`

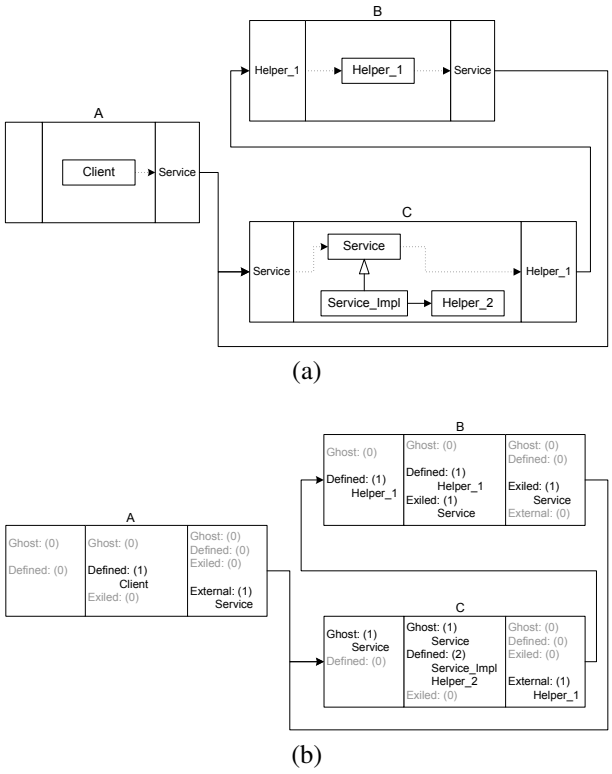

Figure 2. The HODG for the example program at the top level. (a) shows the expanded view, and (b) shows the collapsed view.

2.1.2 Component

A component [16, 18] 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. Each component corresponds to either a class or an aggregate container as described in the previous section.

A component, represented visually as a box, can be shown either collapsed or expanded. In Figure 2 (a), all components are expanded, and their internal structures are visible. In Figure 2 (b), all components are collapsed and labelled with the names of the resources it contains.

To a given aggregate component, a resource falls into one of four categories:

- *Defined* resources, which are declared and defined in the component. For example, `Helper_1` is a defined class of component `B`.

- *Ghost* resources, which were originally declared elsewhere but are implemented in the component. For example, `Service` is a ghost resource of component `C`.

- *Exiled* resources, which are declared but not implemented in the component. For example, `Service` is an exiled resource of component `B`.

- *External* resources, which are declared and defined outside the component. For example, `Service` is an external resource of component `A`.

2.1.3 Ports

Ports are the interfaces through which a component interacts with its environment. We distinguish two kinds of ports: inports and outports [18].

- An *inport*, visually represented as a box on the left side 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. The inport of a component may consist of defined resources and ghost resources of the component. In Figure 2, component `C` provides a ghost resource `Service`.

- An *outport*, visually represented as a box on the right side of a component, describes the resources that the component requires from others. The outport of a component may consist of ghost, defined, exiled, and external resources of the component. For example, in Figure 2, component `A` requires an external resource `Service`.

2.1.4 Connectors

A connector [16, 18] specifies the interrelationship among two components. There are three types of connectors: inheritance, delegation [18], and assembly [18] connectors.

- An *inheritance* connector, visually represented as a solid line with an empty arrowhead, specifies the inheritance relation between two classes. In an HODG, inheritance can exist only within aggregate components; they cannot cross component boundaries, since all inheritance ancestors have already been pulled in to the component.

- 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 components to the corresponding interfaces of its container components. For example, component `C` provides resources

that `Service` provides, and requires resources that `Service` 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 components. For example, component `A` uses the resource `Service` provided by component `C`.

In this paper, we focus on the analysis of assembly connectors between aggregate components because they capture the collaboration between components at the same level of granularity. The resources associated with an assembly connector can be divided into three types:

- *Declared-in-server* (DIS) resources of an assembly connector are defined resources of the server component. For example, `Helper_1` is a DIS resource of the connector from `C` to `B`, as `Helper_1` is a defined resource of `B`.
- *Declared-in-client* (DIC) resources of an assembly connector are defined or exiled resources in the source of the connector. For example, `Service` is a DIC resource of the connector from `B` to `C`.
- *Declared-in-thirdparty* (DIT) resources of an assembly connector are originally defined in a third component. For example, `Service` is a DIT resource of the connector from `A` to `C`.

3 Dependency Analysis

While HODGs can present overviews of object-oriented systems, they also provide a basis for dependency analysis at the coarse-grained level of abstraction. Our approach is to analyze high-level dependencies from two perspectives: the external properties of components, and the characteristics of connectors.

3.1 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 the aggregate component. It is possible that an aggregate component may satisfy multiple patterns at the same time.

3.1.1 Internal Structure and Cross-Package Inheritance

Although the internal details of an aggregate component are typically not considered at a coarse-grained level of abstraction, the number of the contained resources and their types

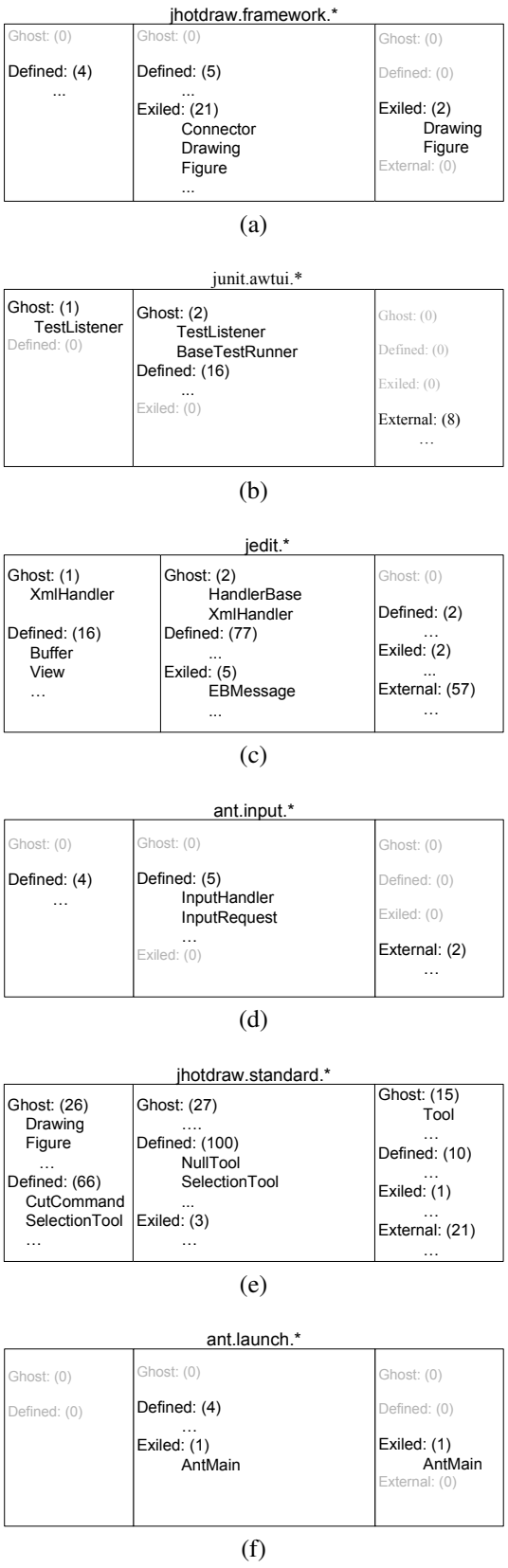

Figure 3. Example Aggregate Components

reveal the cross-package inheritance that the original package of an aggregate component involves, and provide clues for the role the component plays in the software system. We define four categories of aggregate components, based on their internal structure: **interface**, **implementation**, **mixed**, and **self-implemented**.

An aggregate component that contains only exiled resources is called an **interface** component. Such a component declares a set of resources, but provides no implementations for them. Those exiled resources are usually deliberately designed in order to allow a group of possible behaviors. For example, in Figure 3(a), 21 out of 26 resources declared in `jhotdraw.framework.*` are exiled resources. All of them are Java interfaces. This component defines the framework for the JHotDraw program [11].

An aggregate component that contains only ghost resources 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 resources. For example, `junit.awtui.*` in Figure 3(b) has two ghost classes. It implements a listener for test progress, and reuses the code in `BaseTestRunner` via inheritance.

An aggregate component that contains both ghost and exiled resources is called a **mixed** component. A mixed component often has complicated structure. Like an interface component, it declares a group of classes, but provides incomplete description of their behaviors. At the same time, it implements or reuses the resources that are defined in other components. For example, `jedit.*`, shown in Figure 3(c), is a mixed component. It contains the main logic of the JEdit system [10], and is coupled with many other components of the system.

A **self-implemented** component has neither ghost nor exiled resources. It declares resources and provides complete descriptions for their behaviors. A utility component is often self-implemented. For example, `ant.input.*` in Figure 3(d) is a self-implemented component that provides resources to handle user inputs.

3.1.2 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 rationales behind inheritance.

An inport may consist of defined resources and ghost resources of the component. Based on the types of the resources associated to the inports, a component can be classified as **directly used**, **indirectly used**, or a combination of the two.

A **directly-used** component exports only defined resources. Those defined resources are known to other components, and thus are important at the coarse-grained level. A utility component, such as `ant.input.*` in Figure 3(d), is directly used by others.

An **indirectly-used** component exports only ghost resources. Those ghost 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 coarse-grained level. For example, `junit.awtui.*` in Figure 3(b), is known to others as the component that provide the service of `TestListener`, while its derivative is not exported by the component.

An inport revealing both ghost and defined resources often results from mixed design intentions. For example, `jedit.*` in Figure 3(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 provide some defined resources, such as `Buffer` and `View`.

3.1.3 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 [19]. According to the number of exported resources, a component can have an **empty**, **thin** or **wide** inport.

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 3(f), its empty inport indicates that it is the starting point of the system.

A **thin** inport of a component exports a small portion of the resources the component contains, while keeps most resources hidden from the outside. A thin inport implies the designers' intention to hide complicated implementation details, such as a class that serves as an interface to a large, complicated component within a system. For example, `junit.awtui.*` in Figure 3(b) encapsulates 18 classes, but it exports only 1 resource.

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 3(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 3(e) also has a wide inport; it provides a standard implementation of the resources that are originally defined in `jhotdraw.framework`.

3.1.4 Outports and Reuse

The outport of an aggregate component reveals the services that the component requires from others. The outport may associate with defined, ghost, exiled, and external resources of the components. There are three patterns that describe the components regarding their outports: **self-sufficient**, **open-for-extension**, and **open-for-variation** components.

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 3(d) is self-sufficient.

An **open-for-extension** component requires exiled resources. This pattern indicates that the required resources are "open for extension" [15], and could have an unlimited collection of possible behaviors. For example, `jhotdraw.framework.*` in Figure 3(a) requires the exiled resources `Drawing` and `Figure`, both of which have dozens of implementations in other components of JHotdraw system.

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 3(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.*`.

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.

3.2 Connector Analysis

An assembly connector between two components exists if the static description in one component uses the resources that are a part of the static description in the other. 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.

3.2.1 Types of Assembly Connectors

A component may require a set of resources from the others. Based on the types of the associated resources, an assembly connector can be classified into four types:

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.

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.

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.

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.

3.2.2 Strength of Assembly Connectors

Assembly connectors in an HODG 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; one class is related to another by an aggregation. These mechanisms also constitute interaction coupling and component coupling [1, 3]. Although inheritance relations in a HODG are hidden within components, a HODG model of a system makes evident most coupling dependencies between components.

We measure the strength of an assembly connectors using the numbers of its associated DIS, DIC and DIT resources; this number reflects how many type resources

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 1. The relationship between assembly connectors and package dependencies. A and B are HODG components. P_A and P_B are the original packages.

(classes) 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.

3.2.3 Connectors and Package Dependencies

Classes of object-oriented programs are often organized into packages. Both inheritance 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 [5, 14].

Unlike a package diagram, an HODG does not directly capture dependencies between original packages since classes are regrouped into aggregate components during its construction. However, most package dependencies can be easily derived from the assembly connectors. Table 1 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 causes 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.

A provide-structure assembly connector implies that designer's intention to apply the "Dependency Inversion Principle" [14]. 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" [6] and "Move Class" [4] refactoring techniques.

4 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 [21], 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.

4.1 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 HODGs based on the package containment hierarchy, and automatically creates visualizations of the HODGs using GraphViz [8].

The top-level HODG of Apache Ant is composed of 14 components and 45 assembly connectors between them. During the construction of the HODG, 55 abstract classes (41 distinct classes) are duplicated in 8 top-level 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.

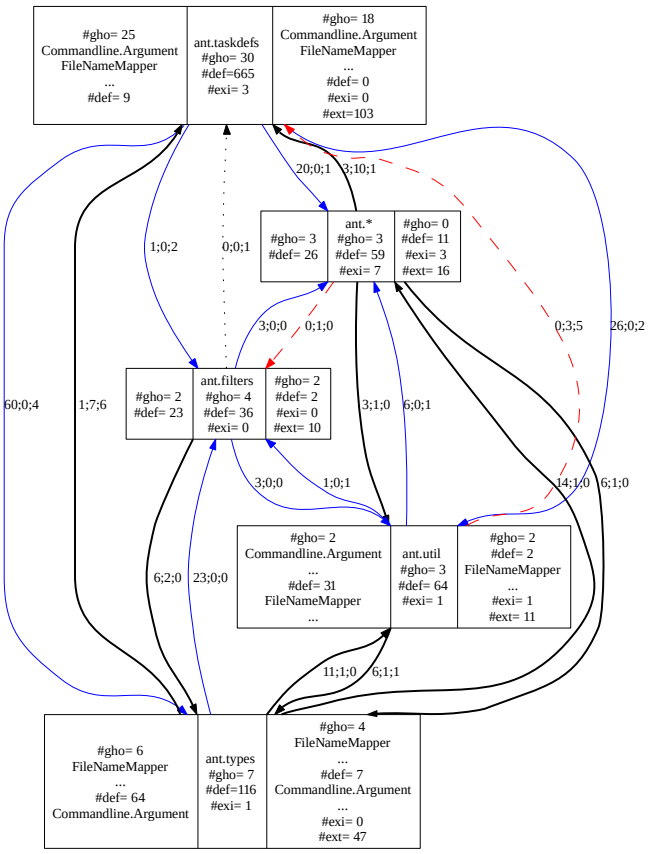

Figure 4. Assembly Connectors among 5 Top-level Components. Bold solid lines represent PSRD connectors; Blue solid lines represent RD connectors; Red dashed lines represent PS connectors. Black dotted lines represent RB connectors. Each connector is labelled with strength(#DIS; #DIC; #DIT).

A total 178 type resources (classes) are shared among the 14 top-level aggregate components. 5 of them provide more than 80% of those shared resources, and compose the backbone of the Ant system. Figure 4 depicts the interrelation among the five components. `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.

The remaining 9 components are either small or have thin inports. In HODG, the inports of components indicate the responsibilities that they reveal to the outside through usage dependencies. Therefore, we are able to peek into the responsibilities of those top-level components by reviewing only a few classes associated with their inports.

4.2 Refactoring Opportunities

Dependency analysis over HODGs not only help to identify architectural significant information, such as the external properties of groups of objects and the collaboration between them, but also can help 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.

4.2.1 Intimate Collaboration

An assembly connector specify a client/server relationship between two components. Sometime, two or more component can be both clients and servers of the same resource at the same time. This happens when an abstract concept is implemented in multiple components, but is always manipulated through the abstract concept. This may indicate poor modularization.

For example, Figure 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" [6] refactoring technique can be applied to reuse code of `Argument` via composition instead of via inheritance [7].

The three top-level 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 [7]. `FileNameMapper` is the base class, and `ant.util` contains the composite classes and most leaf classes. `ant.types` and `ant.taskdefs` each has a couple of leaf classes. Maintainers need to determine whether it is necessary to define similar objects in different packages. "Replace Inheritance with Delegation" [6] and "Move Class" [4] refactoring techniques can be applied to regroup classes.

4.2.2 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 change other components. As software ages, the original design intention may be violated. In this case study, we found such a conflicted design intention in `ant.taskdefs`.

Component `ant.taskdefs` in Figure 4 is likely designed to be used indirectly. It is a giant component with relative thin port. Most of its exported resources are ghost resources. Figure 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 1.6.

Refactoring techniques [6] 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 `RedirectorElement` 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 [6] 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.

4.2.3 Fat Connectors

A strong assembly connector between two components leads to a wide inport and a wide outport. This often indicates that their corresponding packages are tightly coupled.

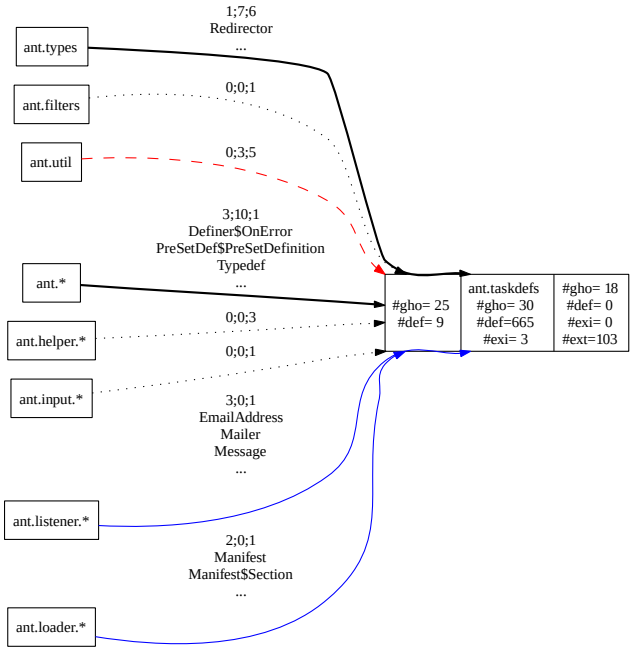

Figure 5. Incoming Assembly Connectors of `ant.taskdefs`.

Reducing the strength of an assembly connector may reduce the coupling between the original packages.

`ant.filters` as shown in Figure 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 `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 of `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" [6] refactoring techniques to further decouple the two components and allow the implementations of the same abstract concept remain in the same package.

4.3 Summary of Case Study

HODGs provide a promising approach to visualize and analyze object-oriented systems at coarse-grained levels of abstraction. An HODG captures all possible usage de-

pendencies between components. As cross-package inheritances are removed, we not long 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 HODGs 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 understand its responsibilities, and know the component as a whole.

HODGs can also help maintainers to recover some original design intentions of the system, including data abstraction, program logic encapsulation. Furthermore, HODGs can help to identify potential design problems – such as tight coupling and compilation dependency cycles – and suggest possible solutions to the problems.

The major disadvantage of the HODG approach is that it is based on static analysis, and so suffers from the usual conservatism about dependency information. Not all polymorphically possible dependencies may in fact be logically possible depending on the logic of the source code; low-level data and control dependency analysis may help narrow the set of potential targets of polymorphic calls, and reduce the impossible usage dependencies.

5 Conclusions and Future Work

In this paper, we use the High-level Object Dependency Graph (HODG) to model all possible usage dependencies among coarse-grained entities. Based on the new model, we further propose a set of methods to analyze high-level dependency from three perspectives: the external properties of component, the usage of resource, and the characteristics of connectors. Our exploratory case study 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.

Future work includes performing additional case studies. This will also be used in identifying the relation between connector strength and traditional coupling measurements. In addition, we plan to apply the proposed analysis approach on different versions of an object-oriented system to understand how software architecture changes throughout the software life cycle.

References

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

[2] F. DeRemer and H. 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.

[3] J. Eder, G. Kappel, and M. Schrefl. Coupling and cohesion in object-oriented systems. Technical report, University of Klagenfurt, 1992.

[4] M. Fowler. Refactorings in alphabetical order. URL: http://www.refactoring.com/catalog/index.html.

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

[6] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1 edition, 1999.

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

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

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

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

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

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

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

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

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

[16] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93, 2000.

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

[18] OMG. Unified modelling language: Superstructure(version 2.0). http://www.omg.org, July 2005.

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

[20] N. Sangal, E. Jordan, V. Sinha, and D. 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.

[21] The apache ant project. URL: http://ant.apache.org/.

[22] R. J. Walker, G. C. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, and J. 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.