Identifying Architectural Change Patterns in 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

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 paper, we propose a new approach to modeling 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 system changes at the architecture 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.

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. Maintenance activities become more difficult, time consuming, and risky.

Explicitly modeling 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 [1, 2, 7, 16]. 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 modeling 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++, the 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 classifiers, and a package diagram is unable to capture inheritance and usage dependency between classes at a coarse-grained level. In our previous work, we have proposed a hybrid model to represent object-oriented systems at coarse-grained levels [3, 4]. In this paper, we apply this modeling approach 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.

The remainder of this paper is organized as follows. Section 2 introduces the notation and the construction of hybrid models. Section 3 outlines our evolution analysis. Section 4 presents the case study. Section 5 discusses current researches related to our work, and section 6 summarizes what we have done and discusses planned future work.

2 The Hybrid Model

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 explicitly describes the boundary of the objects and emphasize their usage dependencies. As shown in Figure 1, a hybrid model is composed of a collection of aggregate components and the connectors between them. We now describe how we customize the Hybrid Model for evolutionary analysis; details of the basic Hybrid Model can be found elsewhere [3, 4].

2.1 Aggregate Component

An aggregate component is created from a package. It models not only the package, but also the collection of objects that can be instantiated from the concrete classes of the package. As Figure 1 depicts, an aggregate component contains three types of classes: defined, exiled, and ghost classes. The defined and exiled classes of a component describes the scope of the package 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:

- *Defined* classes are implemented in the package.
- *Exiled* classes are declared but not implemented in the package.
- *Ghost* classes are the duplicates of the classes that the defined classes inherits from other packages.

2.2 Ports

An aggregate component can be viewed as a logical computation unit that provides resources to and require resources from its environment. Ports are the interfaces through which an aggregate component interacts with its environment. We distinguish two kinds of ports: inports and outports [13].

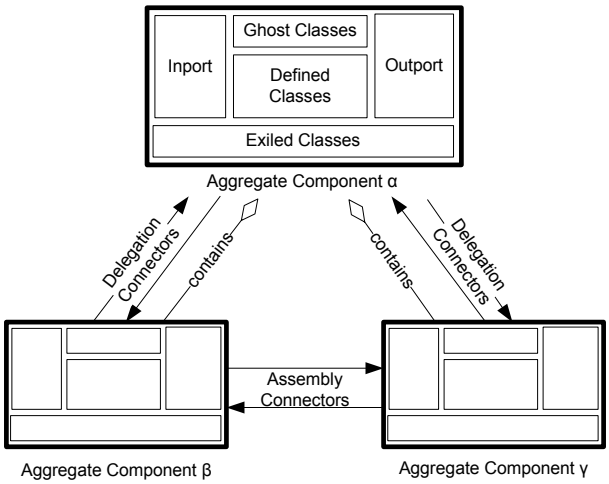

Figure 1. An Example Hybrid Model

- An *inport* lists the set of resources that the component provides and are actually used by other components.
- An *outport* describes the resources that the component requires from others.

In this paper, we focus on **call** dependencies, and refer a resource as a pair: $resource \in C \times M$, where

- C is the set of types of the receiver objects,
- M is the set of message signatures, which consist of the action names, the types of arguments and the return types of messages.

Thus, the inport of component α, $inport(\alpha) \subseteq C \times M$, represents the incoming messages that the component may receive from others, while its outport, $outport(\alpha) \subseteq C \times M$, represents the outgoing messages that the component may send to others.

2.3 Connectors

A connector [12, 13] specifies the interrelationship among two components. There are two types of connectors among aggregate components: delegation and assembly connectors.

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

$$dc_{inport}(\alpha, \beta) = inport(\alpha) \cap inport(\beta)$$
$$dc_{outport}(\alpha, \beta) = outport(\alpha) \cap outport(\beta)$$

- An *assembly* connector specifies the client-server relationship between two components. 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.

$$ac(\mu, \nu) = outport(\mu) \cap inport(\nu)$$

3 Evolution Analysis

In this section, we show how comparing the hybrid models of successive versions of the system can aid in better understanding the evolution of the system's architecture. Currently, we recover the additions and removals of entities in hybrid models by comparing their names. We consider that two classes/packages/components are same if they have the same fully-qualified name, and that two resources are the same if they have the same receiver type and message signature.

3.1 Change in Aggregate Components

Our objective in 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:

- *Defined classes:* The number of defined classes that have been added or removed serve as a measure of the growth of the containing package.
- *Ghost classes:* If the set of ghost classes of a component has changed, then cross-package inheritance relations have also changed.
- *Exiled classes:* 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.
- *Provided resources:* 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 [2].
- *Required resources:* Adding or removing a resource to/from the outport indicates that the aggregate requires different services internally, 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" [11], and the requests are the contract that 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 components, 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.

3.2 Change in Assembly Connectors

An assembly connector specifies a client-server relationship between the components at the same level of granularity. It is associated with a collection of resources that is provided by the server component and used by the client component. 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 is characterized by the following equation:

$$\begin{aligned} \Delta_{ac}(\alpha, \mu) &= \Delta_{outport}(\alpha) \cap \Delta_{inport}(\mu) && (1) \\ &\cup \Delta_{outport}(\alpha) \cap \Xi_{inport}(\mu) && (2) \\ &\cup \Xi_{outport}(\alpha) \cap \Delta_{inport}(\mu) && (3) \end{aligned}$$

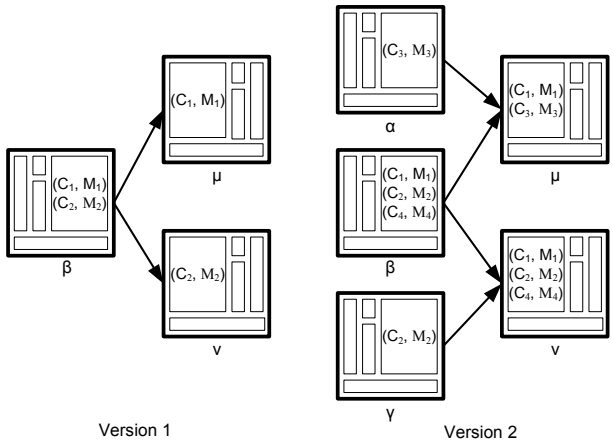

Figure 2. Change in Assembly Connectors.

(1) Co-change between the client and server component. The involved 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 2, component α requires a new resource, (C_3, M_3), and component μ provide 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.

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

$r_{afferent}(\alpha, \mu)$ 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 2, $r_{afferent}(\alpha, \mu) = 1$ and $r_{afferent}(\beta, \mu) = 0$, thus new responsibility is added to component μ solely for the needs of component α.

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

$r_{efferent}(\alpha, \mu)$ 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 2, $r_{efferent}(\beta, \mu) = 0$ and $r_{efferent}(\beta, \nu) = 1$, thus only component ν contributes to the change of component β.

(2) Reuse resources. 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.

(3) Re-implement resources. 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.

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.

3.3 Change in Delegation Connectors

A delegation connector promotes the ports of the subcomponents to their container component. It is associated with a collection of resources shared by the ports of the container and containee component. 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. Due to the length limitation of the paper, 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.

$$\begin{aligned} \Delta_{dc-inpot}(\alpha, \mu) &= \Delta_{inport}(\alpha) \cap \Delta_{inport}(\mu) \quad (4) \\ &\cup \Delta_{inport}(\alpha) \cap \Xi_{inport}(\mu) \quad (5) \\ &\cup \Xi_{inport}(\alpha) \cap \Delta_{inport}(\mu) \quad (6) \end{aligned}$$

(4) Internal change leads to external change. 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 3, 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

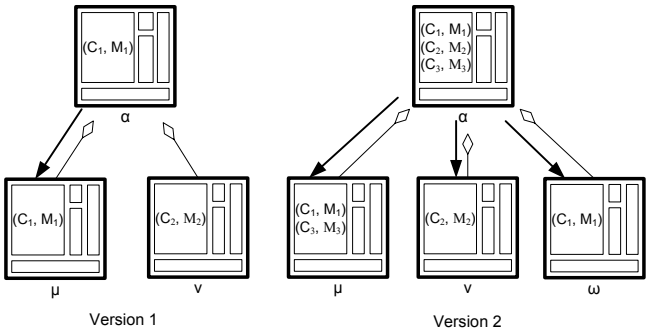

Figure 3. Change in Delegation Connectors.

changes with the change size of the ports of both container and containee components.

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

$r_{coarse-in}(\alpha, \mu)$ 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?

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

$r_{fine-in}(\alpha, \mu)$ 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.

(5) Exposing or hiding internal resources. The container component exports or hides the resource that is significant at the finer-grained level. For example, in Figure 3, 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) Reusing or re-implementing external resources. 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 3, new component ω in version 2 provides resource (C_1, M_1), which was already exported by component α in version 1.

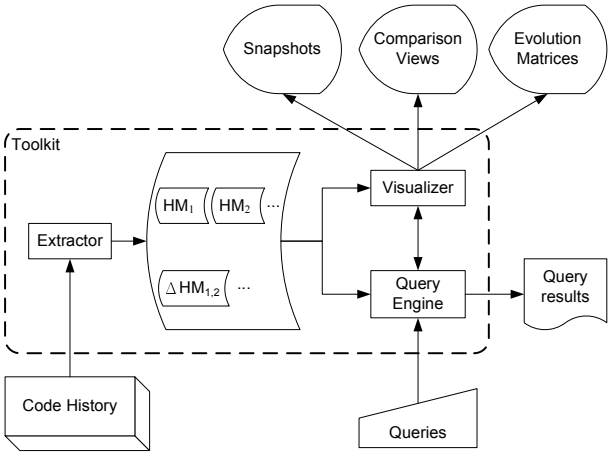

Figure 4. The Architecture of HEAT System.

4 Tool Support

We have developed the High-level Evolution Analysis Toolkit (HEAT) to implement and evaluate our approach. Our goal is to automate the extraction and comparison of hybrid models, and use visualization techniques to help maintainers to intuitively grasp the change at coarse grained level.

As shown in Figure 4, the toolkit is composed of three parts: extractor, query engine, and visualizer. The extractor automatically recovers hybrid models from code history and identifies the difference among extracted hybrid models. The query engineer provides an interactive environment, in which users can ask questions about the properties of hybrid models, and choose the preferred scope and granularity of subsequent visualization. The visualizer, which uses Graphviz to perform the layout and rending [9], can produce three automated evolutionary views of a a target system:

1. A *snapshot view* presents the hybrid model reverse engineered from an object-oriented system at a particular point in its history.
2. A *comparison view* presents the difference between two selected hybrid models. For example, Figure 6 shows the difference between Ant version 1.4 and 1.5.
3. A *evolution matrix* organizes snapshots of the selected aggregate components in a matrix. Figure 5 shows the history of sub-packages of package *ant*.

We apply polymetrics visualization technique [10] to highlight the key properties of hybrid models. For example, 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.

5 Case Study: The Evolution of Apache Ant

In this section, we present an exploratory case study in which we apply hybrid models to investigate how an object-oriented software system evolves over time. Our choice of case study was Apache Ant [14], a Java-based build tool. We analyzed 7 major releases of the system. In the first studied release, there were 6 packages and 116 classes (20 KLOC), while in the latest release, there were 71 packages and more than a thousand classes (234 KLOC).

5.1 How has the package *ant* evolved?

As Figure 5 shows, package *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, based on applying the methodology for studying architectural change that we have introduced above. We pay particular attention to possible indicators of architectural drift.

5.1.1 Increasing number of ghost classes

The number of ghost classes at this level of granularity has steadily increased. There were only two ghost classes in version 1.1, and both belonged to package *taskdefs*. In version 1.7, 63 classes have become ghost classes of one or more of the 7 packages at this level.

Several packages have acquired ghost classes since their first appearance in the system. For example, package *listener* inherited class `BuildListener`. The class and package name suggest that the package was used to group some classes that implement a common abstract concept. It is not surprising that these packages, which were initially designed to be at a low data abstraction level, continuously had 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 too 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 [5], 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`.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.

5.1.2 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 [6].

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.** has 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* has an exiled 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.

5.1.3 Expansion of Ports

One noticeable architectural change in Figure 5 is that the ports of most packages became wider and wider. Some

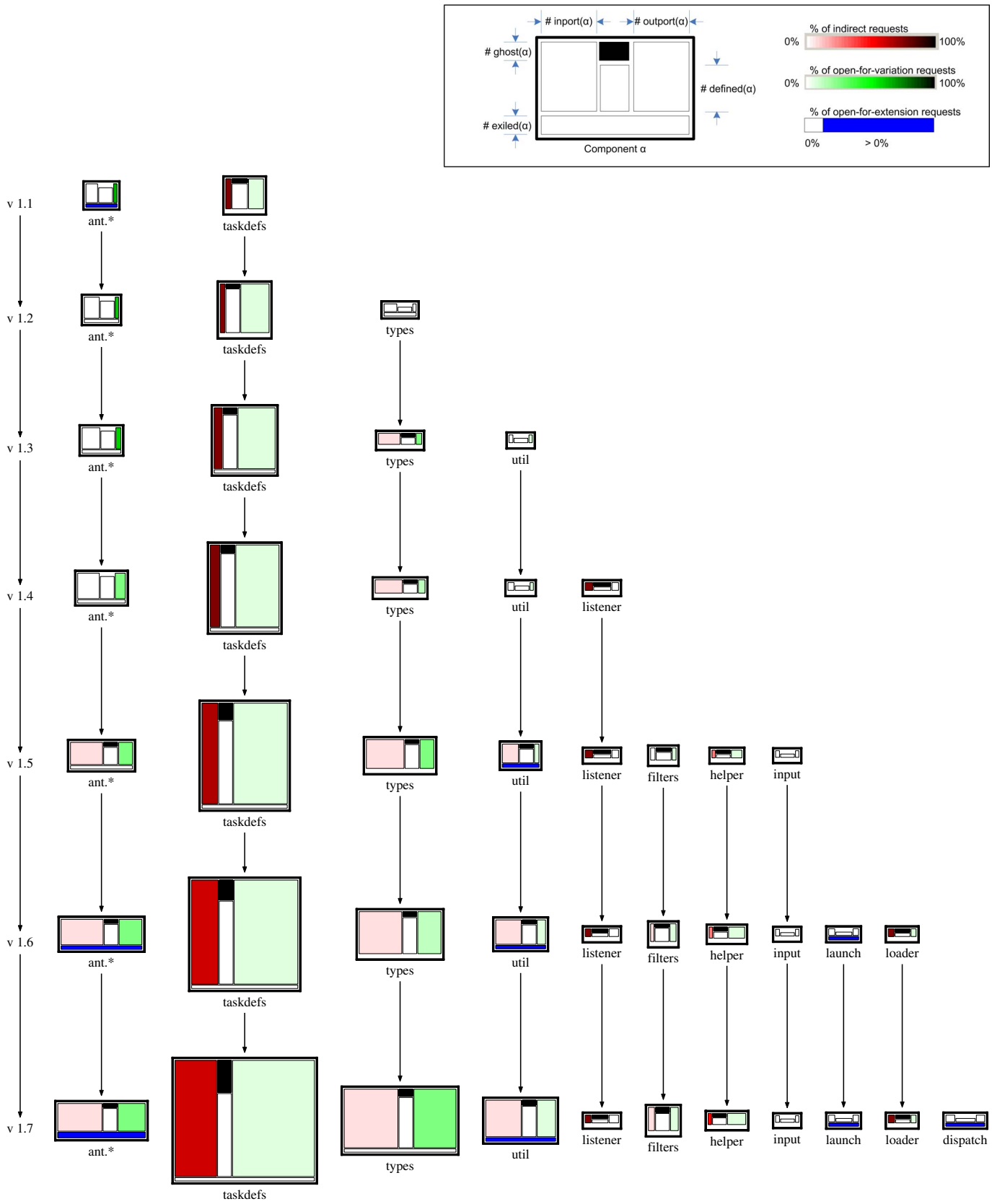

Figure 5. The history of packages package *tools.ant* is displayed in an Evolution Matrix.

changed not only in size, but also in color.

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 responds 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, it also started to receive direct requests from others. This indicates that package *taskdefs* increasingly acquired responsibilities that were not strongly related to the abstract concepts it implements.

5.1.4 Change of Assembly Connectors

We also created comparison views to study the difference between 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 *ant*, it grew much faster than any of its siblings, and it continually demanded changed resources from its siblings. In Figure 6(a), package *taskdefs* has dark outgoing arrows to five 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 order to meet its new requirements.

Second, package *ant.** provides some services that are used and reused by its siblings. Figure 6(b) 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(c) 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.

5.2 Evolution at the Finer-grained Level

As Ant has evolved, 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(d) 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(e) 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 subpackage 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(f) demonstrates, all three subpackages 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.**.

5.3 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 hybrid models for each major releases of Ant system, as we found that there is little package-level change in minor releases. Other systems may not share the same properties.

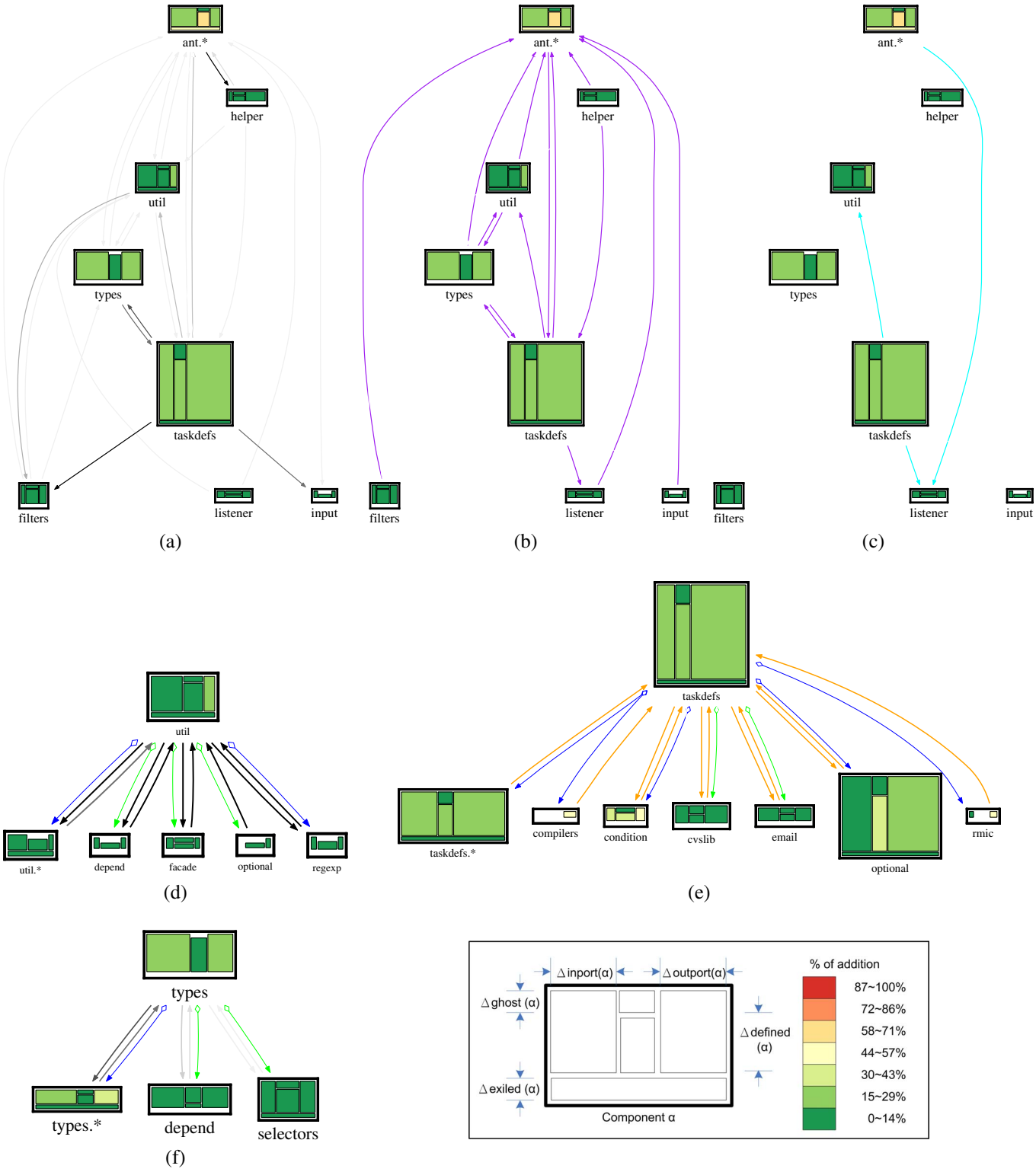

Figure 6. Comparison between Ant 1.4 and Ant 1.5. (a) assembly connector change in package *ant* caused by co-change between components. The color shades of connectors are determined by their $r_{afferent}$ values. (b) assembly connector change in package *ant* for the reuse purpose. (c) assembly connector change in package *ant* for the reimplementation purpose. (d) delegation connector change in package *util* caused by the co-change between components. The color shades are determined by $r_{fine-in}$ and $r_{fine-out}$ values. (e) delegation connector change in package *taskdefs* for the reuse and reimplementation purpose. (f) delegation connector change in package *types* caused by the co-change between components. The color shades are determined by $r_{coarse-in}$ and $r_{coarse-out}$ values.

6 Related Work

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

Change Pattern Detection. Godfrey and Zou employ origin analysis to detect structural change in procedural code [8]. They emphasize the analysis on call relationships, and classify the detected change into renaming, moving, splitting, and merging methods.

Xing and Stroulia presented a technique to recover co-evolution patterns among classes of an evolving software system [16]. 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 approaches have been developed to detect possible refactoring activities [1, 7]. However, they can only recover the change patterns that they intend to identify.

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 [10]. 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 [15]. They developed special coloring techniques to represent one particular property, e.g., fanin and fanout, of a target system at a selected level of granularity.

7 Conclusion and Future Work

In this paper, 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 showed 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 prefered level of granularity.

Future work includes performing additional case studies. We plan to study the evolution of several applications from the same problem domain, e.g., UML modeling tools. We hope to find similar change patterns or general trends in evolution.

References

[1] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *Proc. of the 15th ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA-00)*, pages 166–177, Minneapolis, MN, USA, Oct. 2000. ACM.

[2] D. Dig and R. Johnson. The role of refactorings in api evolution. In *Proc. of the 21st IEEE Intl. Conf. on Software Maintenance (ICSM-05)*, pages 389–398, Budapest, Hungary, Sept. 2005. IEEE.

[3] X. Dong and M. W. Godfrey. A hybrid program model for object-oriented reverse engineering. In *Proc. of the 15th IEEE Intl. Conf. on Program Comprehension (ICPC-07)*, pages 81–90, Banff, AB, Canada, June 2007. IEEE.

[4] X. Dong and M. W. Godfrey. System-level usage dependency analysis of object-oriented systems. In *Proc. of the Intl. Conference on Software Maintenance (ICSM-07)*, pages 375–384, Paris, France, Sept. 2007. IEEE.

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

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

[7] C. Gőrg and P. Weißgerber. Detecting and visualizing refactorings from software archives. In *Proc. of the 13th Intl. Wksp. on Program Comprehension (IWPC-05)*, pages 205–214, St. Louis, MO, USA, May 2005. IEEE.

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

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

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

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

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

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

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

[15] J. Wu, R. C. Holt, and A. E. Hassan. Exploring software evolution using spectrographs. In *Proc. of the 11th Working Conf. on Reverse Engineering (WCRE-04)*, pages 80–89, Delft, Netherlands, Nov. 2004. IEEE.

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