A Hybrid Program Model for Object-Oriented Reverse Engineering

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

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 paper, we propose a hybrid model of object-oriented software that blends the use of classes and entities at different levels of granularity. Each coarse-grained entity represents a set of software objects, and contains the complete static description of the objects it represents. This hybrid model allows maintainers to understand objects as independent units, and focus on the 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.

1 Introduction

An object-oriented software system is composed of a collection of communicating objects that cooperate with one another to achieve some desired goals. Those software objects are often used to model real-world objects. Similar objects form classes, which provides the static description of how objects behave. Aiming to supporting object-oriented program comprehension, reverse engineering seeks to create representations of object-oriented systems about classes/objects and their interrelations.

The biggest challenge for a reverse engineering tool is to capture a large amount of information using descriptive and understandable representations, while at the same time not overwhelming the users with too much detail. Large-scale object-oriented systems typically consist of hundreds of classes as well as a high degree of interdependence among them. However, humans have limited information storing and manipulating abilities [8]. If the provided representation is too complex, maintainers may "drown" in the information overload. Moreover, too much information on one diagram may decrease tool performance significantly [5].

A commonly used strategy to address this challenge is to synthesize representations at a coarse level of granularity. Many existing tools offer to generate package diagrams by dividing classes into packages, which act as coarse-grained proxies for their contained classes [10, 12, 13, 15]. While grouping classes into packages provides better readability of classes and their interrelations, it harms the comprehensibility of objects as independent units. Because the static description of an object can end up being distributed across multiple packages due to inheritance, it can be difficult to capture the external properties of the software object at a coarse-grained level, not to mention to identify the real-world object it models.

In this paper, we propose a hybrid program model that blends the use of model elements 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.

The rest of paper is organized as follows. Section 2 analyzes the reasons why package diagrams are ill suited to object-oriented program comprehension. In section 3, we present a new model for creating representations of object-oriented systems. Section 4 demonstrates a real world comprehension scenario using the new model representation, and section 5 presents our conclusion.

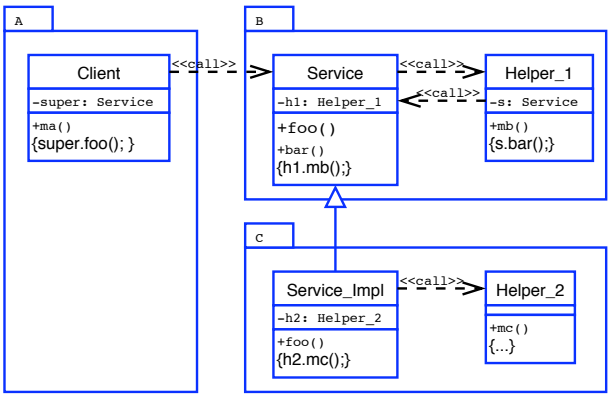

Figure 1. The Class Diagram of an example program. Method invocations are modelled as stereotyped dependencies.

2 A Motivational Example

In an object-oriented system, classes are arranged in an inheritance hierarchy. The term *class* is used to connote two slightly different ideas, one prescriptive and one descriptive. First, a class is a blueprint of objects. It is a programming language construct that encapsulates a set of attributes and the set of operations performed on the attributes. A class may also inherit attributes and operations from its superclasses. Second, a class represents the entire collection of objects that can be created from the blueprint. An object of a class is also polymorphically an object of the ancestors of the class. A class, along with its ancestors, describes a common structure and a common set of behaviors that are shared by the objects the class represents. During the generation of a package diagram, a class is often separated from its ancestors. Hence, a package, which is composed of a set of classes, may not contain the complete blueprints (i.e., semantic descriptions) of the objects that its contained classes represent.

For example, consider the program whose class diagram is shown in Figure 1. 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 five classes into three distinct packages. Using the package diagram, 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, neither package B nor package C can be understood solely based on the code it contains. Class Service in package B declares an abstract method foo, whose behavior is defined in class Service_Impl. When studying package B, maintainers can only guess the behavior of the method based on its signature. Service_Impl inherits method bar from its superclass. The code of the method is a part of the blueprint of the objects it represents. Therefore, it is necessary for maintainers to investigate the internal details of package B to understand how an object of class Service_Impl behaves. If maintainers limit their investigation in either 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, method bar is part of the external properties of package B, not package C, although an object of Service_Impl does provide such a method implicitly.

Third, the interrelationship between packages may not capture the interrelationship between objects contained in the packages. Class interrelationships, such as *calls* and *composition*, 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 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 description of how an object of Service_Impl interacts with other objects. However, none of the implicit *calls* dependencies that Service_Impl is involved in contribute to the interdependencies between package C and other packages.

Due to the above issues, package diagrams may not satisfy the comprehension needs at a high level. Pennington *et al.* reported that programmers develop a situation model at a high level of granularity [11]. Burkhardt *et al.* applied Pennington's approach to object-oriented program comprehension [2]. They believe that the situation model of an object-oriented system consists of both structural and behavioral aspects. The structural aspect is of domain objects, describing their interrelationship and the program goals. The behavioral aspect captures the client-server relationship among domain objects. However, in the package diagrams, the blueprint of a software object is distributed across several packages, none of which captures the complete external properties of the object, and the client-server relationship it involves. In this example, the static description of an object of Service_Impl is distributed in both package B and package C. It is unclear how either package can be mapped directly to the domain objects that Service or Service_Impl

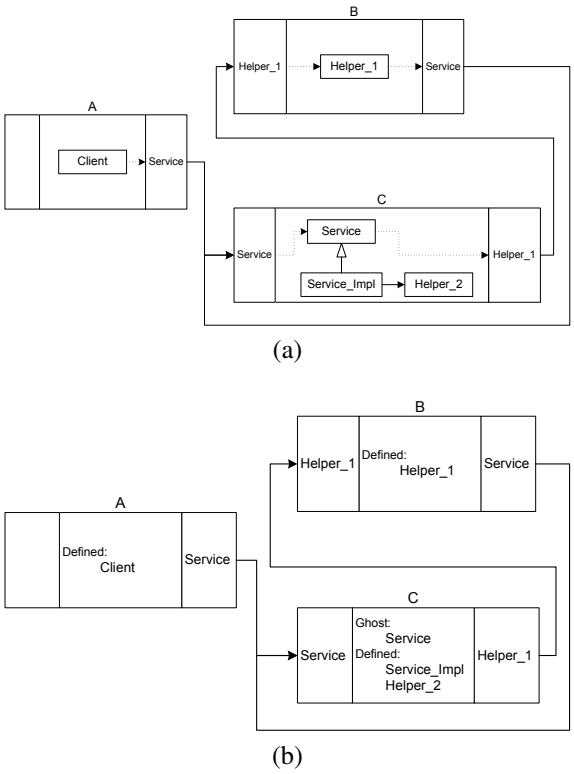

Figure 2. The hybrid model for the example program at the top level. (a) shows the expanded view, and (b) shows the collapsed view.

models, and how the dependencies between packages can be interpreted as interrelationships among domain objects. It is even more difficult for maintainers to use the package diagram to derive the design intentions, such as how the system is decomposed, how the functionality of the system is distributed among the classes, and what is the responsibility of each class. Therefore, package diagrams are of limited usefulness to developers seeking to acquire a high-level understanding of an object-oriented system.

3 The Hybrid Model

The major limitation of a package diagram is due to the fact 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 program language constructs. The interrelationships among packages represent the compilation dependencies among program language constructs, but cannot be interpreted as the relations among objects or classes.

To address the issues mentioned above, we propose a new hybrid model of object-oriented systems that blends the use of classes and entities at different levels of granularity. In our model, each coarse-grained entity includes the complete static descriptions of the objects that are defined and implemented within it. Thus, a coarse-grained entity, like a class, can be interpreted as the set of objects it defines. An interdependency between two coarse-grained entities arises when there is a possible relation between the objects they represent.

3.1 Constructing Hybrid Models

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

3. Any 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 hybrid model of the example program at the top-level abstraction.

In the next section, we present the notation of our hybrid model in order to facilitate further discussion in the following sections.

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

3.2 Notation

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

3.2.1 Resource

We define a resource [3] 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 only show type resources, and consider variable resources and method resources as parts of type resources.

3.2.2 Component

A component [7, 9] 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.

An aggregate component may contain two types of classes:

- *Defined* classes, which are declared and defined in the component. For example, Helper_1 is a defined class of component B.
- *Ghost* classes, which were originally declared elsewhere but implemented in the component. For example, Service is a ghost class of component C.

An aggregate component also serves as a proxy for all of the objects that are objects of the defined classes that it (recursively) contains. Ghost classes of the component provide necessary context information for reasoning about the properties of those objects.

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 [9].

- 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. 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. For example, in Figure 2, component A requires an external resource Service.

3.2.4 Connectors

A connector [7, 9] specifies the interrelationship among two components. There are three types of connectors: inheritance, delegation [9], and assembly [9] connectors.

- An *inheritance* connector, visually represented as a solid line with an empty arrowhead, specifies the inheritance relation between two classes. In a hybrid model, inheritance can exist within only aggregate components, and cannot cross component boundaries.
- 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 B.

3.3 Interpretation

The main goal of hybrid models is to help maintainers apply a divide-and-conquer comprehension strategy to deal with the complexity of large systems. At a high level of abstraction, maintainers can focus on the external properties of the coarse-grained entities and their interrelationships without worrying about their internal implementation. At a low level of abstraction, maintainers can focus on one aggregate-component to acquire deeper knowledge.

With a hybrid model, maintainers are able to study one component at a time. Each component represents a set of objects, and contains a 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 2, only component C contains class Service. Therefore, to understand class Service, maintainers can limit their investigation within that component.

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 2 contains three resources. Among them, only Service is known to other components. Service_Impl provides the implementation for the resource, but it is at a low data abstraction level and is 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 resource Service, while its internal details, Service_Impl and Help_2, 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 (dependencies explicitly defined in source code) and implicit dependencies (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 2, component A may send message to component C, but there is no communication path from component A to component B.

In addition, hybrid models can be used to 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, maintainers may focus on *calls* and *instantiates*. Different relations generate different hybrid models, but share the containment relation among components. We view those hybrid models 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 2 shows the hybrid model with *calls* add-on map. As the base map remains unchange, it severs a platform to compare or combine different relations.

3.4 Tool Support

We implemented a prototype tool to support the construction and visualization of hybrid models. Our tool is composed of three parts: extractor, analyzer and visualizer. The extractor collects static information from Java class files. The analyzer provides an interactive environment, in which users can modify containment hierarchy of an object-oriented system, choose from primitive class interrelations (e.g. *calls*, *instantiates*, *aggregation*, etc.) or their combinations (union, intersection, and difference), and perform queries on the hybrid models, which are automatically generated. The visualizer presents hybrid models using Graphviz [4].

4 Case Studies

In this section, we present an exploratory case study to show how the 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 part of Swagkit [14], 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 [6].

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 try 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 [16, 17], to create the system model. Instead, we will create it manually to show how the hybrid modelling approach can help maintainers to perform several tasks: to chunk fine-grained entities to a higher level of abstraction structure, to form meaningful hypotheses at the high-level abstraction, to confirm or reject those hypotheses using low-level information, and to derive design rationales during the program comprehension process.

4.1 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 know that classes may connect with each other through a variety of relations, such as *inheritance*, *association*, *composition*, *instantiates*, etc. With hybrid models, 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

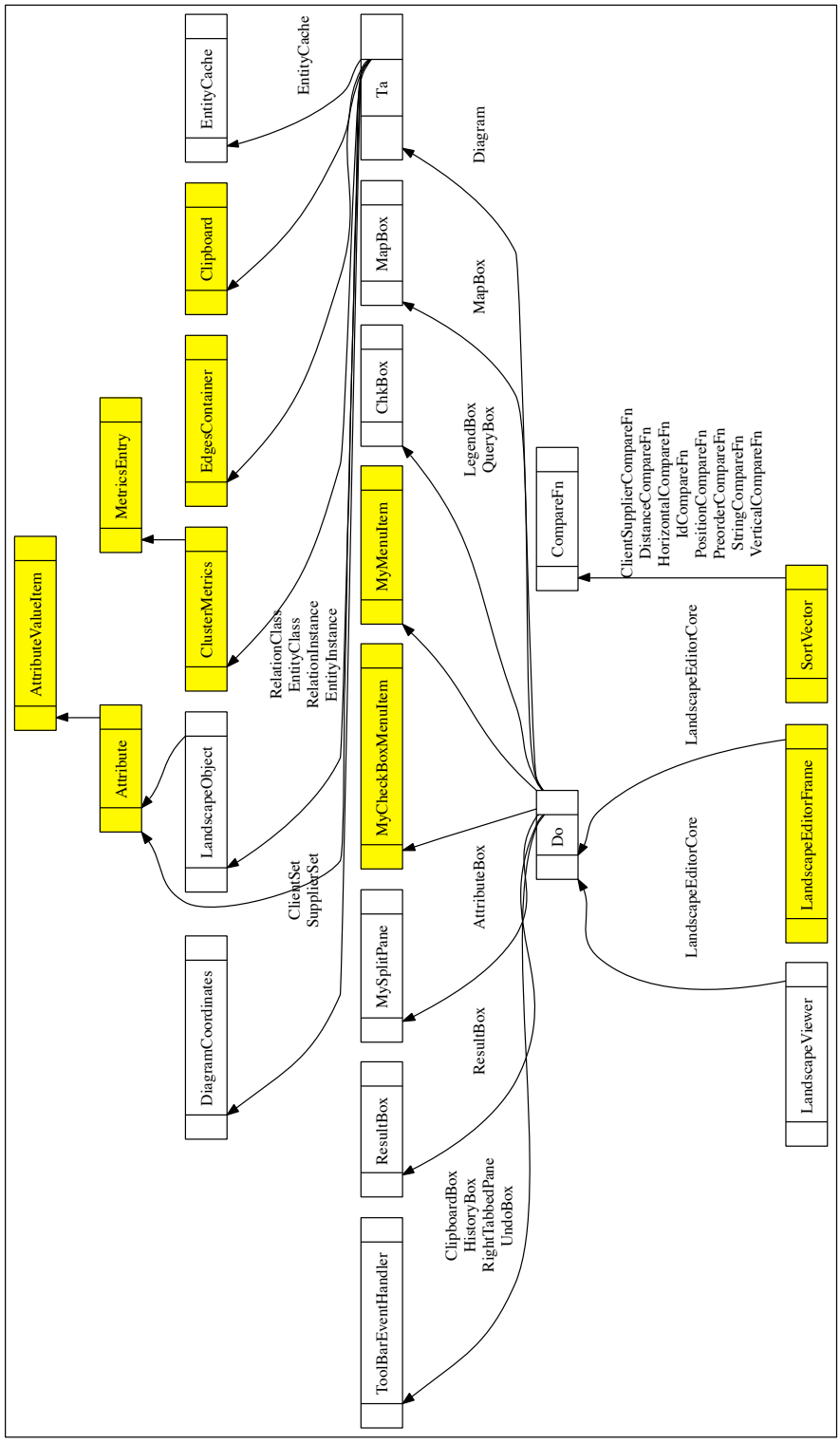

Figure 3. A partial hybrid model with *composition* add-on map. Yellow boxes represent class-components, and are labelled with class names. White boxes represent aggregate components, and are labelled with user-defined names. Connectors to container-components are labelled with associated resources.

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 coarse-grained node, and named it after the root of the tree. For example, ToolBarButton is grouped with its 18 subclasses to form an aggregate component called 'ToolBarButton'. This component can be cross-referred to a general toolbar button. For those classes with multiple parents, we subjectively assign them with one of their ancestors according to the naming convention. For example, EntityInstance is grouped with LandscapeObject instead of DiagramCoordinates, as its name are 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 classes, and map the resulting components to the domain objects that the outer classes model.

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 them are types of graphic components, and 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 user interface. After the above two steps, we reduce 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 3 shows the most complicated part of the hybrid model with *composition* add-on map.

From Figure 3, we are able to identify three groups. The left bottom shows that LandscapeEditorFrame is composed of LandscapeEditorCore, whose family members (ancestors, descendants, and inner classes) are composed MyMenuItem, AtttibuteBox, LegendBox, MapBox, etc. The class names of those involved classes remind us 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 it 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 this case, we adjust the clustering, and check other relations iteratively.

4.2 Constructing Hypotheses

When we acquired some knowledge of the code, especially after studying Figure 3, we consider the system within its problem domain, and form 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 [1]. 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

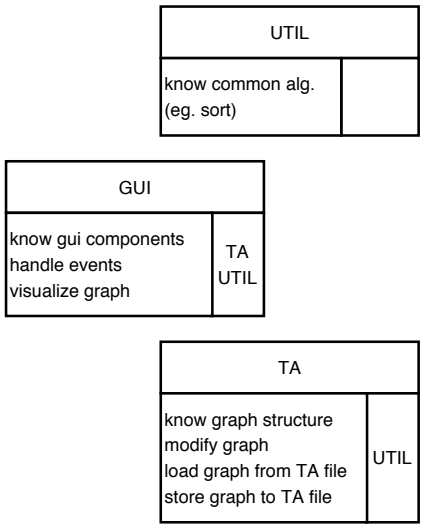

Figure 4. Conceptual Model of LSEdit

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 depicts our conceptual model in CRC notation. The LSEdit system can be decomposed into three main parts: the graph user interface, the typed graph specified in a TA file, and some utility classes.

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

4.3 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 GUI component, 96 classes for TA component, and 29 classes for UTIL component. Then we created the *calls* add-on map as shown in Figure 5.

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 a 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 through a particular dependency. Therefore, Figure 5 give us a peek into the responsibilities of the top-level components.

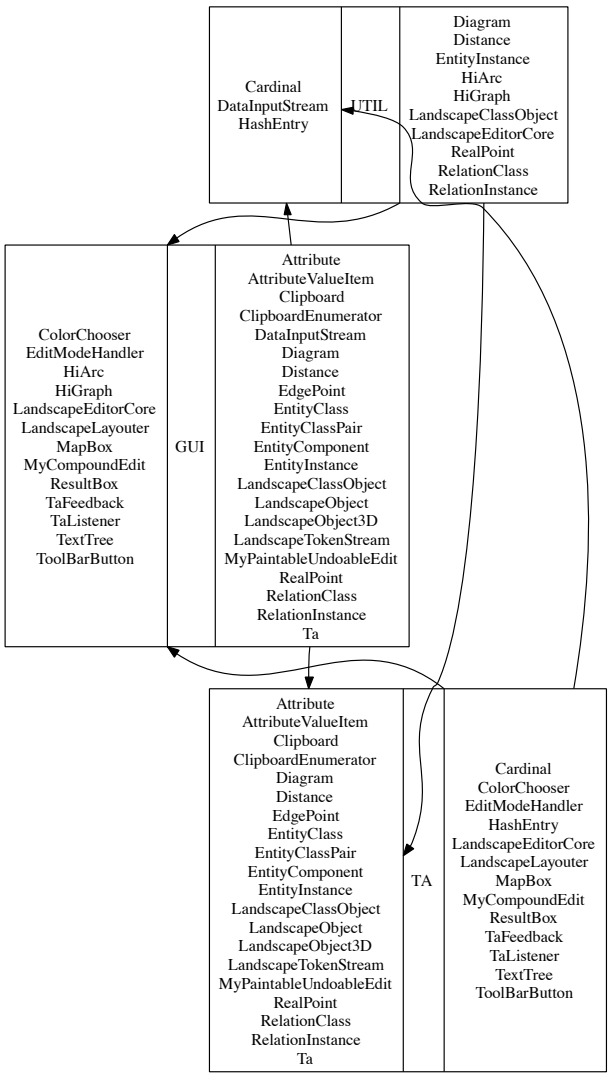

Figure 5. Concrete Model of LSEdit (Hybrid Model with *calls* Add-on Map)

Comparing the conceptual and concrete model of LSEdit, we found that the inports of the top-level components roughly match their responsibilities in conceptual model. The inport of TA includes most classes that correspond to constructs of a TA file, such as Ta, Attribute, EntityClass, etc. This indicates that TA has the responsibilities to maintain TA files. From the inport of GUI, we

are able to identify some GUI components, such as ResultBox and ToolBarButton, the class that deals with user input event, such as EditModelHandler, and the class related to graph rendering, such as LandscapeLayouter. The inport of UTIL includes 3 classes that encapsulate common algorithms. After checking the *static-method-invocation* add-on map, we found that most classes in UTIL contain only static methods and are not meant to be instantiated.

The assembly connectors in Figure 5 describe the interaction among components, and can be interpreted as the collaboration among domain objects. All the collaboration in the conceptual model can be identified from the concrete model. However, there are some unexpected collaborations. For example, UTIL requires resources from both GUI and TA. We examined the internal structure and behavior of 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 TA requires many resources from GUI, especially some graphic components. This can be caused by two reasons, either the partition is not reasonable, or some responsibilities and collaborations are missing from the conceptual model. We studies the TA region. 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 collaborator of TA in our conceptual model should be synchronized with the concrete model.

4.4 Derive Design Rationale

The 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 responsibilities 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. The GUI part is further decomposed into small graphic components according to the structure of user interface arrangement. The TA part 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 needed to interact with it. With hybrid models, it is possible for maintainers to derive 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 the *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 rationales about program logic encapsulation. For instance, GUI component in Figure 5 reveals 13 out of 223 classes it contains. This indicates that most responsibilities this component has are not pertinent to the use of the component and they 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 consider of isolating GUI related responsibilities from TA and moving them to the GUI.

The 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 that 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 Case Study Summary

The hybrid models help program comprehension at coarse-grained level. 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 inports can be interpreted as the functionality of the subsystems. When it is cross referenced to a complicated domain object, the resources attached to its inports indicate the responsibilities expected from other domain objects. Therefore, maintainers can think in the problem domain using the appropriated hybrid models.

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 its own 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.

5 Conclusion

In this paper, we have presented a hybrid model of object-oriented systems. Instead of grouping programming language classes, we aggregate the complete static description of software objects, so that each aggregate component of a hybrid model represents a set of objects. The assembly connectors among components capture all possible relations among the objects in different components.

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 the hybrid models to program comprehension in a realistic comprehension scenario.

References

[1] K. Beck and W. Cunningham. A laboratory for teaching object oriented thinking. In *Proceedings on Object-oriented programming systems, languages and applications*, pages 1–6, New York, NY, USA, 1989. ACM Press.

[2] J.-M. Burkhardt, F. Détienne, and S. Wiedenbeck. Mental representations constructed by experts and novices in object-oriented program comprehension. In *Proceedings of the IFIP TC13 Interantional Conference on Human-Computer Interaction*, pages 339–346, London, UK, UK, 1997. Chapman & Hall, Ltd.

[3] 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.

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

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

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

[7] 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.

[8] G. 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.

[9] OMG. Unified modelling language: Superstructure(version 2.0). http://www.omg.org, 7 2005.

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

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

[12] C. Riva and J. V. Rodriguez. Combining static and dynamic views for architecture reconstruction. In *Proceedings of the 6th European Conference on Software Maintenance and Reengineering*, pages 47–56, Washington, DC, USA, 2002. IEEE.

[13] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. In *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.

[14] Swagkit. URL: http://www.swag.uwaterloo.ca/.

[15] P. Tonella and A. Potrich. *Reverse Engineering of Object Oriented Code*. Springer, 2005.

[16] V. Tzerpos and R. C. Holt. Acdc: An algorithm for comprehension-driven clustering. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, pages 258–267, Washington, DC, USA, 2000. IEEE.

[17] T. A. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *Proceedings of the Fourth Working Conference on Reverse Engineering*, pages 33–43, Washington, DC, USA, 1997. IEEE.