Understanding Source Package Organization using the Hybrid Model

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

Within a large, object-oriented software system it is common to partition the classes into a set of packages, which implicitly serve as a set of coarsely-grained logical design units. However, as such a system evolves and design drift sets in, it becomes increasingly challenging for developers — especially those who are new to the project — to comprehend the underlying criteria behind the package-level design of the system. This problem is exacerbated by the fact that in most object-oriented programming languages the package (or namespace) construct has little semantics beyond that of a simple container, and so fails to capture the essential properties of the objects that its contained classes represent. In this paper, we propose an approach to uncovering package partitioning criteria by analyzing the collaboration patterns between packages. Our analysis approach is based on the Hybrid Model, a program model that describes the coarsely-grained structure and global behaviour of an object-oriented system. We present an exploratory case study to show how our approach can help maintainers to derive the design criteria related to coupling, cohesion, function reuse, and inheritance reuse.

1 Introduction

The design of a large object-oriented system is typically organized around principles and criteria that are more coarsely grained than those of objects and classes. Programmers often use packages to organize classes along several criteria, such as the similarity of the design and the collaboration between classes to implement a higher-level abstraction. Thus, understanding how packages are organized is key to capturing the high-level design of an object-oriented system.

Most existing reverse engineering tools focus on analyzing or visualizing the structural dependencies between packages. Typically, they extract package diagrams from existing source code by "lifting" the interrelationships between classes to become package-level dependencies. While a package diagram captures the structural dependencies between classes at a coarsely grained level, it loses the semantics of those class-to-class relations. That is, the dependency between two packages cannot be interpreted as the possible relationships between run-time objects, because usage dependencies — such as *instantiates*, *references*, and *calls* — must be interpreted in the context of one or more class hierarchies. As a result, using structural dependencies alone makes it hard to answer semantically rich questions, such as "*How is the functionality implemented in package A used by package B?*", and "*What design principles guided the designers to divide the system's classes into this set of packages?*"

In this paper, we study package organization using the Hybrid Model, a program model that enriches a package diagram with semantic information including the important properties of objects and their interrelationships [1, 2]. Our goal is to understand how packages combine to form a large application by analyzing the collaboration among objects from different packages. We conducted a case study on JHotDraw [3, 4] to show how our approach can help maintainers to derive design criteria related to coupling, cohesion, function reuse, and inheritance reuse.

The rest of this paper is organized as follows. Section 2 briefly describes the Hybrid Model. Sections 3 elaborates our approach to analyze package organization. In Section 4, we apply the analysis approach on an exploratory case study, and Section 5 summarizes our contributions.

2 The Hybrid Model

A package diagram contains little semantic information because packages, unlike classes, cannot represent runtime objects, and package dependencies fail to connote the communication between objects. To address this, we created the Hybrid Model to capture the essential properties that relate to objects and their communication. We have described the

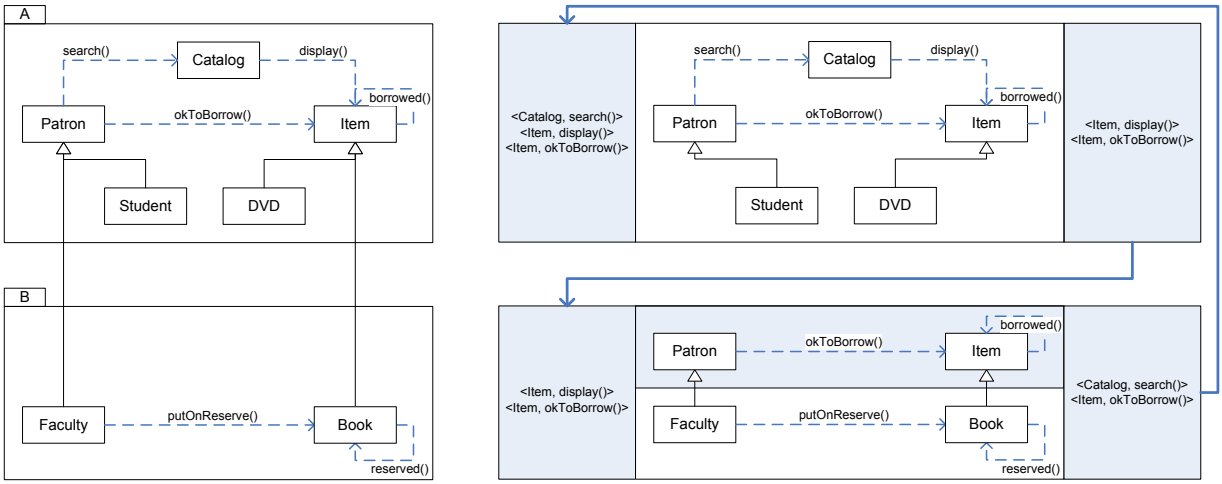

Figure 1. (a) The reverse engineered Class Diagram of an example Library Management System. (b) The Hybrid Model derived from (a).

Hybrid Model previously [2], so we provide only a brief summary of it here.

The basic idea in the Hybrid Model is to let a package represent not only the classes it contains, but also the objects that can be instantiated from the group of classes. To create a Hybrid Model, we first remove all abstract classes from a package diagram, because they contain partial blueprints of the objects that their subclasses represent and should be understood along with their subclasses. Then, we pull into each package all ancestors of its remaining concrete classes. Finally, any dependencies between classes that are not in the same package become dependencies of their corresponding packages. As a result, a package in the Hybrid Model contains the complete static description for the objects that can be instantiated from its containing concrete classes.

For example, Figure 1(b) depicts the Hybrid Model derived from the class diagram of a simple library management system, shown in Figure 1(a). In the Hybrid Mode, each package has two ports, though which the package interacts with its environment.

- An *outport*, visually represented as the right boundary of a package, describes the resources that the package requires from others. A resource is a pair: $resource \in C \times M$, where C is the set of types of the receiver objects, and M is the set of message signatures, which consist of the action names, the types of arguments and the return types of messages. In Figure 1, package A requires two resources: $outport(A) = \{\langle Item, display() \rangle, \langle Item, okToBorrow() \rangle\}$
- An *inport*, visually represented as the left boundary of a package, is the interface through which the package provides resources to others. The inport represents the subset of the available resources that the package provides and are actually required by any packages through their outports. In Figure 1, package A provides two resources to package B: $\langle Catalog, search() \rangle$, and $\langle Item, okToBorrow() \rangle$. Most resources that are provided by a package but not used by others are not considered to be part of the inport list. However, $\langle Item, display() \rangle$ is an exception. It belongs to $inport(A)$ because it is provided by $inport(B)$ and used by $outport(A)$. By adding the resource to $inport(A)$, we make the information regarding the same resource complete on the boundaries of packages.

3 Analyzing Package Design

Packages are typically used to organize classes belonging to the same category or providing similar functionality. Thus, knowing how packages are related and what they share in common is of key importance in reasoning about the partitioning criteria. We examine package relations from two perspectives: the similarity in ports, and the collaboration among packages.

3.1 Similarity in Ports

Due to the usage of polymorphism, more than one type of objects may response to the same message. When two packages exhibit high similarity in their inports, they provide a common set of resources. This pattern often occurs when an abstract concept has many implementations, and

packages are used to separate implementations for different purposes. Examples of this pattern are found in package *org.opencms.db* from OpenCms version 7.0.4 [5]. It consists of 9 sub-packages with almost identical inports. Each sub-package implements the database connection and access functionality for a different database.

It is also possible that more than one package may use the same resource. If two packages exhibits high similarity in their outports, then they may have similar behaviours. Examples of this pattern are found in package *org.opencms.xml* from OpenCms version 7.0.4 [5]. It contains three sub-packages, which provide functionalities and utilities for managing basic XML documents, structure content, and unstructured content, respectively. Three packages require similar services from the rest of the system.

3.2 Collaboration Patterns

Depending on the numbers of providing and requiring packages, a resource may contribute to four different types of collaboration among packages: one-to-one, many-to-one, one-to-many, and many-to-many.

- A resource with one consumer and one provider specifies a one-to-one collaboration ($1 \rightarrow 1$) between two packages. If two packages are involved in only one-to-one collaborations, then they communicate with each other via an ad hoc interface. The interface is likely to have been designed specially for the provider package to meet the particular needs of the consumer package.

- A resource with a single provider but multiple consumers describes a many-to-one collaboration ($m \rightarrow 1$). A many-to-one collaboration typically indicates object or function reuse. If most of the resources that a package provides are used in many-to-one collaboration, then the package may contain a collection of abstractions, such as objects and functions, that were carefully designed so that they can be used by objects from different packages.

- A resource with one consumer and multiple providers represents a one-to-many collaboration ($1 \rightarrow n$). A one-to-many collaboration often indicates the use of polymorphism. The resource is a contract that the consumer and provider packages agree on, and is implemented in different packages. At runtime, any one of the provider packages may execute the contract.

- A resource with multiple consumers and multiple providers implies a many-to-many collaboration ($m \rightarrow n$). Many-to-many collaborations also involve polymorphism. The provider packages may contain multiple implementations for the same message, and each may respond to the message at runtime.

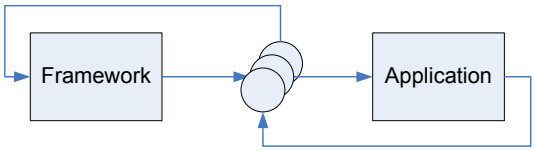

Figure 2. Framework-Application Pattern

Many-to-many collaborations are often more complicated than the other three kinds, because any change to the resource will propagate to all the packages involved. However, carefully designed many-to-many collaborations are also the key to design of the relationship between a framework and its applications. As shown in Figure 2, the framework package defines a collection of extension points, and provides implementations that contains basic functionalities. The application package is built on the framework. It reuses the basic implementations, and adds its own specialized implementation too.

4 Example Case Study - JHotDraw

We apply the Hybrid Model to investigate the package organization of JHotDraw [3, 4], a Java GUI framework for two-dimensional graphics. The versions analyzed are JHotDraw 6.0 beta 1 and JHotDraw 7.1. We compare the package organization of the two versions in order to find out whether the partitioning criteria changed as the architecture changed.

In both version, we focus on the top level packages. Table 1 and Table 2 list the numbers of resources related to the four collaboration patterns, and show how those resources are distributed in the ports of the packages. Comparing the package collaboration in version 6.0 and 7.1, we found the following differences:

- One-to-one collaboration significantly increased in JHotDraw 7.1, and it exhibits a clear layered structure. Based on Table 2, the packages in JHotDraw 7.1 can be divided into at least three layers. The bottom layer includes *geom*, *io*, and *util*, because all three packages have empty outports. The package *samples* forms the top layer because it is the consumer package in most of the one-to-one and many-to-one collaborations.

- Compare to version 6.0, JHotDraw 7.1 has fewer one-to-many and many-to-many collaborations. In version 6.0, 303 resources, about 46% of the total externally visible resources, are implemented in about three packages on average. The use of polymorphism at the top package level was decreased in version 7.1. 272 resources are provided by more than one package. The majority of them are implemented in packages *draw* and *samples*.

	$1 \rightarrow 1$	$m \rightarrow 1$	$1 \rightarrow n$	$m \rightarrow n$
	144	201	112	191
Inports				
applet	3	–	7	8
application	3	12	22	16
contrib	34	2	74	151
figures	30	9	57	105
framework	11	17	–	–
samples	–	–	71	125
standard	32	93	61	139
util	31	68	35	36
total	144	201	327	580
Outports				
applet	–	40	4	9
application	3	73	12	29
contrib	41	163	10	164
figures	18	107	21	88
framework	–	–	–	–
samples	64	191	5	106
standard	15	29	35	131
util	3	23	25	42
total	144	626	112	569

Table 1. Resources visible in package *org.jhotdraw* from JHotDraw 6.0

	$1 \rightarrow 1$	$m \rightarrow 1$	$1 \rightarrow n$	$m \rightarrow n$
	403	119	182	90
Inports				
app	13	6	15	4
beans	–	–	–	1
draw	255	–	163	86
geom	27	63	–	–
gui	31	6	–	–
io	16	3	–	–
samples	32	–	182	90
undo	10	1	–	–
util	6	13	–	–
xml	13	27	4	1
total	403	119	364	182
Outports				
app	58	11	13	4
beans	–	–	–	1
draw	42	116	146	86
geom	–	–	–	–
gui	5	1	2	–
io	–	–	–	–
samples	298	115	15	89
undo	–	4	–	–
util	–	–	–	–
xml	–	2	6	2
total	403	249	182	182

Table 2. Resources visible in package *org.jhotdraw* from JHotDraw 7.1

- The resources that used by more than one package decreased from 392 resources in JHotDraw 6.0 to 209 resources in JHotDraw 7.1. This is largely due to the fact that code reuse via cross-package inheritance reduced significantly.
- In JHotDraw 7.1, package *sample* forms a framework-application relationship with package *draw* and *app*, while there is no clear framework-application pattern found in JHotDraw 6.0. Package *samples* from version 7.1 includes five applications that are built on the packages *app* and *draw*. They reused the default implementations for document-oriented applications provided by package *app*, and the basic drawing capabilities provided by package *draw*. On the other hand, package *samples* from version 6.0 exhibits high port similarity with three packages, *contrib*, *figures*, and *standard*. If there is a framework on which the package *sample* is built, the implemented basic functionalities of the framework are scattered among at least three packages.

According to the above observation, it seems that reducing coupling, especially inheritance coupling, between packages is the main concern in version 7.0.

5 Conclusion

In this paper, we proposed an approach to analyze package design using the Hybrid Model, a program model that enriches a package diagram with semantic information including the important properties of objects and their interrelationships. We identified four collaboration patterns among objects from different packages, and analyze package relations based on how objects are communicated with each other. Our case study shows that our approach can help in analyzing why the packages are organized the way they are, and can suggest possible criteria that led to the particular choice of partitioning. In the future, we will put the analysis approach in the context of package containment hierarchy, and define a set of metrics to help further understanding of package organization.

References

[1] X. Dong. *A Hybrid Model for Object-Oriented Software Maintenance*. PhD thesis, University of Waterloo, 2008.

[2] 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.

[3] JHotDraw 6.0 beta 1. http://www.jhotdraw.org/.

[4] JHotDraw 7. http://www.randelshofer.ch/oop/jhotdraw/.

[5] OpenCms — Professional Content Management. URL: http://www.opencms.org/.