A Service-Oriented Componentization Framework for Java Software Systems

Shimin Li and Ladan Tahvildari

Dept. of Electrical and Computer Engineering
University of Waterloo, Ontario, Canada, N2L 3G1
{shimin, ltahvild}@swen.uwaterloo.ca

Abstract

In the fast growing global market for services, service-oriented computing has drastically changed the way in which we develop software systems. Providing competitive services to these markets will be critical to the success of businesses and organizations. Some competitive services have already been implemented in existing systems. In this paper, we present a novel service-oriented componentization framework that automatically supports: i) identifying critical business services embedded in an existing Java system by utilizing graph representations of the system models, ii) realizing each identified service as a self-contained component that can be deployed as a single unit, and iii) transforming the object-oriented design into a service-oriented architecture. A toolkit implementing our framework has been developed as an Eclipse Rich Client Platform (RCP). Our initial evaluation has shown that our framework is effective in identifying services from an object-oriented design and migrating it to a service-oriented architecture.

1 Introduction

An efficient and cost-effective software reuse process helps organizations to increase the productivity, quality and reliability of the software system, in less time and lower cost. Component-Based Software Engineering (CBSE) is a successful code reuse process. However, one major issue in CBSE is the lack of a list of components with high reusability, mobility, and adaptability, despite the large amount of software that already exists in the portfolios of many software vendors. Therefore it is crucial for CBSE to seek a cost-effective way to identify and extract reusable components from existing software systems. An effective way of leveraging the value of legacy systems is to expose their functionalities as reusable components to a larger number of clients through well-defined component interfaces.

Today, more and more organizations are migrating to service-oriented architectures (SOA) to achieve net-centric

operations. The migration offers the potential of leveraging legacy systems by exposing some parts of them as services within SOA. However, there is often lack of effective engineering approaches for identifying, describing, modeling, and realizing services embedded in existing systems. The core of an SOA is a service which is a coarse-grained, discoverable, and self-contained software entity that are interacting with applications and other services through a loosely coupled, often asynchronous, message-based communication model.

We propose a service-oriented framework that componentizes an object-oriented system to re-modularize the existing assets to support service functionality. The proposed framework automatically supports: i) identifying critical business services embedded in an existing Java system, ii) realizing each identified service as a self-contained component, and iii) transforming the object-oriented design into a service-oriented architecture. This paper contributes in several research directions by: i) presenting comprehensive graph representations of object-oriented systems in different levels of abstraction, ii) introducing an incremental program comprehension approach, iii) exploring an efficient effective way to identify services embedded in an existing object-oriented system, and iv) proposing a novel approach to migrate object-oriented designs to service-oriented architectures.

The remainder of the paper is organized as follows. We review the related work in Section 2, and give an overview of the proposed framework in Section 3. In Section 4, we introduce the architecture recovery and its representation, and in Section 5, we present techniques and algorithms for identifying services embedded in existing systems. We introduce the approach for packaging identified services into self-contained components and discuss the system transformation process in Section 6 and 7, respectively. In Section 8, we present the evaluation criteria, and in Section 9, we discuss the results obtained by applying the framework on an open source Java software system. We summarize our work and outline directions for further research in Section 10.

Figure 1. The Architecture of the Service-Oriented Componentization Framework.

2 Related Work

Currently, most legacy systems are integrated in service-oriented applications by using wrappers and adaptors. In [23], the authors presented a framework to address these issues on migrating legacy systems into a web-enabled environment by involving the CORBA wrapper and SOAP/CORBA IDL translator. Caldiera and Basili introduced the *Care* (*Computer Aided Reuse Engineering*) system to support identifying reusable components using user-defined *reusability attribute model* based on software metrics in the context of a procedural paradigm [6]. Etzkorn and Davis presented an approach for identifying reusable classes from object-oriented systems based on the understanding of comments and identifiers in the source code [10]. Their tool *CHRis* uses natural-language techniques to help users decide whether a class implements certain useful functionality. Shin and Kim proposed techniques for transforming an available object-oriented design into a component-based design [18]. Their techniques focus on formal model specification and transformation. In [3], Bansiya and Davis introduced a *Quality Model for Object-Oriented Design* (QMOOD) which measures functional, structural and relational details of the system based on high-level attributes. In the model, they calculate reusability based on coupling, cohesion, and design size. None of these methods, however, provides hierarchical structures and proposes the reconstruction of the system's original architectural design. We aim to develop techniques for recovering high-level design, to extract the service hierarchy embedded in object-oriented systems, and to migrate object-oriented designs to service-oriented architectures.

Dominance analysis from graph theory allows one to locate subordinated software elements in a rooted dependency graph. Dominance analysis on call graphs of procedural language applications has been used in reverse engineering to identify modules and subsystems and recover system ar-

chitectures [5, 9, 11]. In this research work, we explore the use of dominance analysis to identify services from an object-oriented application. We extend the refactoring approach presented in [21] to automatically generate an interface for each component corresponding to an identified service.

3 The Proposed Framework

As illustrated in Figure 1, the proposed componentization framework is comprised of four stages :

- **Architecture Recovery :** The source code models and the UML-compliant architectural model of the system are extracted in this stage. The models are described as graphs and exported as XML documentations.
- **Service Identification :** In this stage, the top-level services of the system and the low-level services underneath each top-level service are identified where the services are modeled.
- **Component Generation :** In this stage, the identified services are packaged into self-contained components.
- **System Transformation :** In this stage, based on the packaged components, the object-oriented design is migrated to a service-oriented architecture.

The following sections elaborate further on each of the processes contained in the above stages.

4 Architecture Recovery Process

Software architecture recovery aims at reconstructing views on the architecture as-built. Architecture recovery from legacy systems has been claimed to offer great contributions to software maintenance and reuse. There are two goals we are trying to achieve in this stage : i) having complete data models for Java source code at different abstracted levels to support a wide range of structural analysis and recovery, and

ii) establishing a repository of relationships among classes and interfaces which can easily be queried in the service identification stage. There are two main processes in this stage that are explained further in the following sections.

4.1 Fact Extraction

The fact extraction is an automated process to parse a Java program to extract the relationships among its entities. The extracted facts are presented as abstract source code models which describe the Java classes, packages, and source files. These models are essential for representing the system at source code level and computing reusability attributes for each individual class. The source code models are presented as XML documents and form the *Basic View* (BView) of the system [13].

4.2 Architecture Modeling

Before presenting the details of this process, we define some concepts and their representations used in this research.

Definition 4.1 A *Labeled Directed Graph (LDG)* is a tuple $\Gamma(V, E, L_V, L_E, l_V, l_E)$, where V is a set of nodes, E is a set of edges, L_V is a set of node labels, L_E is a set of edge labels, $l_V : V \rightarrow L_V$ is a label function that maps nodes to node labels, and $l_E : E \rightarrow L_E$ is a label function that maps edges to edge labels.

Definition 4.2 The *Class/Interface Relationship Graph (CIRG)* of an object-oriented system is an LDG defined in Definition 4.1, where V is the set of all classes/interfaces of the system, $l_V(v)$ returns the full name (i.e. package name concatenates class or interface name) of v for any $v \in V$, $E = \{(v, w) \in V \times V \mid v \text{ references } w\}$, and $l_E(e)$ returns the types of relationships between the source node and target node of e for any $e \in E$.

Definition 4.3 The *Class/Interface Dependency Graph (CIDG)* of an object-oriented system is an LDG defined in Definition 4.1, where V is the set of all classes/interfaces of the system, $l_V(v)$ returns the full name (i.e. package name concatenates class or interface name) of v for any $v \in V$, $E = \{(v, w) \in V \times V \mid v \text{ references } w\}$, $L_E = \phi$, and hence $l_E(e)$ returns an empty label for any $e \in E$.

In order to comply with UML, the considered types of relationships between two classes/interfaces in Definition 4.2 are *inheritance*, *realization*, *association*, *aggregation*, *composition*, and *usage* which are adapted from UML 2.0 superstructure specification [2]. Based on the abstract source code models obtained from the source code, the architecture modeling process generates the CIRG and CIDG. By their definitions, the CIRG is a UML-compliant model and the CIDG is a further abstraction of the CIRG. In addition to building the CIRG and CIDG, reusability attributes for each class are computed and integrated into these graphs. The

service identification and extraction tasks in the next stage are performed upon the transformation of these graphs. We export the CIRG and CIDG as XML documents that can be easily queried in the service identification stage. The CIRG and CIDG form the *Structural View* (SView) of the system [13].

5 Service Identification Process

Our service identification, as shown in Figure 2, is supported by a combination of top-down and bottom-up techniques. In the top-down portion of the process, we identify the top-level services and the atomic services (to be discussed later) underneath each top-level service. In the bottom-up portion, we aggregate the atomic services to identify services with higher level of granularity.

Figure 2. The Process of Service Identification.

5.1 Service Representation

We categorize services embedded in a system into two categories: i) *Top-Level Services* (TLSs) that are not used by another service but may contain a hierarchy of low-level services further describing the service, and ii) *Low-Level Services* (LLSs) that are underneath a top-level service and can be weaved together to produce a new service (the desired business result). From the user's point of view, top-level services are provided by the system that can be accessed

independently. These services are hence independent from each other. In this research, we describe a service (both top-level and low-level) as a tuple $(name, C_F, SHG)$ and export it as an XML document that is defined by the XML schema illustrated in Figure 3. We adapt the UML profile defined in [17] to represent the XML schema defined in this work. In the above tuple, $name$ is the name of the service, C_F is the facade class set that contains classes/interfaces presenting the functionality of the service to outside world, and SHG is the *Service Hierarchy Graph* (SHG) of the top-level service represented by the tuple. The SHG is defined in Definition 5.1.

Figure 3. The UML Representation of XML Schema for a Service.

Definition 5.1 The *Service Hierarchy Graph* (SHG) corresponding to a top-level service is a rooted LDG, where the root, $r \in V$, represents the top-level service, $V \setminus r$ represents the set of low-level services contained in the top-level service, $l_V(v)$ returns the C_F set of v for any $v \in V$, $E = \{(v, w) \in V \times V \mid v \text{ contains } w\}$, $L_E = \emptyset$, and hence $l_E(e)$ returns an empty label for any $e \in E$.

The SHG shows the structural relationships between the services underneath a top-level service. It gives a high-level representation of services that is understandable by both developers and business experts. Furthermore, the SHG describes the modularization of its top-level service. Note that there is no SHG corresponding to a low-level service because the service has already been presented in the SHG of its top-level service. The SHGs of all top-level services form the *service view* (ServView) of the system.

5.2 Dominance Analysis

Dominance analysis is a fundamental concept in compiler optimizations and has been used extensively to identify loops in basic block graphs [15]. Cimitile and Visaggio introduced dominance analysis as a method to identify related parts of an imperative system [9]. This idea was elaborated further in [5, 11]. They applied dominance analysis on call graphs of applications based on procedural languages to identify modules and subsystems. In this research, we explore the use of dominance analysis on SHGs.

Definition 5.2 Let $G = (V, E, r)$ be a rooted directed graph where V represents all nodes in G , E represents all

edges in G , and $r \in V$ is the root node of G . Given any two different nodes $v \in V$ and $w \in V$, node v dominates node w , written $v \text{ dom } w$, iff every path from root r to w contains v . Node v directly dominates node w , written $v \text{ ddom } w$, iff all the nodes that dominate w dominate v . Node v strongly directly dominates node w , written $v \text{ sddom } w$, iff $v \text{ ddom } w$ and v is the predecessor of w .

Definition 5.3 Let $G = (V, E, r)$ be a directed graph where V represents all nodes in G , E represents all edges in G , and $r \in V$ is the only root node of G . The dominance tree corresponding to G is a tree $T = (V, E_d, r)$ where $E_d = \{(v, w) \in V \times V \mid v \text{ ddom } w \vee v \text{ sddom } w\}$. The maximal sddom subtree is a maximal subtree that contains only sddom edges.

Figure 4 shows a simple directed graph and its dominance tree. There are two maximal sddom subtrees in this dominance tree: $\{3, 7, 10\}$ and $\{6, 9\}$.

Figure 4. (a) A Simple Directed Graph. (b) The Dominance Tree Corresponding to the Graph in (a).

In the analysis process of software reverse engineering, it is essential to have an effective way of abstracting information. The dominance tree provides such an abstraction. More importantly, it represents high-level modularization of the software system through its branches. Each branch of the dominance tree represents a concept or high level functionality of the system. In the context of object-oriented design, one benefit of using dominance trees in program comprehension is the reduction of the visualization complexity of the class dependency graph by decreasing a large number of edges. In the class dependency graph of a real world software system, a class may have been referenced by hundreds of classes and a reduction to a single edge on the dominance tree greatly clarifies the graphic.

5.3 Top-Level Service Identification

Top-level services partition a system into independent parts. We identify services of the system by starting with the top-level services and then extracting a service hierarchy for each top-level service. To identify the top-level services, the CIDG is decomposed into a set of connected directed graphs

with an unique root such that each graph is an independent partition of the CIDG. Algorithm 5.1 describes the decomposition process. We name each of the result graphs as a *Modularized CIDG* (MCIDG). Each MCIDG might therefore embed a top-level service. Algorithm 5.2 describes the details of top-level service identification.

Algorithm 5.1: CIDG-Transformation

Input: *CIDG*

Output: A set of *MCIDGs*

```

MCIDGs ← ∅;
CGraphs ← ConnectedComponents(CIDG);
foreach graph g ∈ CGraphs do
    RGraphs ← RootedComponents(g);
    MCIDGs ← MCIDGs ∪ RGraphs;

```

In Algorithm 5.1, *ConnectedComponents()* computes the connected components of a directed graph, $G(V, E)$. Two nodes u and v in V are *connected* if there is a path between u and v . u and v are *path equivalent* if there is a path from u to v and a path from v to u . Node connection partitions V into maximal disjoint sets of connected nodes and these sets are called *connected components* of G . While path equivalence partitions V into maximal disjoint sets of path equivalent nodes and these sets are called *strongly connected components* of G . *RootedComponents()* decomposes a connected directed graph, G , into a set of rooted components. A rooted component is a subgraph of G that consists a unique root and all its successors. Note that the output MCIDG is a connected directed rooted graph and each node in MCIDG represents a single class or interface.

Algorithm 5.2: Top-Level Service Identification

Input: *CIDG*.

Output: *TLSs*: A set of identified top-level services that are represented by $(name, C_F, SHG)$ tuples.

```

/* Top-level service candidate identification */
MCIDGs ← Run CIDG-Transformation Alg. on CIDG;
Candidates ← ∅;
foreach  $MCIDG(V_m, E_m) \in MCIDGs$  do
    Create a new graph  $G(V, E)$ ;
     $V \leftarrow \emptyset$ ;  $E \leftarrow E_m$ ;
    for  $i \leftarrow 1$  to  $|V_m|$  do
         $V(i) \leftarrow Facade(V_m(i), MCIDG, CIDG)$ ;
    Create a new tuple  $T(name, C_F, SHG)$ ;
     $T.name \leftarrow null$ ;  $T.C_F \leftarrow Root(G)$ ;  $T.SHG \leftarrow G$ ;
    Add tuple  $T(name, C_F, SHG)$  to Candidates;
/* Candidate validation and concept assignment */
TLSs ← ∅;
foreach tuple  $T \in Candidates$  do
    The user validates the candidate by examining  $T.C_F$ ;
    if  $T$  is acceptable then
         $T.name \leftarrow$  Meaningful name for the service;
        Add  $T(name, C_F, SHG)$  to TLSs;

```

For a given class/interface set in a MCIDG and the CIDG, *Facade()* in Algorithm 5.2 computes the set of classes/interfaces that have incoming edges from classes/interfaces in the CIDG but not in the MCIDG. *Root()* returns the root of a given directed graph. The user validates a candidate by examining its facade class set since classes in the set represent the functionality of the service. At this stage, the SHG corresponding to each top-level service is built from the MCIDG and therefore can be viewed as a subgraph of the CIDG. In other words, the SHG is a abstraction of a MCIDG hiding the non-necessary information for understanding the service hierarchy. The functionality of low-level services in the hierarchy is provided by a single class. Hence, these services are called *atomic services*. In most cases, these atomic service are too fine-grained and have little reusability. However, the SHG at this stage provides us a right starting point to identify services with greater granularity by using service agglomeration techniques that are presented in the following sections.

5.4 Low-Level Service Identification

SHGs built in Section 5.3 are rooted directed graphs that represent the structural dependency between the top-level service and its atomic services. As we have mentioned, these atomic services are very fine-grained and therefore have very limited reusability. At this stage, we aim to agglomerate highly related atomic services to build a new SHG for each top-level service such that the services contained in the new SHG have higher granularity and thus present a higher potential of reuse. The service agglomeration is an iterative process and the desired new SHG is achieved incrementally. The services obtained from each iteration have higher granularity and modularize the top-level service in a different way. The result services of each iteration are presented as an intermediate SHG to users. Then, users can make a decision on repeating or terminating the process according the pre-defined termination criteria.

Algorithm 5.3 describes the low-level service identification process for a given top-level service. *MQ()* in Algorithm 5.3 computes the *Modularization Quality* (MQ) metric of a given top-level service. The MQ metric was first introduced in [14]. It has been used in a number of software engineering projects to evaluate the quality of software modularization achieved by graph partitioning [8, 19]. Let $C(G_1, G_2, \dots, G_k)$ be a partition of a given graph $G(V, E)$, the MQ metric is defined as follows:

$$MQ(C, G) = \frac{\sum_{i=1}^k s(G_i, G_i)}{n} - \frac{\sum_{i=1}^{k-1} \sum_{j=i+1}^k s(G_i, G_j)}{n(n-1)/2} \quad (1)$$

The function $s()$ used in Formula (1) is defined as the ratio of the actual number of edges between two subsets of V of graph G with respect to the maximum number of possible

edges between those two sets [8]. Let $U \in V$ and $W \in V$, then we have

$$s(U, W) = \frac{e(U, W)}{|U||W|} \quad (2)$$

where $e(U, W)$ denotes the number of edges connecting a vertex in U to a vertex in W . MQ determines the quality of the modularization quantitatively as the trade-off between inter-connectivity and intra-connectivity of subsystems. This trade-off is based on the assumption that well-designed software systems are organized into cohesive subsystems that are loosely interconnected. Hence, MQ is designed to reward the creation of highly cohesive clusters, and to penalize excessive coupling between clusters. The value of MQ is between -1 (no internal cohesion) and 1 (no external coupling). A straightforward consequence is that a higher MQ value can be interpreted as better modularization since it corresponds to a partition with either fewer edges connecting vertices from distinct blocks, or with more edges lying within identical blocks of the partitions, which is what most clustering and modularization algorithms aim to achieve [8].

Algorithm 5.3: Low-Level Service Identification

Input: $CIRG, CIDG, T(name, C_F, SHG)$

Output: $LLSs$: A set of identified low-level services that are represented by $(name, C_F, SHG)$ tuples.
 $T(name, C_F, SHG)$: The input top-level service with new SHG

Compute $MQ(T.SHG, CIDG)$;

repeat

Run Service-Agglomeration Alg. on $T.SHG$;
Update $T.SHG$ to the newly output SHG;
Compute $MQ(T.SHG, CIDG)$;

until *Termination Criteria is satisfied*;

$LLSs \leftarrow \phi$;

foreach non-root node $v \in T.SHG$ **do**

Create a new tuple $L(name, C_F, SHG)$;
 $L.name \leftarrow$ Meaningful name for the service;
 $L.C_F \leftarrow l_V(v)$; $L.SHG \leftarrow \phi$;
Add $L(name, C_F, SHG)$ to $LLSs$;

We define two *Termination Criteria* to stop the service agglomeration in Algorithm 5.3. The first criterion is that the value of MQ becomes greater than a user-defined threshold value. This means that the top-level service has been modularized nicely. The second criterion is that low-level services are presenting appropriate granularity. Granularity of services must be matched to the level of reusability and flexibility required for a given the context.

Algorithm 5.4 agglomerates highly related low-level services into a service with higher granularity and reconstructs a new SHG using these new services. The new SHG contains fewer services with higher granularity than the input SHG. In order words, it modularizes the corresponding top-

Algorithm 5.4: Service-Agglomeration

Input: $SHG, CIRG, CIDG$

Output: SHG_{new} - A new service hierarchy graph contains services with higher granularity.

$SHG \leftarrow CollapseCliques(SHG, CIRG, CIDG)$;
 $SHG \leftarrow CollapseStronglyConnectedComponents(SHG)$;
 $DTree \leftarrow GenerateDominanceTree(SHG)$;
 $ReduceDominanceTree(DTree, Heuristic1)$;
 $ReduceDominanceTree(DTree, Heuristic2)$;
 $SHG_{new} \leftarrow ReconstructSHG(DTree, CIDG)$;

level service in another way. $CollapseCliques()$ collapses the services in a 3-clique in the input SHG if the similarity of services in the clique exceeds a user-defined threshold. We develop a methodology for computing the similarity between two service based on the coupling analysis of the classes that implements these services. A *clique* in a directed graph is a collection of nodes that each pair of nodes are joined by an edge. If we try to add any other node to the collection, this will be no longer the case. A *k-clique* is a clique of order k . An example of such a clique can be obtained from Figure 4 (a). The set $\{3, 6, 7\}$ is a clique of order 3. $CollapseStronglyConnectedComponents()$ iteratively detects the strongly connected components (described in Section 5.3) in a directed graph and then collapses all nodes in the component into one node and updates the edges accordingly until there is no strongly connected component any more. Consequently, the output graph of this function is a *directed acyclic graph* (DAG). $GenerateDominanceTree()$ produces a service dominance tree from a given SHG. $ReduceDominanceTree()$ reduces a dominance tree by applying a given reducing heuristic. We define two reducing heuristics as follows:

Heuristic 1 Remove each maximal *ssdom* subtree by only keeping the root node of the subtree.

Agglomerating all services that are part of a maximal *ssdom* subtree into a service makes sense because these services constitute an independent unit that can only be accessed by the rest of services of the system through the root of the subtree. In order to simplify the visualization, we only need to present the root because the rest of the subtree is only visible to the root and can be hidden in the root.

Heuristic 2 Remove all leaf nodes in a subtree that contains both *ddom* and *sddom* edges, which are linked to the root of the subtree by *sddom* edges.

These leaf nodes represent low level services which are only accessible to the service represented by the root of the subtree. Therefore, these low level services can be considered sub services of the root. Function $ReconstructSHG()$ recovers the service hierarchy for the services presented in a

service dominance tree. It needs the CIDG to provide extra information since the service dominance tree is a abstraction of service hierarchy graph with some information lost.

6 Component Generation Process

A service-oriented architecture encourages individual services to be self-contained. At the third stage of the proposed framework as illustrated in Figure 1, we package each top-level service and the low-level services contained in its SHG into self-contained components.

To automate the process, we describe a component as a tuple, $\Gamma(name, i_f, C_F, C_C, CHG)$, and export it as an XML document defined by the XML schema depicted in Figure 5. In the tuple, $name$ is the name of the component, i_f is the interface that provides the entry point of the component, C_F has the same meaning as in a service, C_C is the set of classes/interfaces that are necessary to implement the component, and CHG is the abbreviation of *Component Hierarchy Graph* that is associated to a top-level component to present its low-level components.

Figure 5. The UML Representation of XML Schema for Component.

Given the identified services, the component generation process is performed automatically. In order to describe this process clearly, we need the following definitions about the reachability concepts in the CIDG and CIRG.

Definition 6.1 Node $w \in CIDG.V$ is said to be reachable from node $v \in CIRG.V$ if there exists a directed path from v to w , denoted by $v \xrightarrow{*} w$.

Definition 6.2 Node $w \in CIRG.V$ is said to be inheritance (realization) reachable from node $v \in CIRG.V$ if there exists a directed path from v to w and the labels of all edges in this path contain inheritance (realization) types, denoted by $v \xrightarrow{IN*} w$ ($v \xrightarrow{RE*} w$).

Let $serv$ be a service represented by a tuple $serv(name, C_F, SDG)$, the key steps, which yield the tuple $comp(name, i_f, C_F, C_C, CHG)$ representing the corresponding component, are enumerated as follows:

- $comp.name = serv.name$.
- $comp.C_F = serv.C_F$.
- $comp.C_C = comp.C_F \cup \bigcup_{all\ c \in comp.C_F} \{v \in CIRG.V \mid (c \xrightarrow{*} v)\}$.

- Create a new interface named i_f . Modify each class in $comp.C_F$ to implement i_f . Modify each interface in $comp.C_F$ to extend i_f .
- Add declarations of all public methods defined in each class in V_{IN} to i_f , where $V_{IN} = \bigcup_{all\ c \in comp.C_F} \{v \in CIRG.V \mid (c \xrightarrow{IN*} v)\}$.
- Copy declarations of all public methods declared in each interface in V_{RE} to i_f , where $V_{RE} = \bigcup_{all\ c \in comp.C_F} \{v \in CIRG.V \mid (c \xrightarrow{RE*} v)\}$.
- Add declarations of setter and getter methods for all public class fields declared in each class/interface in $comp.C_F$ to i_f . Implement the corresponding setter and getter methods in classes in $comp.C_F$.
- $comp.i_f = i_f$.
- $comp.CHG = \begin{cases} serv.SDG & serv.SDG \neq \phi; \\ \phi & \text{otherwise.} \end{cases}$

Once the tuple $(name, i_f, C_F, C_C, CHG)$ for a component has been constructed, we can package all classes/interfaces in C_F and C_C together with the newly created interface i_f into a JAR file named $name.jar$. The packaged component is self-contained and loosely coupled, so that each component can be used independently.

7 System Transformation Process

In the previous stages of the proposed framework, we have identified and extracted the top-level and the low-level components. At this stage, we need to introduce a transformation technique that automatically reconstructs the source system into a component-based target system. The target system, as depicted in Figure 6, is composed of one or more top-level components and a set of classes/interfaces.

Figure 6. Meta-model for the Target System.

We reconstruct the target system by adopting a bottom-up integration technique that collaborates with the extracted

components, starting with the components in the lowest position in the component hierarchy. Algorithm 7.1 describes the transformation process which takes the source system and the extracted components represented as tuples in the form of $(name, i_f, C_F, C_C)$. The output of the algorithm will be an instance of the meta-model described in Figure 6.

Algorithm 7.1: System-Transformation

Input: Source system, the extracted components

Output: A component-based system

```

foreach Top-level component t do
  while  $|t.CHG| > 0$  do
     $c \leftarrow$  node without descendants in  $t.CHG$ ;
     $P \leftarrow$  parents of  $c$  in  $t.CHG$ ;
    /* Refactoring the parents of  $c$  */
    foreach  $p \in P$  do
      Change the code of classes in  $p.C_C$  that reads
      (or writes) the public fields of classes in  $c.C_F$  to
      the code that invokes the corresponding getter
      (or setter) methods in interface  $c.i_f$ ;
      Replace the reference types in classes in  $p.C_C$ ,
      which refer to any classes in  $c.C_F$ , with
      interface  $c.i_f$ ;
    /* Update  $t.CHG$  */
    Remove node  $c$  from  $t.CHG$ ;

```

8 Evaluation Criteria

Since the proposed framework is trying to extract reusable components from an object-oriented system and migrate the object-oriented design to a service-oriented architecture, the evaluation criteria need to address *component reusability* and *architectural improvement*. The acquired components are structurally reusable for two reasons. First, the internal structures are encapsulated and second, the components are self-contained and thus have no dependency upon the entities outside of them.

Figure 7. Component Reusability Model.

In order to quantify the reusability of the components extracted by our framework, we define a component reusability model, as illustrated in Figure 7, from the user point of view. This model is an adaptation of the reusability model introduced by Washizaki et al. [22]. We extend Washizaki's model to quantify the complexity of components by utilizing the metric *Reference Parameter Density* (RPD) proposed

in [4]. Thus, the adapted model includes aspects related to the *Understandability*, *Adaptability*, and *Portability* factors given by ISO 9126 [1]. According to the model, we formulate reusability measurement as follows:

$$\begin{aligned}
 Reusability = & w_{complexity} * RPD + \\
 & w_{observability} * RCO + \\
 & w_{customizability} * RCC + \\
 & w_{ex-dependency} * \left(\frac{SCC_r + SCC_p}{2} \right)
 \end{aligned} \tag{3}$$

By their definitions, the values of all metrics in above formula are in $[0, 1]$. Since the complexity and external dependency have a negative effect on reusability, the weight $w_{complexity}$ and $w_{ex-dependency}$ could be values in $[-1, 0]$, while the observability and customizability have a positive effect, hence weight $w_{observability}$ and $w_{customizability}$ could be any values in $[0, 1]$. Nevertheless, the sum of these four weights is set to 1. Consequently, the reusability value will be in $[0, 1]$ and a higher value represents a higher level of the reusability.

We wish to measure the degree of conformance that the target (restructured) architecture has to the architectural principles of high intra-module cohesion and low inter-module coupling. Entropy from an information theoretic point of view has been proposed in [20] for evaluating the structuredness of a software design. We adopt the definition of entropy for an object-oriented design introduced in [7] to compute the entropy of our source systems and target systems, respectively. The smaller the entropy value, the better structure the system has. We then compare the results to see whether the structures of our target systems are improved. The entropy of a object-oriented system S with n classes is defined as follows:

$$H(S) = - \sum_{i=1}^n p(c_i) \log_2 p(c_i) \tag{4}$$

where the probability $p(c_i)$ is extracted from the CIDG as the ratio of the number of incoming edges of class c_i over the total number of edges in the CIDG.

9 Jetty: A Case Study

The proposed service-oriented componentization framework has been implemented as an Eclipse Rich Client Platform (RCP) [16] named *JComp - Java Componentization Kit* [13]. In this section, we apply the JComp on Jetty [12] to empirically evaluate the proposed framework. Jetty is an open source HTTP and Servlet Server written in Java. The version we worked on is 5.1.10 that has 45K LOC source code. It consists of 318 Java source files that defines 273 classes and 47 interfaces distributed in 25 packages.

To extract reusable components from Jetty, we applied the JComp toolkit to identify the services embedded in the

Figure 8. The Reusability Measurement for the Components Extracted from Jetty.

system. We ran the Parser and the Modeler, two plug-ins of JComp, to build the source code models and system architectural models that are represented by CIRG and CIDG. Based on the CIRG and CIDG, the Extractor, another plug-in of JComp, first identified 33 top-level service candidates from the CIDG by running Algorithm 5.2. Then, we validated each candidate by examining the facade class set of the candidate. Finally, we accepted 16 top-level services that are shown in Table 1. The unacceptable candidates are dead code, debugging modules, or testing modules. For instance, we found 8 dead classes in *org.mortbay.util* package and a debugging module whose entry point is the class *org.mortbay.servlet.ProxyServlet*.

ID	Top-Level Service	N1	N2
T1	Win32 Server	248	11
T2	Dynamic Servlet Invoker	207	12
T3	Jetty Server MBean	126	9
T4	Proxy Request Handler	113	7
T5	XML Configuration MBean	87	5
T6	Web Application MBean	86	6
T7	Administration Servlet	56	5
T8	CGI Servlet	49	5
T9	Host Socket Listener	46	5
T10	Web Configuration	34	3
T11	Authentication Access Handler	30	3
T12	Servlet Response Wrapper	27	2
T13	IP Access Handler	18	0
T14	Multipart Form Data Filter	16	2
T15	HTML Script Block	12	1
T16	Applet Block	9	1

Table 1. Identified Top-Level Services in Jetty.

After all top-level services are validated, the Extractor then identified low-level services contained in each top-level services by running Algorithm 5.3. For example, as shown in Table 2, there are 11 low-level services identified from *Win32Server* top-level service (T1) that run Jetty as a Windows HTTP server. The third column (N1) and the fourth column (N2) of Table 1 are the number of atomic

services (i.e., Java classes) and the number of identified low-level services underneath a top-level service, respectively. When identifying low-level services, we used the *criterion1* described in Section 5.4 to terminate the iteration in Algorithm 5.3 by setting $MQ = 0.75$.

Low-Level Service	Reusability
Jetty Server	0.9
Service Handlers	0.6
Resource Handler	0.7
Security Handler	0.7
Socket Listener	0.8
HTTP Connection	0.9
HTTP Request	0.7
HTTP Response	0.5
Web Application Context	0.6
Servlet	0.7
Servlet Handler	0.8

Table 2. Low-Level Services Identified in T1.

To compute the reusability of components generated by the Generator plug-in of JComp, we applied the component reusability model by computing Formula (3). In this empirical study, we set $w_{complexity} = -0.3$, $w_{observability} = 0.8$, $w_{customizability} = 0.8$, and $w_{ex-dependency} = -0.3$. Figure 8 shows reusability values of the top-level components and the average value of the low-level components contained in a top-level component. From Figure 8, it was observed found that all top-level components except C16 have reusability value above 0.5 and all the average values are between 0.6 to 0.8. Thus, we could conclude that components extracted by our framework have a reasonable level of the reusability.

The Transformer plug-in of the JComp toolkit transformed Jetty into a component-based system based on the generated components. We named the target system *Jetty-JComp*. By Algorithm 7.1, Jetty-JComp has the same functionality as Jetty. Jetty-JComp now contains 16 independent JAR files. Each JAR file provides a top-level service and can be used independently. Also, each independent JAR

file is a component-based system that consists of a set of JAR files. We have computed the entropy of both Jetty and Jetty-JComp by applying Formula (4). When computing the entropy of Jetty-JComp, we used the component hierarchy graphs instead of the CIDG because Jetty-JComp is comprised of components. We found that the entropy of the Jetty-JComp was reduced by 45.5%. Hence, we can safely conclude that our transformation dramatically improves the structure of the system.

10 Conclusions and Future Work

In this paper, we presented a service-oriented componentization framework for Java systems. The framework componentizes an object-oriented system to re-modularize the existing assets for supporting service functionality. We introduced an approach for identifying, modeling, and packaging critical business services embedded in the system. In addition to producing reusable components realizing the identified services, the framework also provides a component-based integration approach to migrate an object-oriented design to a service-oriented architecture. Our initial evaluation has shown that our framework is effective in identifying services from an object-oriented design and migrating it to a service-oriented architecture. Moreover, the BView, SView and ServView built by our framework help users gain a program understanding of the system.

Currently, we are working on extending the framework to measure the reusability and maintainability of the extracted component more concisely. We also aim to investigate algorithmic processes that can be used to automatically categorize the identified services.

References

- [1] Software product evaluation-quality characteristics and guidelines for their use. *ISO/IEC Standard ISO-9129*, 1991.
- [2] *UML 2.0 Superstructure Specification*. Object Management Group, Framingham, Massachusetts, USA, October 2004.
- [3] J. Bansiya and C. G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28:4–17, January 2002.
- [4] M. A. Boxall and S. Araban. Interface metrics for reusability analysis of components. In *Proceedings of the Australian Software Engineering Conference (ASWEC)*, pages 40–51, April 2004.
- [5] E. Burd and M. Munro. Evaluating the use of dominance trees for c and cobol. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 401–410, September 1999.
- [6] G. Caldiera and V. R. Basili. Identifying and qualifying reusable software components. *IEEE Computer*, 24:61–70, February 1991.
- [7] A. Chatzigeorgiou and G. Stephanides. Entropy as a measure of object-oriented design quality. In *Proceedings of the Balkan Conference in Informatics (BCI)*, pages 565–573, November 2003.
- [8] Y. Chiricota, F. Jourdan, and G. Melancon. Software components capture using graph clustering. In *Proceedings of the International Workshop on Program Comprehension (IWPC)*, pages 217–226, May 2003.
- [9] D. Cimitile and G. Visaggio. Software salvaging and call dominance tree. *Journal of Systems and Software*, 28:117–127, February 1992.
- [10] L. H. Etzkorn and C. G. Davis. Automatically identifying reusable oo legacy code. *Computer*, 30:66–71, October 1997.
- [11] J.-F. Girard and R. Koschke. Finding components in a hierarchy of modules: a step towards architectural understanding. In *Proceedings of the 13th International Conference on Software Maintenance (ICSM'97)*, pages 58–65, October 1997.
- [12] Jetty. A Java HTTP Server and Servlet Container. <http://jetty.mortbay.org/jetty/index.html>, 2006.
- [13] S. Li and L. Tahvildari. JComp: A reuse-driven componentization framework for Java applications. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, June 2006.
- [14] S. Mancoridis, B. Mitchell, Y. Chen, and E. R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 50–62, August 1999.
- [15] R. Morgan. *Building an Optimizing Compiler*. Butterworth-Heinemann, Boston, Massachusetts, 1998.
- [16] Rich Client Platform, 2005. www.eclipse.org/rcp.
- [17] N. Routledge, L. Bird, and A. Goodchild. Uml and xml schema. In *Proceedings of the the Australian Database Conference (ADC)*, pages 274–281, February 2002.
- [18] S. K. Shin and S. D. Kim. A method to transform object-oriented design into component-based design using object-z. In *Proceedings of the International Conference on Software Engineering Research, Management and Applications (SERA)*, pages 274–281, August 2005.
- [19] A. Shokoufandeh, S. Mancoridis, and M. Maycock. Applying spectral methods to software clustering. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 3–10, November 2002.
- [20] G. Snider. Measuring the entropy of large software systems. HP Technical Report HPL-2001-221, www.hpl.hp.com/techreports/2001/HPL-2001-221.pdf.
- [21] H. Washizaki and Y. Fukazawa. A technique for automatic component extraction from object-oriented programs by refactoring. *Science of Computer Programming*, 56:99–116, April 2005.
- [22] H. Washizaki, H. Yamamoto, and Y. Fukazawa. A metrics suite for measuring reusability of software components. In *Proceedings of the International Software Metrics Symposium (METRICS)*, pages 211–223, September 2003.
- [23] Y. Zou and K. Kontogiannis. Towards a web-centric legacy system migration. In *Proceedings of ICSE Workshop on Net-Centric Computing (NCC): Migrating to the Web, International Conference on Software Engineering (ICSE01)*.