Automated Software Engineering manuscript No.
(will be inserted by the editor)

Cloud Refactoring: Automated Transitioning to Cloud-Based Services

Young-Woo Kwon · Eli Tilevich

Received: date / Accepted: date

Abstract Using cloud-based services can improve the performance, reliability, and scalability of a software application. However, transitioning an application to use cloud-based services is difficult, costly, and error-prone. The required re-engineering effort includes migrating to the cloud the functionality to be accessed as remote cloud-based services and re-targeting the client code accordingly. In addition, the client must be able to detect and handle the faults raised in the process of invoking the services. As a means of streamlining this transitioning, we developed a set of refactoring techniques—automated, IDE-assisted program transformations that eliminate the need to change programs by hand. In particular, we show how a programmer can extract services, add fault tolerance functionality, and adapt client code to invoke cloud services via refactorings integrated with a modern IDE. As a validation, we have applied our approach to automatically transform two third-party Java applications to use cloud-based services. We have also applied our approach to re-engineer a suite of services operated by General Electric to use cloud-based resources to better satisfy the GE business requirements.

Keywords cloud computing · services · refactoring · service extraction · fault-tolerance · program transformation.

1 Introduction

One of the foundations of cloud computing is Software-as-a-Service (SaaS), a computing paradigm in which clients access a piece of remote, cloud-hosted functionality through a public interface. To take advantage of cloud-based services, centralized software applications must be re-engineered, so that a portion

Department of Computer Science, Virginia Tech
Blacksburg, VA 24060
{ywkwon,tilevich}@cs.vt.edu

of their functionality is hosted in the cloud. One may want to replace an existing application functionality with an equivalent cloud-based service for a variety of reasons, both business and technical. Replacing a locally implemented functionality with a remotely maintained service reduces the maintenance burden. A service provider can more effectively improve quality and reduce costs by leveraging the economies of scale, when the same service is used by multiple clients. A service may have access to computing resources that are superior to the resources of a local machine. Services often conglomerate other services, thus offering additional benefits. For example, if a service persists user data, it is likely to offer backup and restoration facilities not common outside of large server infrastructures.

Using cloud-based services has become a common avenue for leveraging remote computing resources, with the benefits that include reduced costs, increased automation, greater flexibility, and enhanced mobility [3]. Despite all the benefits of leveraging cloud-based resources, transitioning a centralized application to effectively use remote services requires extensive changes to the application's source code. In addition, the possibility of partial failure requires that proper fault handling functionality be added to any application that invokes services remotely. As a result, programmers manually transition applications to use cloud-based services, changing code in difficult, costly, and error-prone ways. Therefore, there is great potential benefit in automating these changes and making the automation available to the software engineering community.

A popular technique for automating common program transformations is called *a refactoring*. Defined generally, a refactoring is a semantics preserving program transformation performed under programmer control [10]. In other words, refactoring is automated: a programmer determines if a refactoring should be performed and then engages a refactoring engine that transforms the code automatically. In this article, we advocate the vision of using refactoring as a means of facilitating the transitioning to cloud-based services. We argue that transitioning an application to take advantage of cloud-based services preserves the application's semantics in the sense that the overall functionality does not change. Executing some functionality in the cloud does not change the semantics from the end user's perspective.

This article presents a set of refactoring techniques that facilitate the process of transforming centralized applications to use cloud-based services. These techniques automate the program transformations required to (1) render portions of functionality of a centralized applications as cloud-based services and re-target the application to access the services remotely; (2) handle failures that can be raised in the process of invoking a cloud-based service; and (3) switch a service client to use an alternate, equivalent cloud-based service. These refactoring techniques—collectively named *Cloud Refactoring*—have been concretely implemented in the context of the Eclipse IDE and added to its refactoring engine.

To validate *Cloud Refactoring*, we applied its constituent refactoring techniques to transform two centralized, monolithic Java applications to use cloud-

based services. We also applied *Cloud Refactoring* to re-engineer a commercial application used by General Electric (GE) to use cloud-based services in an effort to demonstrate how refactoring can help realize the GE strategic vision to take advantage of cloud computing.

Our results indicate that *Cloud Refactoring* can be a powerful tool for the programmer charged with the task of transitioning a centralized application to one that uses cloud-based services. Not only can *Cloud Refactoring* transform code with a high degree of automation, but it can also properly account for the demands of distributed execution. Thus, *Cloud Refactoring* represents a robust and pragmatic approach that can reduce maintenance costs and increase programmer productivity.

The rest of this article is structured as follows. Section 2 motivates our approach and then introduces the main technologies discussed in this article. Section 3 describes the new refactoring techniques. Section 4 reports our experiences of applying *Cloud Refactoring* to third-party applications. Section 5 compares our approach to the existing state of the art. Finally, Section 6 presents future work directions and concluding remarks.

2 Motivation and Technical Background

In this section, we introduce an example that motivates this research and then provide a technical background our approach uses.

2.1 Motivating Example

Consider JNotes [1], a third-party diary and project management application written to run on a single desktop machine. Our goal is to refactor JNotes into a cloud-based application, with the server part deployed on a remote server and the client part accessed from a mobile device. This transformation offers several advantages. All the JNotes documents and calendars can be saved at the remote server, whose file system can be regularly backedup and replicated, so that data consistency will not suffer from a failure of the client's file system. Furthermore, through a simple change in server deployment, JNotes can be made into a collaborative application, with multiple clients sharing the same server components.

A major technical impediment to realizing the transitioning outlined above is that maintenance programmers have to change the application's source code by hand. JNotes is a typical centralized application that comprises a collection of Java classes. Splitting JNotes into the service and client parts, deploying the services in the cloud, and having the parts communicate with each other reliably can quickly turn into a complex programming undertaking. Furthermore, the resulting cloud-based application is likely to contain software imperfections, commonly introduced when manipulating code by hand.

[1] `http://memoranda.sourceforge.net`

Although software frameworks have been introduced to ease rendering classes as Web services, the classes must adhere to a rigid set of architectural conventions. Thus, it is unlikely that these frameworks can help transition arbitrary classes to Web services. As an example, consider JAX-WS [11], a framework that represents a significant industry effort to simplify the development and deployment of Web services. With JAX-WS, a programmer can export a standard Java class as a Web service by annotating the class with `@WebService` and the the class's methods with `@WebMethod`. A code generation tool that comes with JAX-WS reads these annotations and creates all the required supporting harness to exposes and deploy the annotated methods as XML-based Web services.

However, this service extraction model simply renders existing methods as Web services without any consideration for the resulting performance and reliability. To ensure good performance and high reliability, the classes that are to become Web services may need to be restructured first. For example, a service may need to use only a subset of the class's fields, thus requiring splitting the class into client and server partitions.

Consider moving class `FileStorage` to the cloud as a means of saving all the JNotes documents in a shared cloud storage. With JAX-WS, the programmer can transform the entire class with all its methods into a Web service. Unfortunately, this "all or nothing" inflexible distribution model may fall short of meeting the needs of realistic applications. For example, some functionality in `FileStorage` pertains to local file paths, and as such should not be moved to the cloud. That is, the functionality tied to the client environment cannot be moved. More specifically, the programmer needs to split methods `storeResourcesList(...)` and `openResourcesList(..)` from the rest of the class before it is transformed into a remote service.

The refactoring approach that we advocate here enables the required level of flexibility when transforming classes into remote services. Our *Extract Service* refactoring takes as input a class name and a set of methods that are to be rendered as a remote service. This refactoring then transforms the given methods into remote service methods, leaves the remaining methods on the client, rewrites all communication between the original and remote methods into remote service calls.

Because centralized and distributed applications have different failure modes, simply rendering a subset of a centralized application remote does not preserve the semantics. Distributed applications are subject to partial failure, in which its different components (client, server, or network) may fail independently from each other. Although one cannot handle all the possible failures in a distributed application, some failures have well-known handling strategies. Thus, to better preserve the original execution semantics, the *Extract Service* refactoring also adds client-side fault tolerance functionality as specified by the programmer. For example, the programmer may specify that an unsuccessful attempt to reach a service be repeated a given number of times. In our approach, the programmer can specify and configure the fault tolerance strategies to apply by means of an XML-based domain-specific language.

Finally, a service application may need to use more than one service implementation for a given functionality. For example, in the case when one service implementation is not available, the client should switch to using a different service, whose method interface is different from that used by the original service implementation. Thus, the client code will need to be adapted to use different service interfaces. The *Adapt Service Interface* refactoring automates the transformations required to be able to switch the client to using an equivalent service exposed through an incompatible service interface.

2.2 Technical Background

In the following discussion, we provide an overview of the cloud and service computing technologies used in this article.

2.2.1 Cloud Computing and Services

Cloud computing provisions resources on-demand through three main virtualization approaches: (1) infrastructure virtualization—provisioning computing power, storage, and machine (e.g., Amazon EC2); (2) platform virtualization—provisioning operating systems, application servers, and databases (e.g., Amazon S3); (3) software virtualization—provisioning complete Web-based applications (e.g., `Salesforce.com`). The main benefits of cloud computing include elasticity, scalability, and availability. From the software development perspective, however, taking advantage of cloud computing requires that the programmer follow a strict set of architectural and design guidelines.

Service Oriented Architecture (SOA) provides uniform access to a variety of computing resources across multiple application domains. Loosely coupled services may be co-located in the same address space or be geographically dispersed across the network. Among the software engineering advantages of services are strong encapsulation, loose coupling, ease of reusability, and standardized discovery. In addition, due to the strong separation between service interfaces and implementations, service developers have the flexibility to mix any middleware platforms and applications as well as to switch service infrastructures without affecting service clients. It is these desirable software engineering properties that made SOA a widely used paradigm for realizing cloud computing solutions.

2.2.2 OSGi Framework as a Cloud Computing Platform

Open Service Gateway Initiative (OSGi) [25]—a service implementation and provisioning infrastructure—has been embraced by numerous industry and research stakeholders, organized into the OSGi Alliance [2]. As a service platform,

[2] Open-source OSGi implementations include Apache Felix (`http://felix.apache.org/site/index.html`) and Knopflerfish (`http://www.knopflerfish.org`). Among large commercial OSGi projects are the Spring Framework (`http://www.springsource.org/`) and the Eclipse IDE (`http://www.eclipse.org/equinox/`).

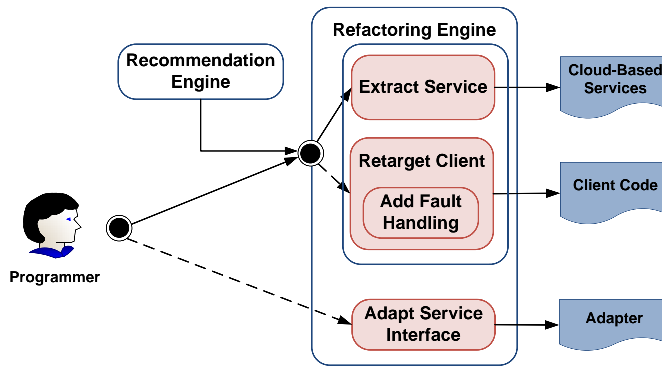

Fig. 1 Migrating to Cloud-Based Services via *Cloud Refactoring*.

OSGi can render any Java class as a service bundle. OSGi manages published bundles, allowing them to use each other's services through public interfaces. OSGi manages the lifecycle of a bundle (i.e., moving between install, start, stop, update, and delete stages) and allows it to be added and removed at runtime.

OSGi has made substantial inroads into the domain of cloud computing. Some enterprises have adopted OSGi as the platform for realizing their private clouds [26]. In light of these developments, a consortium of major industry and academia stakeholders has issued the Request for Proposal (RFP) 133 [24], which codifies how OSGi should be leveraged as a platform for cloud computing.

3 Our Approach: *Cloud Refactoring*

The goal of our approach is to alleviate the code transformation hurdles involved in adapting existing applications to take advantage of cloud-based services. To reduce development efforts/costs and increase programmer productivity, we have expressed as refactorings several common program transformations that programmers perform when adapting applications to use cloud-based resources. Although our approach is not fully automatic, programmers only determine if the source code should be transformed. The actual transformations are performed by a refactoring engine. In the following discussion, we first give an overview of our approach and then detail its individual parts.

3.1 Approach Overview

Our approach focuses on those common program transformations occurring when using cloud-based services that are well-amenable to be expressed as a

refactoring. In particular, we focus on three software re-engineering scenarios. One scenario involves moving some of a centralized application's functionality to the cloud. The second scenario involves adding fault tolerance functionality to the client to handle the faults raised during the invocation of a cloud-based service. The third scenario involves switching an application to use an alternate cloud-based service exposed through a different service interface.

Figure 1 shows how the constituent components of our approach fit together. The two main parts of our approach are *The Recommendation Engine* and *The Refactoring Engine*. The recommendation engine uses static and dynamic program analysis techniques to infer class coupling; this optional component can inform the programmer about which classes can be converted into a cloud-based service. The two refactoring techniques of the refactoring engine are intended be used à la carte. By integrating the engine with the Eclipse IDE, our approach makes it possible to use the new refactoring techniques indistinguishably from the existing ones. Furthermore, some of the existing, widely used refactoring techniques can be quite useful for applications that use cloud-based services. For example, *Extract Service* refactoring can be used to move a method to a class prior to converting that class to a cloud-based service.

3.2 Service Recommendation

To make sure that moving functionality to the cloud does not render the application unusable due to exploding latency costs, programmers should use service components rather than individual objects as a distribution boundary. Because few existing applications consist of service components, programmers should first ensure that an intended service is not tightly coupled with the rest of the application. For example, they can apply a *Façade* pattern that exposes some tightly coupled functionality through a crude-grained interface.

Nevertheless, it may be difficult to determine which functionality is a good candidate to be exposed as a cloud-based service. To that end, our approach provides a recommender tool that computes the coupling metrics for all the classes in an application and then displays the classes that are least tightly coupled. Accessing the functionality represented by these classes from a remote cloud-based service should impose only a limited performance penalty on the refactored application.

Figure 2 shows the process diagram for identifying classes that can be converted into cloud-based services. Our approach leverages two recommendation mechanisms: profiling- and clustering-based recommenders. The profiling-based recommender engages a static program analysis and runtime monitoring to collect program information. By combining the class coupling metrics collected through both static analysis and runtime monitoring, the recommendation algorithm then suggests a subset of an application that can be transformed to cloud-based services. The profiling-based recommender sorts application classes based on their execution duration and frequencies, so that

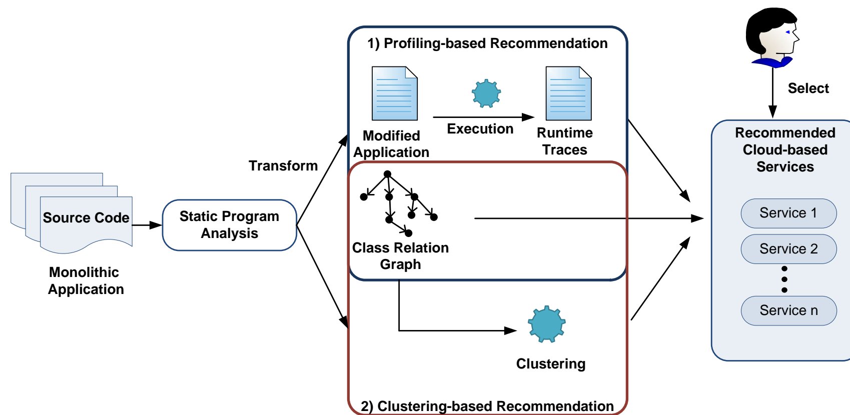

Fig. 2 Service recommendation process.

the programmer can know what classes are computation-intensive and how frequently they are accessed. The clustering-based recommender clusters classes with similar functionality, thus identifying class clusters whose functionality can be naturally exposed as a cloud-based service. Because the clustering-based recommender groups classes based on their functionality, the programmer can avoid duplicating a functionality in the cloud by selecting candidates for cloud-based service from different clusters.

Note that these recommendation mechanisms are provided as a tool that can inform the programmer about the properties of the applications about to undergo a refactoring. The programmer is ultimately responsible for deciding which classes should be transformed into cloud-based services and even if the transformation should take place to begin with. This design choice is in line with the automated nature of our refactoring techniques. In the following discussion, we describe our two recommendation mechanisms in detail.

3.2.1 Profiling-based Recommendation

In essence, the recommender strives to find a distribution strategy that would not render the application unusable due to the drastically increased latencies of invoking tightly coupled methods across the network. Because the recommender only takes the coupling metrics into consideration, it cannot produce a recommendation that is guaranteed to always exhibit a superior performance. Other factors, such as business logic and system resources used, can impact the performance drastically. As a result, the programmer can only use the recommender as a tool to explore the coupling metrics of the refactored application rather than as an absolute arbiter that determines which functionality is to be extracted into remote services.

Figure 3 shows our service recommendation algorithm that operates on a class relation graph. Given a graph, the algorithm calculates a service utility

INPUT:	A class relation graph, CRG
OUTPUT:	A set $CS = \{c_1, c_2, \ldots, c_n\}$ of possible remote classes

```
classes ⟵ calculateUtility(CRG);
destinations ⟵ edgesOutOf(class);

while (destinations ≠ ∅) do
  class ⟵ destinations.next();
  utility ⟵ class.getUtility();
  coupling ⟵ class.getCoupling();
  if (utility ≥ util_threshold && coupling ≤ coup_threshold) then
    CS.add(class);
  end if
  destinations ⟵ edgesOutOf(class);
end while
```

Fig. 3 Profiling-based service recommendation algorithm

value for each class in the program, a rank that expresses how fit a class is to be rendered as a cloud-based service. Specifically, the algorithm uses the service utility function defined as follows:

$$F(i) = \sum_{i \in edges} \{W_\alpha \times \frac{T_i}{MAX(T_0, .., T_n)} + W_\beta \times \frac{N_i}{MAX(N_0, .., N_n)}\}$$

where N, T, and W denote execution number, execution time, and weight to each measurement metric, respectively. If W_α is larger than W_β, classes related to business logic will be suggested. Conversely, if W_β is larger than W_α, frequently accessed classes will be suggested. Then, we defined our own coupling metrics as follows:

$$CP(i,j) = CC(i,j) + CR(i,j) = \frac{1}{\#ofhops} + \frac{\sum(e_i \cap e_j)}{\sum e_i + \sum e_j}$$

where CP, CC and CR denote coupling, class connectivity, and class relation values. Class connectivity, CC, denotes how the given two classes are closely connected. If class x_i has lower hops to go class x_j, they are strongly connected. If x_i and x_y are directly connected, $CC(i,j)$ is 1. Otherwise, $CC(i,j) = \frac{1}{\#ofhops}$. Class relation, CR, denotes how the given two classes are related. If class x_i creates, reads, writes, and invokes only class x_j, class x_i is tightly related to class x_j. CR is computed using the number of in/out edges from the given class to other classes. Then, the algorithm described traverses the graph from the root to the leaf nodes and then suggests cloud-based service candidates based on the calculated service utility values. Based on this suggestion, programmers can then choose classes that are suitable candidates for cloud migration.

3.2.2 Spectral Clustering-based Recommendation

The second recommender clusters related classes together. In recent years, spectral clustering has become one of the most widely used clustering algorithms. Spectral clustering techniques make use of the spectrum of the similarity matrix of the data to perform dimensionality reduction for clustering in

INPUT:	A class relation graph, CRG
OUTPUT:	A set $CS = \{c_1, c_2, \ldots, c_n\}$ of possible remote classes

```
clusterNum ⟵ 2;  //initialize the number of clusters
while (true) do
  SIM ⟵ constructSimilarityMatrix(CRG);
  cluster ⟵ buildCluster(SIM, clusterNum);

  if(cluster = ∅) then exit; end if

  while (cluster ≠ ∅) do
    class ⟵ cluster.next();
    if (class is accessed from other clusters) then
      CRG.remove(class);
      CS.add(class);
    end if
  end while

  clusterNum ⟵ clusterNum + 1;  //increase the number of clusters
end while
```

Fig. 4 Clustering-based service recommendation algorithm.

fewer dimensions. Given a set of data points $x_1, \ldots, x_n$ and similarity $S(i, j)$ between all pairs of data points x_i and x_j, the similarity matrix is defined as S. If the similarity $S(i, j)$ between the corresponding data points x_i and x_j is positive, two vertexes are connected. The similarity matrix is computed as follows:

$$S(i, j) = CC(i, j) + CR(i, j) + D(i, j) + L(i, j) + T(i, j)$$

where CC, CR, D, L and T denote class connectivity, class relation, class distance, library usage, and type similarity, respectively. We use the same formula to compute CC and CR, which are defined above. Class distance, $D(i, j)$ is calculated using the Levenshtein distance algorithm [16]. If x_i and x_j have the same package name, these classes are considered more similar to each other that classes in different packages. With respect to library usage, L shows how the relationship between two classes in terms of the similarity of the libraries they use. Using the same library indicates a similarity in functionality. If x_i and x_j use the same libraries, their $L(i, j)$ is 1. Otherwise, their $L(i, j)$ is 0. Finally, type similarity, T, denotes the similarity of classes in terms of their types. If the classes implement the same interfaces or inherit from the same super class, their $T(i, j)$ is 1. Otherwise, their $T(i, j)$ is 0.

Figure 4 shows the clustering-based service recommendation algorithm, which is parameterized with a class relation graph. First, the graph's similarity matrix is constructed. Since each node of the graph has method/class information, the similarity between each pair of classes is calculated according to their similarity metrics. Then, a recursive spectral clustering algorithm continuously partitions the similarity matrix until it reaches the base case (the partition size equals 1).

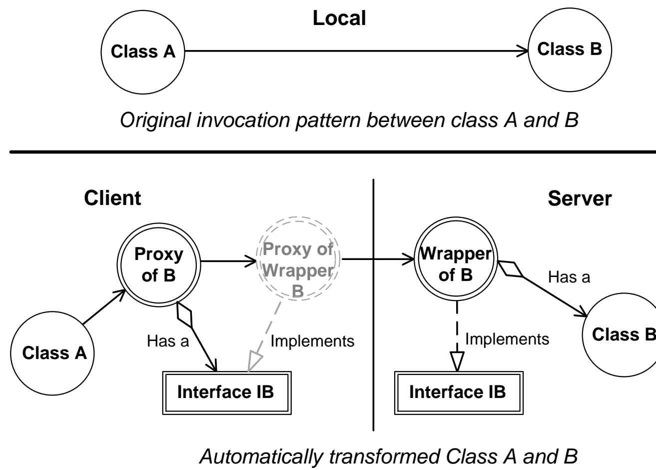

Fig. 5 *Extract Service* refactoring—service transformation and client redirection.

3.2.3 Constraints on Extracting Cloud-Based Services

Not all classes can be easily migrated into remote, cloud-based services. Various constraints make it impossible to refactor some classes for cloud-based execution. These constraints pertain to the use of local resources, parameter passing, and serialization. Classes that make use of local resources, such as databases, disk files, and sensors cannot be moved to be executed by a different host. In our refactoring approach, we assume that the programmer is aware of such local resource usage and would not try to migrate the affected classes to the cloud. Our refactoring techniques can only pass by-copy parameters, which includes primitive and read-only parameters. The classes whose methods contain other types of parameters cannot be transformed into cloud-based services; our recommenders identify and exclude such classes. Finally, OSGi requires that non-primitive remote method parameters be serializable and attempts to automatically serialize them.

Next, we describe two refactoring techniques that form the foundation of *Cloud Refactoring*: 1) Extract Service and 2) Adapt Service Interface.

3.3 Cloud Refactoring—1) *Extract Service*

Extract Service refactoring automates the program transformations required to transform regular classes into remote services. A typical *Extract Service* refactoring performs the following four program transformations: 1) rewrite a class making all its methods into remote service methods, 2) partition class methods into service methods and regular methods, rewriting all the communication between the two into remote service calls, 3) re-target all clients of

the original class to access its functionality in the cloud by means of remote service calls, and 4) add fault handling functionality to client code.

3.3.1 Transforming Program Code to Extract Services

Figure 5 shows local classes `A` and `B` can be transformed by means of the *Extract Service* refactoring, so that `B` becomes a fault tolerant cloud-based service. At the server side, class `B` is exposed via a generated interface `IB` and class `B` becomes wrapped into an instance of class `WrapperB`. The exposed service interface `IB` can be invoked via standard service protocols (e.g., HTTP-SOAP, REST, etc). At the client side, class `A` imports the service via interface `IB`, with the original class `B` at the client being replaced with a generated proxy class. This example demonstrates the transformation of all the methods in a class into cloud-based services. If only a subset of the methods are to be transformed into services, additional transformations are necessary.

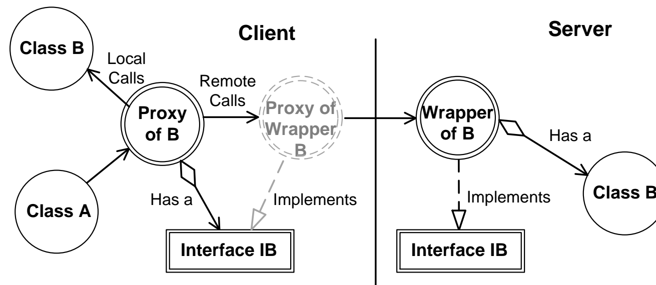

Fig. 6 Splitting a proxy into two parts.

To split a class, the refactoring engine takes as input its name and then either a set of fields or methods to move to the cloud. If the refactoring input is specified by means of fields, the selected fields and the methods accessing them are moved to the cloud. If the refactoring input is specified in terms of methods, the selected methods and the fields accessed by them are moved to the cloud. Figure 6 shows how the refactoring engine splits class `B` to redirect all the invocations to class `B` to the cloud-based service interface `IB` and the local object `LB`. Figure 7 shows an automatically generated proxy class, which redirects all the invocations to class `B` to the cloud-based service interface `IB` and the local object `LB`.

3.3.2 Handling Service Faults

Whether some functionality is accessed locally or remotely across the network should not change the application's functionality if not for the presence of partial failure. Unlike in a centralized application, components of cloud-based

```
public class B {
  private IB proxy;
  private LB local;

  public B() {
    proxy = (IB) getService(IB.class);
    local  = new LB();
  }
```

```
  /* Re−targeted methods */
  public String foo(int i1, int i2) {
    return proxy.foo(i1, i2);
  }

  /* Remaining methods */
  public void bar() { local.bar() }
}
```

Fig. 7 Generating a proxy class.

services can fail independently, making such failures difficult to diagnose and handle. Such failures must be handled effectively not only to ensure the overall application utility and safety, but to preserve the semantics of the original centralized applications. Thus, any refactoring technique that separates any functionality to be accessed remotely must take the issue of remote failures into consideration. In our approach, the *Extract Service* refactoring automatically adds well-known fault tolerance strategies configured through a domain-specific language.

Because it would be impossible to handle all possible errors, our refactoring approach focuses on well-known strategies for handling common faults, such as network volatility, service outages, and internal service errors. The generated fault tolerance functionality includes both detection and handling. A fault can be detected via timeout mechanism, exception handling, or runtime execution monitoring. Then, the detected faults should be properly handled to keep continuing the required functionality. Figure 8 shows these fault-handling procedures. First, a service administrator needs to provide fault tolerance descriptions written in Fault-Tolerance Description Language (FTDL) [8, 14], a domain-specific language we have developed earlier for expressing fault handling strategies. These descriptions parameterize a fault handling component that detects the specified faults and then counteracts their effect by executing the specified handling strategies. The refactoring engine inserts all the required fault handling code into proxy classes.

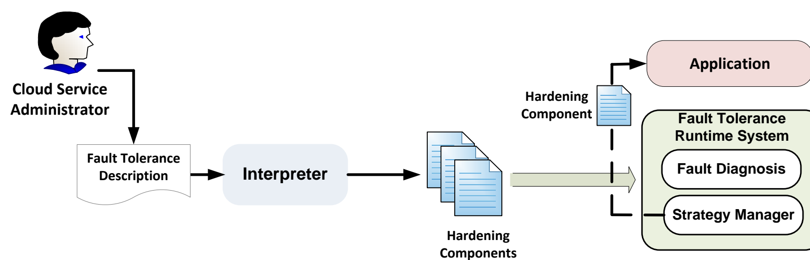

Fig. 8 Overview of fault handling.

```
<ftdl>
  <service uri=[remote address] method=[method name]/>
  <condition>
    <timeout>[0−9]∗</timeout>
    <exception>[exception types]</exception>
  </condition>
  <strategy>
    <!− attributions for the retry strategy −−>
    <retry numRetries=[0−9]∗ backoffInterval=[0−9]∗
        backoffType=["exponential" | "linear"] />

    <!− attributions for the sequential strategy −−>
    <sequential numRetries=[0−9]∗ backoffInterval=[0−9]∗
        backoffType=["exponential" | "linear"] >
      <service uri=[http://remote] />
        ... ...
    </sequential>

    <!− attributions for the user defined strategy −−>
    <defined>
      <handling name=[fault handling name] />
      ... ...
    </defined>
  </strategy>
</ftdl>
```

Fig. 9 FTDL constructs.

Fault Tolerance Description Language

One of the key novelties of our refactoring approach is using a domain-specific language to configure a refactoring engine to synthesize fault tolerance functionality. In our previous work [8,14], we explored how remote services can be made resilient against failures using domain-specific languages—*Hardening Policy Language (HPL)* [14] and *Fault Tolerance Description Language (FTDL)* [8]. In this work, we combined the features of these two languages—language constructs from FTDL and a runtime system from HPL— to create a refactoring transformation that automatically adds fault tolerance functionality. Figure 9 shows how FTDL is used by the refactoring infrastructure. The FTDL design has striven to combine expressiveness and ease of use. The specific design goals of FTDL have included: *Expressiveness*—a programmer should be able to express any kind of fault easily, with the resulting code being easy to understand, maintain, and evolve; *Extensibility*—it should be possible to integrate existing fault tolerance strategies with FTDL strategies; *Platform Independence*—FTDL strategies should be service platform independent, with the same strategy capable to counteract a fault raised by any service implementation.

Fault Tolerance Strategies

Traditionally, fault handling functionality in services computing tends to follow well-defined patterns. To render services reliable, representative approaches replicate SOAP web services [6,9,18], introduce transactional processing [19], and add fault handling code to server and client sides of a service-oriented application [27,33].

Our approach focuses on client-side fault handling for the faults caused by network volatility, service outages, and internal service errors. The first step in handling a fault is detecting it. The fault conditions can be detected via timeout mechanism, exception handling, or runtime execution monitoring. In the context of service-oriented applications, the following fault tolerance strategies are used quite commonly:

Retry The strategy that is arguably employed most widely is **Retry**, which, for a given number of times, reattempts to invoke a service in response to a failure.

Sequential Another common strategy is **Sequential**, which is also known as *passive* replication. This strategy iterates through different endpoints of a service when encountering a failure. For instance, when experiencing a timeout in response to invoking the service endpoint at `a.com/foo`, `b.com/foo` can be invoked next. This strategy, thus, increases the probability that some invocation will finally succeed. The term *passive* replication refers to the fact that this strategy does not kick in until a failure occurs.

Parallel An example of a more complex strategy is **Parallel**, which *actively* replicates a service, to invoke endpoints concurrently as a mechanism to counteract potential service unavailability. For example, both endpoints `a.com/foo` and `b.com/foo` would be invoked simultaneously. As a form of speculative parallel execution, this strategy proceeds with the first successfully executed request.

Composite Because a single strategy may not be sufficient, software designers often combine multiple strategies. For example, all the heretofore described strategies can be combined into composite strategies.

Generating Fault Handling Code

Figure 10 shows the fault handling functionality in a generated proxy class. Specifically, this proxy handles all the raised exceptions by passing them to method `notify()` in class `FaultHandler`. The fault handler is our light-weight fault-handling runtime that can execute fault-handling strategies. The runtime can execute both the standard fault tolerance strategies as well as the combinations of thereof. The standard strategies include retry, sequential, and parallel. These strategies can be combined in arbitrary ways into composite strategies

```
public class B {
   ...

   /* Fault handling code */
   public String foo(int i1, int i2) {
      try {
         return proxy.foo(i1, i2);
      } catch(CloudServiceException e) {
         return FaultHandler.notify(new Fault(...));
      }
   }
}
```

```
public class MyFaultHandling implements FaultListener {
   public Object faultNotified(Object service, Method m, Object[] params) {
         //retry the failed service invocation
   }
}
```

Fig. 10 Automatically generated fault handling code.

by writing a simple FTDL script. To specialize fault handling even further, one can implement any required fault-handling strategy by implementing interface `FaultListener`. The fault tolerance strategies can be reused across applications and can serve as building blocks for custom strategies.

The lightweight runtime system depicted in Figure 11 consists of a fault diagnosis module and a strategy manager. The fault diagnosis module catches raised exceptions or failures. The strategy manager associates exceptions with fault tolerance strategies. In response to detecting an exception, the manager initiates the handling strategy as configured by a given FTDL script. A strategy implementation is simply a sequence of corrective actions whose execution counteracts the effect of experiencing the fault. These actions are implemented as part of a library. In our prior work, we have demonstrated the effectiveness of this approach to improve the reliability of OSGi-based systems [14,15].

3.4 Cloud Refactoring—2) *Adapt Service Interface*

Cloud-based services expose their functionality through a set of public interfaces. It is also common that the same business functionality is offered by more than one service provider. For various business and technical reasons, an application may need to choose between multiple service providers for the same functionality. For example, multiple services may need to be consulted to check whether the information they provide is consistent. Multiple service implementation can also be used for fault-tolerance purposes.

Services providing equivalent functionality are likely to have different service interfaces. One option is to treat the invocation of different equivalent services as unrelated. This way, the client code required to invoke the ser-

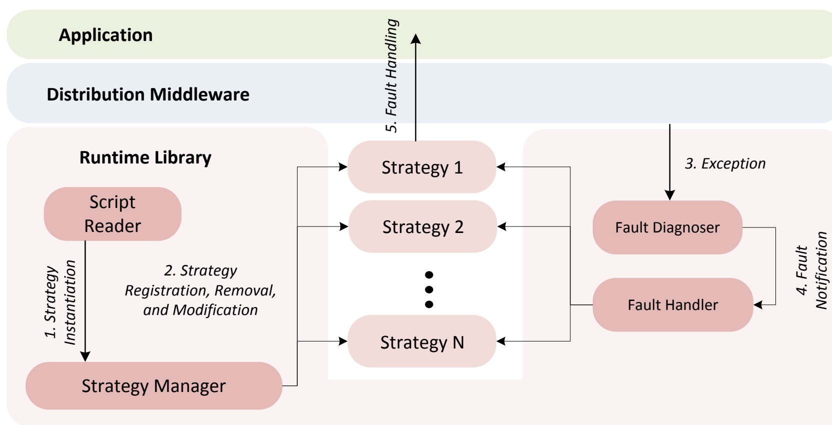

Fig. 11 Fault tolerant runtime system.

vices is replicated for each service. Another option is to systematically adapt one service's client-side interface bindings for another service interface. This adaptation is automated by means of the *Adapt Service Interface* refactoring.

The *Adapt Service Interface* refactoring automates the transformations required to apply the adapter pattern. Figure 12 shows how one service's client bindings can be adapted to use another service. As the first step, a programmer should specify the differences between the original and adapted service interfaces. That is, the programmer uses our refactoring browser to map interface method names to each other. Based on this method name mapping, the refactoring engine generates a skeletal implementation of the required adapter. The programmer can then fill in this skeletal implementation with the adaptation logic. For example, parameters can be simply reordered, missing parameters provided, and extra parameters omitted.

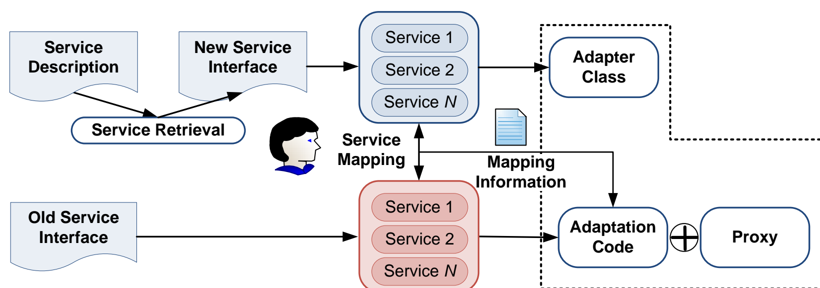

Fig. 12 Procedure of service adaptation.

As a specific example, consider switching the remote service invocations of interface `IB` described in Section 3.3 to interface `NewIB`:

```
public interface NewIB {
  String newFoo(int, int, int);
}
```

To switch services, our approach requires that the programmer provides the original and adapted service interfaces. If the adapted service interface is not available locally, the refactoring engine can automatically create one from a WSDL document. Because most web services describe their operations as a WSDL document, a Java interface describing the operations can be retrieved.

As mentioned above, programmers parameterize the refactoring engine by mapping to each other the original and adapted service interfaces. Figure 13 shows how the adaptation code switches the old service invocations to another service's implementation. Figure 14 shows the automatically generated adapter class. In this example, method `foo()` is being redirected to method `newFoo()`. The refactoring engine generates an adapter class `AdapterB` which is a singleton. If the adapted service methods differ in terms of their parameter numbers or types, the programmer needs to write code to adapt the parameters and/or return value. This part of the approach is manual, as parameter adaptation is highly application-specific and thus cannot be automated.

3.5 Implementing *Cloud Refactoring*

Figure 15 shows the main components of the refactoring tool, which were developed using several state-of-the-art software tools and libraries such as Eclipse plug-ins, OSGi, and Soot Java analysis framework. The refactoring tool consists of three components—1) GUI, 2) recommendation engine, and 3)

```
public class B implements IB { // Proxy class
  public String foo (int i1, int i2) {
    try {
      if(AdapterB.v().isAvailable()) { // Redirected service invocation
        return AdapterB.v().foo(i1, i2);
      }
      else { // Original service invocation
        return rService.foo(i1, i2);
      }
    } catch(CloudServiceException e) {
      return FaultHandler.notify(new Fault(...));
    }
  }
}
```

Fig. 13 Automatically generated proxy class for service adaptation.

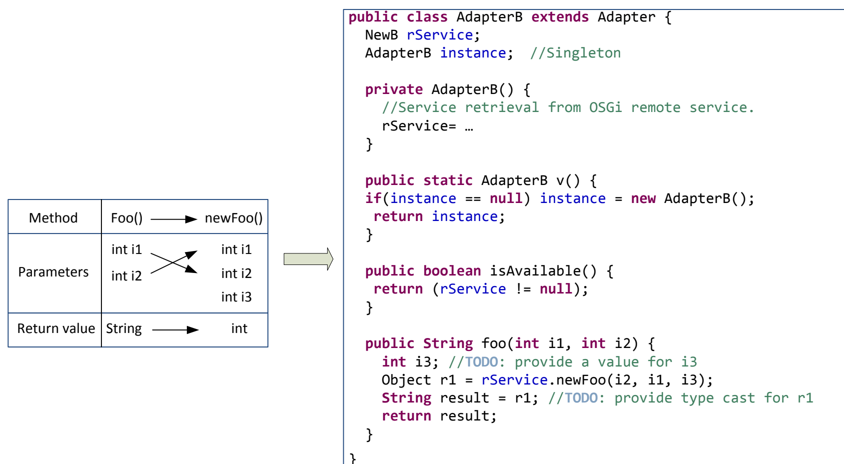

Fig. 14 Generating an adapter class from interface differences.

refactoring engine. The GUI part was implemented within the Eclipse-IDE's refactoring menus, so that a programmer can easily modify our refactoring and extend the refactored application within the Eclipse-IDE. The recommendation engine was implemented using a static program analysis framework—the Soot Java analysis framework, which manipulates and optimizes Java bytecode. The static analyzer and trace analyzer compute relationships between classes and the service recommender suggests service candidates via the editors and wizards of the eclipse IDE. Lastly, the refactoring engine has a series of code generators including proxy/wrapper generators for remote communications, interface generator for exposing services, adapter generator for switching a service interface.

3.6 Discussion

Next we discuss some of the advantages and limitations of using refactoring to transition applications to use cloud-computing resources.

3.6.1 Advantages

By automating the required program transformations, a refactoring is more likely to preserve the correctness of a modified program than when a programmer modifies the code by hand. Our cloud refactoring techniques also generate new code used by the modified code. For example, our refactoring engine generates several kinds of proxy classes used at the client. Generating code automatically also helps preserve program correctness. Our recommendation engine also informs the programmer about the parts of the centralized

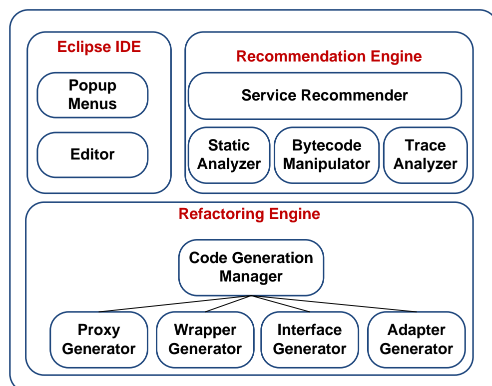

Fig. 15 Service refactoring tool's components.

program that can be moved to the cloud while minimizing the incurred performance overhead. Our runtime library features several fault tolerance mechanism implementations that can be used out of the box, thereby increasing the probability that the resulting application will be capable of handling partial failure.

3.6.2 Limitations

A refactoring may not be a proper approach for transforming all kinds of software applications to cloud-based services. First, transforming tightly coupled applications without incurring a significant performance overhead may require deep architectural changes that are not supported by our refactoring techniques. Ensuring good performance requires that remote communication be crude-grained and infrequent. In addition, cloud-based communication is inherently unidirectional: client talks to server but not vice versa. If the original application does not follow this communication pattern, its architecture needs to be changed before our refactoring techniques can be applied.

Second, to improve accuracy, the recommendation systems require special application-specific parameters. Based on the accuracy of the provided parameters, the recommendation system will show different results. Therefore, the programmer can experiment with different parameters to obtain a recommendation that is most aligned with the business requirements in place.

Third, our refactoring techniques do not make any provision for a situation when a newly extracted cloud service is used by multiple clients. Then the application logic would have to be modified accordingly to ensure a consistent and efficient access by multiple clients.

Lastly, our fault handling strategies cannot cover all the possible failure cases. In some scenarios, the programmer may need to implement some failure

handling strategy by hand, outside the framework provided by our refactoring infrastructure.

3.7 Motivation for Cloud Refactoring

An important question is what motivates enterprises to move software components to execute remotely in the cloud, thus necessitating the cloud refactoring techniques presented here. One motivation for leveraging cloud resources is to improve performance efficiency by processing large volumes of data in parallel (e.g., using Hadoop). However, this work is motivated by a different set of business cases for using cloud-based resources.

For many business applications, using remote cloud-based service is inevitable, even if the resulting performance efficiency would remain the same or even deteriorate. For example, some shared functionality may need to be shared between multiple clients (e.g., a local accounting component that has to be shared between multiple financial applications). As another example, some functionality may need to be moved into the cloud to take advantage of the cloud provider's data backup and replication services (e.g., a local database-dependent component can be moved to a cloud service along with its database files to guarantee long-term data integrity). Finally, companies may consolidate some replicated functionality and expose it as a cloud service to reduce the software maintenance efforts. Because services are exposed through a public service interface, the service's implementation can change at will without perturbing its clients, as long as the service interface remains fixed.

All these scenario represent a clear need for the cloud refactoring techniques discussed in this article, even though the refactored (i.e., cloud-based) versions of these applications are unlikely to show any increase in performance efficiency. However, unless the invocation of cloud services is in the critical performance paths of these applications, the overall performance impact of cloud refactoring is likely to remain insignificant. From the business perspective, migrating services to the cloud can reduce the overall software development costs and can even enable companies to break into new markets, as software-as-a-product (SaaP) can be easily reused and repurposed.

4 Evaluation

To evaluate the applicability of our *Cloud Refactoring* techniques, we applied them to two third-party applications to help transition them to cloud-based execution.

4.1 Micro Benchmark: Clustering-Based Recommendation

To evaluate the effectiveness of our recommendation approach, we applied the clustering-based recommendation to seven third-party applications and one our own application.

- Crypto [7]: Java implementation of the Unix crypt utility.
- Compress [7]: Java implementation of the Unix compression utility.
- Dictionary: our own application using the Lucene search engine library [3] to search definitions, find synonyms, and suggest corrects for misspelled words.
- JAligner[4]: an open source Java implementation for biological local pairwise sequence alignment.
- Barecue[5]: an open source Java library to create barcodes.
- JNotes: an open source Java management tool for memos, events and projects.
- PMD[6]: an open source Java program for potential problems like bugs, dead code, suboptimal code, overcomplicated expressions, and duplicate code.
- Weka [12]: an open source Java data mining software implementing a collection of machine learning algorithms.

Table 1 shows the benchmark results. Each column represents the number classes, the number of suggested remote services, the number of selected remote services, and the number of adaptable remote services. The first refactoring suggested possible remote services, and we manually selected appropriate remote services. The all suggested classes can be cloud-based services, however, we selected appropriate classes for the reason of the performance, call-by-reference, and meaning of features. Then, the last column shows how many refactored services can be adopted to the third-party services. We found few public Web services through public Web service repositories and manually investigated how the refactored services can be adapted to the new services.

Based on the micro benchmark result, we selected two applications to show refactoring procedures. In the next discussion, we show two case studies—JAligner and JNotes.

4.2 Case Study I—DNA Sequence Alignment—JAligner

As the first case study, we applied our refactoring techniques to JAligner—a third-party bioinformatics pairwise sequence alignment tool written to run as a standalone application on a single machine. JAligner takes as input two DNA sequences and computes their similarity metrics. We successfully refactored the application to use a fault tolerant cloud-based service and then switched the alignment functionality to use an equivalent third-party service.

[3] http://lucene.apache.org/java/docs/index.html
[4] http://jaligner.sourceforge.net
[5] http://barbecue.sourceforge.net
[6] http://pmd.sourceforge.net

Table 1 The experimental results.

Name	# classes	# suggested remote services	# selected remote services	# adaptable remote services	ratio
Crypto	5	4	4	3	80 %
Compress	7	3	3	0	43 %
Dictionary	7	3	3	1	43 %
JAligner	39	8	5	1	12.8 %
Barecue	56	1	1	0	1.7 %
JNotes	164	9	8	0	4.8 %
PMD	597	27	3	0	0.5 %
Weka	1243	14	7	0	0.5 %

4.2.1 Extracting Remote Service

While the clustering-based recommender suggested 8 classes as potential remote services, the profiling-based recommender suggested 4 classes:

– `Class Commons`: returns basic informations about the application.
– `Class SequenceParser`: parses the given DNA sequence and returns a `Sequence` object.
– `Class SmithWatermanGotoh`: aligns two DNA sequences.
– `Class Example`: returns example DNA sequences.

Although all the recommended classes can be refactored to cloud-based services, in this study we selected only one class—class `SmithWatermanGotoh`, which implements the main functionality of JAligner. For performance reasons, classes `Commons` and `SequenceParser` should not be transformed into cloud-based services. Moreover, because class `Example` forms its own cluster, it should not be moved to the cloud. As the first step, the refactoring engine generated interface `ISmithWatermanGotoh`, which is exposed by underlying middleware, and class `WrapperSmithWatermanGotoh`, a wrapper class of `SmithWatermanGotoh`, as well as some OSGi specific files. The following code snippet shows the automatically generated Java interface, which is exposed through remote OSGi services[7] After the refactoring engine finishes transforming all the code, the newly created service implementation can be deployed in the cloud and accessed remotely.

```
public interface ISmithWatermanGotoh {
  public Alignment align(
    Sequence s1, Sequence s2, Matrix m, float o, float e);
}
```

For the client execution, the refactoring engine re-targets client code to the cloud-based service. To that end, it generates a proxy class— `SmithWatermanGotoh` and OSGi specific files such as remote service configuration files. The generated interface is used for importing the exposed Web service. Through this

[7] The services are exposed through Apache CXF-DOSGi.

```
<ftdl>
  <service uri="http://192.168.0.1/SmithWatermanGotoh" method="align"/>
  <condition>
    <timeout>1000</timeout>
  </condition>
  <strategy>
    <sequential numRetries="10" backoffInterval="1000"
    backoffType="linear">
      <service uri="http://192.168.0.2/SmithWatermanGotoh" />
    </sequential>
  </strategy>
</ftdl>
```

Fig. 16 FTDL description to handle network volatility.

refactoring, the client is wrapped into a standard OSGi bundle and then uses the Smith-Waterman alignment service over the network. Figure 16 shows the FTDL description to handle network volatility.

4.2.2 Adapt Service Interface

We switched the extracted through refactoring remote service—`Smith-Waterman` alignment service—to a third-party Web service provided by European Bioinformatics Institute (EBI[8]). EBI provides several bioinformatics Web services, including local and global alignment services. We selected `Waterman-Eggert` algorithm and then adapted the client to use this service.

```
<portType name="water"> <operation name="runAndWaitFor">
  <input message="runAndWaitFor"/>
  <output message="runAndWaitForResponse"/>
</operation> </portType>

<complexType name="runAndWaitFor">  <sequence>
  <element name="aSequence" type="SeqInput"/>
  <element name="bSequence" type="SeqInput"/>
  <element name="gapopen" type="float"/>
  <element name="gapextend" type="float"/>
</sequence> </complexType>

<complexType name="SeqInput"> <sequence>
  <element name="direct_data" type="string"/>
  <element name="usa" type="string"/>
  <element name="format" type="string"/>
</sequence> </complexType>
```

Fig. 17 WSDL contract of the EBI's service.

[8] http://www.ebi.ac.uk/soaplab/

Figure 17 shows the WSDL document that describes the Web service specification. First, based on this WSDL document, we created a Java interface `Water` and its return/argument types such as class `RunAndWaitFor` and `RunAndWaitForResponse`. Figure 18 depicts the automatically generated new service interface and other necessary classes. Then, the refactoring generates the skeleton of adapter classes. The only manual part of this refactoring is for the programmer to write code that maps different parameters and return values (e.g., `class Sequence` and `class SeqInput`). As a result, when an unchanged existing client invokes the old service interface, the adapter intercepts the invocation and redirects it to the new service.

```
public interface SmithWaterMan {
    public RunAndWaitForResponse runAndWaitFor(RunAndWaitFor msg );
}
public class RunAndWaitFor {
  SeqInput aSequence, bSequence;
  float gapopen, gapextend;
}
public class SeqInput{
  String direct_data, usa, format;
}
```

Fig. 18 Generated interface and classes.

4.3 Case Study II—JNotes

As the second case study, we selected JNotes, our motivating example application. We refactored JNotes to used cloud-based services by means of *Extract Service*. As discussed in Section 2, we moved class `FileStorage` to the cloud by splitting it into two classes. In this example, we left resource saving functionality at the local machine and moved other functionality to the server. Figure 19 shows a proxy class that splits the original class into remote and local parts.

4.4 GE Portfolio Analysis Service

We demonstrate how our approach can benefit real companies that want to take advantage of cloud computing. We applied the *Extract Service* refactoring to *Portfolio Analysis Tool*, a real-world application developed by GE Global Research Center and GE Energy to analyze world economy scenarios and predicts how they may affect their customers' billing and costs. The application was developed using standard Web technologies that included the Spring framework and Java servlets.

```
public class FileStorage implements Storage {
  IFileStorage proxy;
  LFileStorage local;

  public FileStorage() {
    proxy = (IFileStorage) Activator.v().getService(IFileStorage.class);
    local = new LFileStorage();
  }

  /* Re-targeted methods */
  public void openEventsManager() {
    try {
      proxy.openEventsManager();
    } catch(CloudServiceException e) {
      return FaultHandler.notify(new Fault(...));
    }
  }

  public void openProjectManager() {
    try {
      proxy.openProjectManager();
    } catch(CloudServiceException e) {
      return FaultHandler.notify(new Fault(...));
    }
  }
  //more remote methods

  /* Remaining methods */
  public ResourcesList openResourcesList(Project prj) {
    return local.openResourcesList(prj);
  }

  public void storeResourcesList(ResourcesList rl, Project prj) {
    local.storeResourcesList(rl, prj);
  }
  //more local methods
}
```

Fig. 19 Generated proxy class.

In particular, Portfolio Analysis Tool 1) calculates billing and costs using hundreds of parameters that are maintained through a DBMS, 2) provides several complex financial components which are computation-intensive functionality, and 3) contains several common functionality that can be reused across multiple applications. Therefore, moving some key components of this application to the cloud would simplify maintenance—the infrastructure (i.e., a Web server, an application server, a DBMS, etc.) does not need to be maintained separately for each installation. Moreover, commonly accessed services can be effectively reused.

The recommendation tool of the *Extract Service* refactoring suggested several cloud-based services that can be extracted from the original application. Then, we used our refactoring engine to extract cloud-based services, with the

server components deployed in a private cloud environment and the client code transformed to access the cloud-based services remotely.

5 Related Work

The presented Cloud Refactoring techniques are related to program partitioning, software clustering, migrating application to services, and fault handling techniques. Next, we compare and contrast our techniques to the most relevant approaches in each of these categories.

One line of research has explored coarse grained program partitioning. The programmer, by means of a GUI, designates different parts of a centralized application, typically at a class or object granularity, to run on different network nodes. The resulting distribution specification then parameterizes a compiler-based tool that automatically rewrites the centralized application for distributed execution. To introduce distribution, a partitioning tool may need to both change the structure of the application (e.g., to introduce a proxy indirection) and add middleware functionality (e.g., to replace local calls with remote ones). In the Java world, recent automatic partitioning tools include Addistant [31], Pangaea [29], and J-Orchestra [32]. Addistant and J-Orchestra partition programs at a class granularity; Pangaea can partition at the individual object level. J-Orchestra addresses the challenges of partitioning programs safely in the presence of unmodifiable code that comes as part of their runtime systems.

Several prior research efforts aim at decomposing software systems into subsystems using clustering techniques [23]. The Bunch tool [23] uses a variety of clustering algorithms (e.g., hill climbing, genetic, etc.) to modularize existing systems; it extracts modules based on their dependence graph and calculates the resulting modularity quality. Clustering can be based on structural data (e.g., dependence graphs) and non-structural data (e.g., names, comments, behavior, etc.) [2]. Combining structural and non-structural clustering can improve the resulting modularity [1].

Much research has gone into decomposing the large, legacy systems into sub systems by assistance of the above clustering techniques. Such decomposition was performed for better understanding to the systems or maintenance for very large systems. However, in recent research, there was an attempt to adopt data mining techniques for partitioning into distributed applications or service oriented applications, including RuggedJ [22]. RuggedJ adopted a classification techniques which determines classes' types and locates them distributed nodes such as server/client.

In addition, our approach is related to migrating legacy systems toward objects [20] and services [5]. The module dependence graph has recently been shown to be effective at guiding the migration toward services [17]; loosely-coupled modules become service components. In addition, a model-based approach has been proposed to extract UML from legacy code and to use proxy wrappers as service interfaces [21].

Commonly used approaches to recover from failure include termination [28], restarting [30], micro-rebooting [4], and checkpoint-restart [13]. Although traditionally such approaches have been integrated with system design, our approach can expose them as a fault strategy configured through FTDL. Therefore our approach can increase the resiliency against faults even further.

6 Conclusion

In this article, we have presented *Cloud Refactoring*, a set of semantics preserving transformations that can help migrate a centralized application to using cloud-based services. We realized *Cloud Refactoring* in the context of a modern IDE, enhancing its refactoring engine. *Cloud Refactoring* comprises two main refactoring techniques: *Extract Service* and *Adapt Service Interface*. The *Extract Service* refactoring renders a portion of a centralized application's functionality as a remote cloud-based service, rewriting the client code and enhancing it with the required fault-tolerance strategies. The *Adapt Service Interface* refactoring automates the transformations needed to switch a service client to use an alternate, equivalent cloud-based service. We have evaluated *Cloud Refactoring* by transforming third-party applications to cloud-based services, including an application used by General Electric. Our experiences indicate that refactoring can become a valuable tool in the toolset of software developers charged with the challenges of migrating applications to take advantage of cloud-based resources.

Acknowledgements GE Global Research has provided realistic cloud migration scenarios that motivated some of the refactoring techniques discussed in the article. This research is supported by the National Science Foundation through the Grant CCF-1116565.

References

1. B. Andreopoulos, A. An, V. Tzerpos, and X. Wang. Clustering large software systems at multiple layers. *Inf. Softw. Technol.*, 49(3):244–254, 2007.
2. P. Andritsos and V. Tzerpos. Information-theoretic software clustering. *IEEE Trans. Softw. Eng.*, 31(2):150–165, 2005.
3. R. Buyya, C. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6):599–616, 2009.
4. G. Candea and A. Fox. Recursive restartability: turning the reboot sledgehammer into a scalpel. In *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, pages 125–130, May 2001.
5. G. Canfora, A. Fasolino, G. Frattolillo, and P. Tramontana. A wrapping approach for migrating legacy system interactive functionalities to service oriented architectures. *Journal of Systems and Software*, 81(4):463–480, 2008.
6. V. Dialani, S. Miles, L. Moreau, D. De Roure, and M. Luck. Transparent fault tolerance for web services based architectures. *Euro-Par 2002 Parallel Processing*, pages 107–201, 2002.
7. S. Dieckmann and U. Hölzle. A study of the allocation behavior of the specjvm98 Java benchmark. In *ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 92–115, London, UK, 1999. Springer-Verlag.

8. J. Edstrom and E. Tilevich. Reusable and extensible fault tolerance for restful applications. In *The 11th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, pages 737–744, 2012.
9. C. Fang, D. Liang, F. Lin, and C. Lin. Fault tolerant web services. *Journal of Systems Architecture*, 53(1):21–38, 2007.
10. M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, MA, USA, 1999.
11. J.-W. E. Group. JSR-224 Java api for xml-based Web services 2.0. Technical report, Java Community Process.
12. M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *SIGKDD Explor. Newsl.*, 11(1):10–18, 2009.
13. S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, 2005.
14. Y.-W. Kwon and E. Tilevich. A declarative approach to hardening services against QoS vulnerabilities. In *Proceedings of the 2011 IEEE International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA)*, 2011.
15. Y.-W. Kwon, E. Tilevich, and T. Apiwattanapong. DR-OSGi: Hardening distributed components with network volatility resiliency. In *Proceedings of the ACM/IFIP/USENIX* 10th *International Middleware Conference (Middleware)*, 2009.
16. V. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet Physics Doklady*, volume 10, pages 707–710, 1966.
17. S. Li and L. Tahvildari. A service-oriented componentization framework for Java software systems. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 115–124, Washington, DC, USA, 2006. IEEE Computer Society.
18. D. Liang, C. Fang, and C. Chen. FT-SOAP: A fault-tolerant web service. In *Tenth Asia-Pacific Software Engineering Conference, Chiang Mai, Thailand*, 2003.
19. A. Liu, Q. Li, L. Huang, and M. Xiao. FACTS: A framework for fault-tolerant composition of transactional web services. *Services Computing, IEEE Transactions on*, 3(1):46–59, 2010.
20. G. A. D. Lucca, A. R. Fasolino, P. Guerra, and S. Petruzzelli. Migrating legacy systems towards object-oriented platforms. In *ICSM '97: Proceedings of the International Conference on Software Maintenance*, pages 122–129, 1997.
21. A. Marchetto and F. Ricca. From objects to services: toward a stepwise migration approach for Java applications. *Int. J. Softw. Tools Technol. Transf.*, 11(6):427–440, 2009.
22. P. McGachey, A. L. Hosking, and J. E. B. Moss. Pervasive load-time transformation for transparently distributed Java. *Electron. Notes Theor. Comput. Sci.*, 253:47–64, December 2009.
23. B. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208, 2006.
24. OSGi Alliance. RFP 133 cloud computing. Technical report, 2010.
25. OSGi Alliance. OSGi release 4.3 specification. Specification, 2011.
26. Paremus Ldt. The Paremus service fabric - a technical overview, 2008.
27. G. T. Santos, L. C. Lung, and C. Montez. FTWeb: A fault tolerant infrastructure for web services. *Enterprise Distributed Object Computing Conference, IEEE International*, 0:95–105, 2005.
28. S. Sidiroglou, G. Giovanidis, and A. D. Keromytis. A dynamic mechanism for recovering from buffer overflow attacks. In *In Proceedings of the 8 th Information Security Conference (ISC*, pages 1–15. Springer-Verlag, 2005.
29. A. Spiegel. *Automatic Distribution of Object Oriented Programs.* PhD thesis, FU Berlin, FB Mathematik und Informatik, 2002.
30. M. Sullivan and R. Chillarege. Software defects and their impact on system availability-a study of field failures in operating systems. In *Fault-Tolerant Computing, 1991. FTCS-21. Digest of Papers., Twenty-First International Symposium*, pages 2–9, Jun 1991.
31. M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A Bytecode Translator for Distributed Execution of "Legacy" Java Software. In *European Conference on Object-Oriented Programming (ECOOP)*, 2001.

32. E. Tilevich and Y. Smaragdakis. J-Orchestra: Automatic Java application partitioning. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, ECOOP '02, 2002.
33. Z. Zheng and M. Lyu. Optimal fault tolerance strategy selection for web services. *International Journal of Web Services Research*, 7(4):21–40, 2010.