Automated Refactoring of Objects for Application Partitioning

Vikram Jamwal and Sridhar Iyer
IIT Bombay, INDIA
(vikram, sri)@it.iitb.ac.in

Abstract

Distributed infrastructures are becoming more and more diverse in nature. An application may often need to be redeployed in various scenarios. Ideally, given an application designed for one deployment scenario, one should be able to generate an application version for a new scenario through an automated refactoring process. For this to happen, one of the principal requirements is that application components should be amenable to partitioning.

To achieve this: (i) We use a structurally simple and slightly modified model of object called Breakable Object (BoB)*, for structuring such applications. BoB can be treated as an object which is designed to be well disposed towards automated refactoring. We also devise a programming model for BoBs in Java called Java$_{BoB}$. (ii) We provide algorithms for automated refactoring of a Java$_{BoB}$-based program.*

1. Introduction

Distributed systems have grown from having nodes with uniform computing and communication capabilities, to having nodes with widely varying capabilities. An application designed for one scenario may not be amenable to direct redeployment in another scenario.

Consider an e-mail application. One may access email in a variety of scenarios, such as using a desktop on a LAN or using a PDA on a 3G network. We may also want different versions of the application to cater to different modes of operation, such as (i) *Online mode* in which the messages are kept on a server and the client manipulates them remotely using an appropriate interface, (ii) *Offline mode* in which the client fetches the messages to the local machine and the messages are deleted from the server, and (iii) *Disconnected mode* in which the client fetches the messages to the local machine and the messages are also retained at the server; the client and server periodically synchronize to appropriately reflect the actions taken on any message.

Although the functionality of the application remains same in all the three modes, the amount of functionality that is implemented at the server or the client varies, depending upon the mode. *Ideally, given an application designed for one scenario, one should be able to automate refactoring and generate the application for a new scenario*.

Our experience with e-mail applications (icemail, jwma, pooka, popmail [1]) has shown us that refactoring an existing application to operate in a different scenarios is extremely difficult. Besides dealing with (i) *environmental heterogeneity issues* [10] - differences in environments encountered due to hardware and software constrains on target environment, and (ii) *distribution issues* [14] - distributing the application components across nodes and making them work as distributed components, one issue of primary concern is *functionality partitioning* of the application.

Functionality partitioning implies - apportioning application functionality into deployment specific sub-sets of objects. However, this is hard to achieve in practice as we cannot draw clean lines of functionality separation through an application. *Some functionality may span across multiple classes, or a single class may include parts of multiple functionality*. For example, in the email application , the `Store` class may be on the server in all the three modes. However, the `Folder` class may have its functionality partitioned between client and server for online and disconnected modes. For the offline mode, it may be only on client. Hence it is required that we refactor the `Folder` class for transforming the application from one scenario into another.

To enable automatic refactoring of an application component, such as the `Folder` class, we need to first reformulate it in terms of a component that can be easily partitioned into sub-components.

We use a structurally simple and slightly modified model of object called *Breakable Object - BoB*, for structuring such applications. A BoB is an object that can be readily broken into sub-objects. It has an added construct - *together*, to denote the methods that are designated inseparable by the designer of the BoB. The sub-objects can replace the original BoB in a program, without affecting the original semantics (operational) of the program. We define BoB in section 2 and give an overview of the BoB programming method-

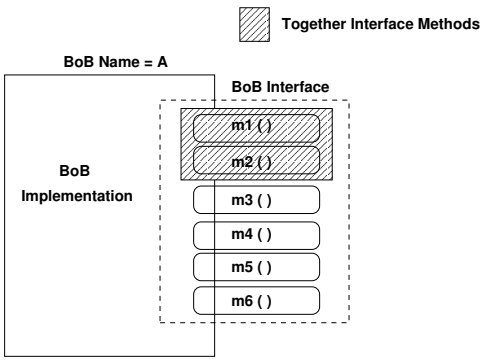

Figure 1. BoB

ology by using the distributed e-mail application outlined above as an illustrative example.

Once an application has been developed using BoBs, we can more easily refactor the application for various deployment scenarios by appropriate scenario-specific splitting of the BoBs. We claim that a BoB-based application design greatly facilitates automated partitioning of the application for various redeployment scenarios.

This paper presents:

1. **Programming model for BoB:** We provide a BoB implementation in Java, called Java$_{BoB}$, in section 3. The features of a Java$_{BoB}$ class are subset to those of a Java class except for one new language/preprocessor construct, namely, `together`.
2. **Alogrithms for Splitting Engine:** The splitting engine performs compile-time refactoring of a BoB-based program for various configurations. It takes as input the BoB-based program along with a *split-configuration* (which provides sub-object specifications for each partition) and produces a scenario-specific Java program as the output. Section 4 describes the details of these algorithms.

In Section 5 we discuss the related work on automated application partitioning and object refactorings. Section 6 provides the conclusions and future directions of our work.

2. BoB Based Programming

A BoB is an entity (class/object/component) in a program that can be readily split into sub entities. Sub entities should be so formed that they can replace the BoB while retaining the semantics(operational) of the original program.

2.1. BoB Features

1. BoB Interface: It defines the services provided by the BoB. It has the following salient features:

- The set of `public` methods exported by the BoB provide the interface.
- There are no `public` attributes (fields) in a BoB interface. Access to attributes, if needed, is provided through *get()* and *set()* methods.
- *Together methods*: Some of the methods can be grouped together by the designer of a BoB. These interface methods cannot be separated in the course of a split. We introduce a language/ preprocessor construct `together` to specify such methods.

2. BoB Implementation: A BoB in a class-based programming language is implemented using a `Class` in that language. There are two features that BoBs do not support: viz., BoBs *do not inherit* and BoBs are *not active* objects. We discuss this in the later sections.

Figure 1 depicts a BoB for a class-based programming language like C++, Java, etc. It consists of name of the BoB class - `A` and an interface consisting of public methods `m1, m2, m3, m4, m5, and m6` exported by the class. Methods `m1` and `m2` are together. The specification is: `together{m1, m2};`

2.2. BoB Splitting Specifications

BoBs are split on the basis of their *interface methods* and an externally specified *split-configuration* file.

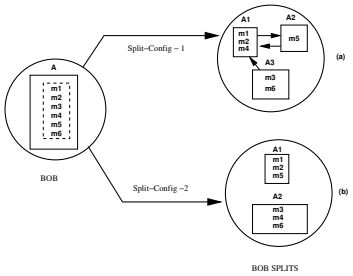

Figure 2. BoB splits generation

For example, consider the BoB `A` (Figure 1). It has:

Interface $\mathtt{I_A} = \{\mathtt{together(m1, m2)}, \mathtt{m3}, \mathtt{m4}, \mathtt{m5}, \mathtt{m6}\}$

Given split-configuration is:

$\mathtt{Split_{cfg-1}A} = \{\mathtt{s_1} =< \mathtt{m1, m2, m4} >, \mathtt{s_2} =< \mathtt{m5} >, \mathtt{s_3} =< \mathtt{m3, m6} >\}$

After the splitting process (Figure 2a), `A` is split into sub class-set $\mathtt{A_1}, \mathtt{A_2}, \mathtt{A_3}$ supporting the interfaces:

$\mathtt{I_A^1} = \{\mathtt{together(m1, m2)}, \mathtt{m4}\}$,

$\mathtt{I_A^2} = \{\mathtt{m5}\}$, and

$\mathtt{I_A^3} = \{\mathtt{m3}, \mathtt{m6}\}$ respectively.

The original BoB `A` is replaced by these split-classes $(\mathtt{A_1}, \mathtt{A_2}, \mathtt{A_3})$ in the program. This program transformation is denoted as: $\mathtt{A} \xrightarrow[\mathtt{Split_{cfg}}]{\otimes} \mathtt{A_1} + \mathtt{A_2} + \mathtt{A_3}$

In Figure 2b, we use another split-configuration to achieve a different sub class-set A_1, A_2 for the original BoB A.

Keeping the specification of *split-configurations* external to a BoB helps to separate the splitting and implementation concerns.

2.3. BoB Methodology

We explain here briefly the stages involved in a BoB based programming process (Figure 3):

1. Program Design and Implementation (Steps 1-3, Figure 3): The program designer proceeds in a manner similar to object-oriented analysis and design, and uses requirement guidelines to divide the application functionality into a set of objects and BoBs.

2. Splitting and Reorganization (Steps 4-6, Figure 3): The *split-configurations* for a given scenario are specified for all the relevant BoBs. A splitting engine prepares *class-definitions* for the new set of BoB-splits that will replace the original BoB in the program. The rest of program is reorganized to convert the references to original BoBs into references to their splits.

3. Redeployment (Steps 7-8, Figure 3): The application components are redistributed across the various nodes of network by a distribution engine (based on *deployment-configuration specifications*).

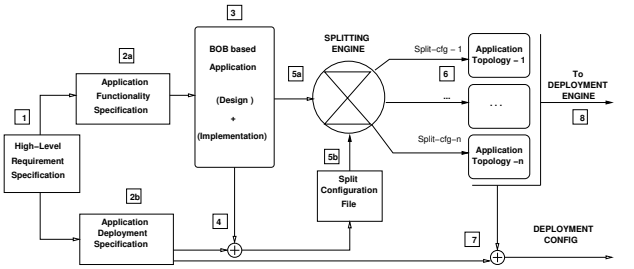

Figure 3. BoB programming process

For the e-mail application motivated in the introduction, we identified two objects for BoB implementation: `BoBFolder`, which supports a mechanism to create a cache of sub-folders and message-infos, and contains an object for IMAP protocol handling; and `BoBMessage`, which carries actual e-mail messages that can be single or multipart. Figure 4 shows the refactoring of the `BoBFolder` for the *disconnected* mode of the given e-mail application.

In this paper we focus on the splitting engine. Given a BoB-based program and a split-configuration file, we show how the splitting engine generates the splits corresponding to each BoB in the program. The distribution engine and deployment aspects are beyond the scope of this paper.

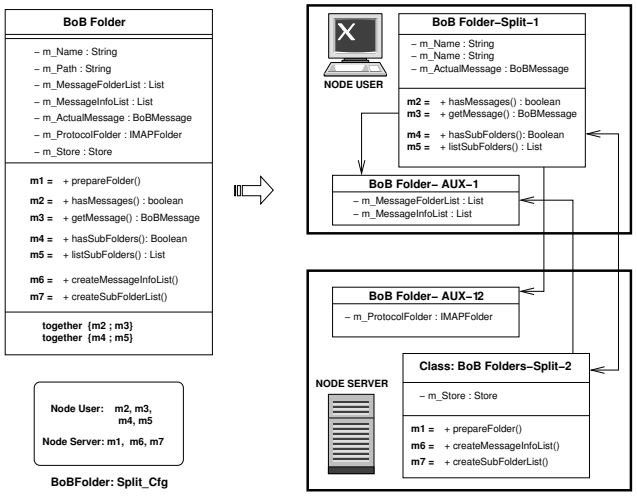

Figure 4. Splitting and redeployment for `BoBFolder`

3. BoB Model for JAVA

Our programming model for **BoBs**, called **Java**$_{BoB}$ is based on the object oriented language Java. Though simple, the programming model is functionally comprehensive. Table 1 presents the status of various constructs used in Java$_{BoB}$.

3.1. Java$_{BoB}$

BoB class resembles a Java class except the restrictions that are placed on certain features. There is an additional programming language construct in Java$_{BoB}$, viz., *together*, which is used to specify inseparable methods. A preprocessor (refer splitting engine, section 4) generates Java class definition files (.java files) from the BoB class definition files (.bob files). Thus Java BoBs can be used with existing Java Virtual Machines(JVMs) without any modification to the latter. Figure 5 provides the schematics of a Java$_{BoB}$ class. A more formal description of Java$_{BoB}$ class is available in [6].

Some of the constructs in present Java language have been disallowed for the sake of simplicity[1]. The Java$_{BoB}$ differs from Java as defined in language reference [4] principally in the following ways:

`public` **fields:** No public fields are exported by the BoB. The designer provides *getter* and *setter* methods for accessing the fields if required.

1 However, this is not a binding on the model. If required and if appropriate solutions for refactoring are available, the model can be extended to incorporate *new* or many of these diallowed features.

Java Construct	Status in Java$_{BoB}$
Class declarations	
`public`	Allowed
`abstract`	Not Allowed
`final`	Allowed (default)
`class` *Name Of Class*	Allowed
`extends` *Super*	Not-Allowed
`implements` *Interface*	Allowed
Variable (Field) declarations	
`public`	Not Allowed
`private`	Allowed
`protected`	Not Allowed
`package`	Not Allowed
`static`	Allowed
`final`	Allowed
`transient`	Not Allowed
`volatile`	Allowed
Method declarations	
`public`	Allowed
`private`	Allowed
`protected`	Not-Allowed
`package`	Not-Allowed
`static`	Allowed
`abstract`	Not-Allowed
`final`	Allowed (default)
`native`	Not Allowed
`synchronized`	Allowed
Miscellaneous features	
Constructors	Allowed
Exceptions	Allowed
Threads	Not Allowed
Nested class/Inner class	Not-Allowed

Table 1. Constructs for BoB Model - Java$_{BoB}$

Inheritance: The class that needs to be split cannot be a derived class. This means that all the members that form a part of the class are specified within it. The only class that a BoB class implicitly inherits from is `object`, the root Java object class. Also each BoB is a *final* class.

Interface inheritance is allowed with a restriction that all the methods of an interface are designated *together*. This is done to maintain interface based polymorphic refereces to the objects even after splitting.

We propose the use of aggregation and delegation, as discussed in [7], as the preferable composition mechanisms for BoBs.

Threading: BoB is not an active object [2] [8], that is, it cannot run as separate threads on its own. Also, since we do not allow inheritance, Java BoB cannot inherit from the `Thread` class and hence cannot be run as a separate thread in a program. However, BoB methods can be accessed by different threads in a program and we can specify the methods as `synchronized`. The responsibility of ensuring thread safety lies with the designer of the BoB.

The next section provides the details of splitting engine.

```
<public> final BoBClass A{

  private static field declarations
  private field declarations
 ------------------------------------
 together{ (';' separted list
             methodName(List ArgumentTypes),
           (...),...
         }
 ------------------------------------
  constructor declarations
...
  private static method declarations
  private method declarations
...
 ------------------------------------
 public static  method declarations
 public method declarations
 ------------------------------------
...
}
```

Together methods specification

BoB Interface

Figure 5. A.bob

4. Splitting Engine Details

In this section, we present the algorithms used by the splitting engine (refer Figure 6). A program $\mathbb{P}$ comprising of a set of BoB classes and Java classes and a *split-configuration* file, forms the input. The transformed program $\mathbb{P}'$ containing only Java classes is the output of the splitting engine. Both the operations, viz., BoB splitting and client reorganization, are done at compile time. Prior to splitting, a *validator* checks whether the splittings specified in the configuration file form valid splits on the specified BoBs.

4.1. Main Algorithm

The splitting engine takes a specified BoB class, splits it and performs client reorganizations. For the example shown in Figure 6, it first takes the BoB class `A.bob` and based on the specifications in the `split.conf` file and the internal dependency graph of `A.bob`, it generates Java class definitions for the new splits - `A1.java, A2.java, A3.java`.

Input: BoB and Java class files for $\mathbb{P}$, Split-Config file
Output: Java class files for $\mathbb{P}'$
foreach *BoB class* $A \in$ *Split-Config* **do**
  if *(valid* $A \wedge$ *valid* $split_{cfg} A$*)* **then**
    Prepare IDG for A ;
    Split A ;
    forall *classes in* P *that refer* A **do**
      Reorganize ;
    end
    Delete A;
  end
  else
    report 'invalid splits'
  end
end

Algorithm 1: Splitting engine main algorithm

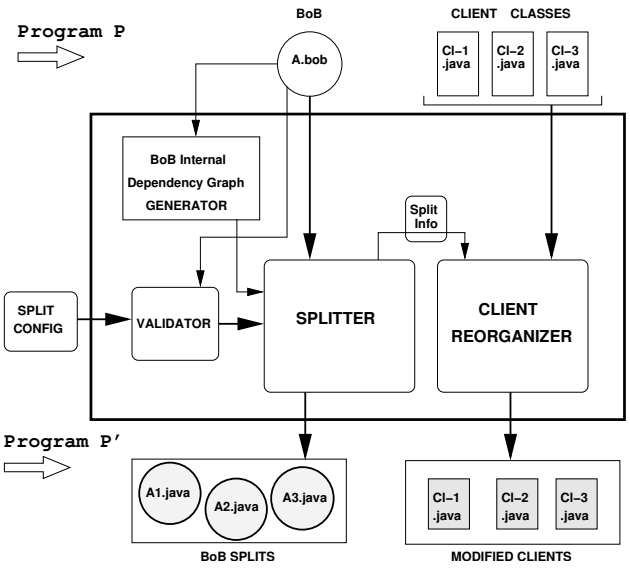

Figure 6. Mechanisms of Splitting Engine

It then reorganizes all the client classes, `Cl-1.java`, `Cl-2.java`, and `Cl-3.java`, that referred to `A.bob` to now refer to `A1.java`, `A2.java`, `A3.java`. This process is then repeated for all the specified BoB classes. This algorithm is described in Algorithm 1.

4.2. Split Configuration File

It specifies the number and the form of splits. The format is:

```
Number of BoBs = n;
BoB-Name-1 {
  No of splits = k;
  Split 1 = (';' separated list of methods
                  specified as -
            MethodName (ArgumentTypes) )
   . . .
  Split k = . . .
} . . .
BoB-Name-n {...}
```

A valid configuration file satisfies the following properties:

1. Only public methods are specified.
2. Every public method in each BoB has to be specified as part of some split.
3. A method cannot belong to more than one split.
4. Together methods cannot be specified as belonging to different splits.

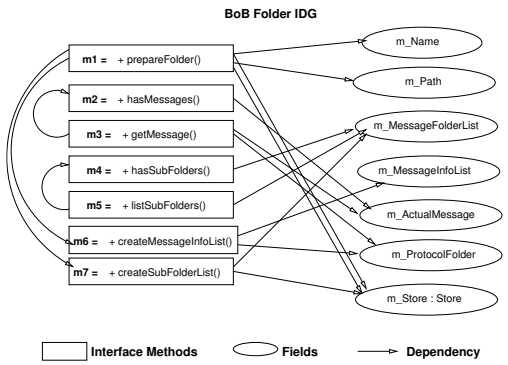

Figure 7. IDG for BoBFolder

4.3. BoB Internal Dependency Graph (IDG)

BoB internal dependency graph is used for understanding various dependencies between fields and methods, and among methods in a BoB class. The algorithm constructs a directed graph capturing these internal dependencies of a BoB. It has two types of nodes: field (F) nodes and method (M) nodes. We need to consider only those object and class references in a method, which form the fields of that class. Figure 7 presents the IDG for `BoBFolder`.

4.4. Algorithm for Splitting a BoBClass

Input to this algorithm is a valid BoB class file ($A.bob$) and its corresponding split-configuration file (A_{cfg}). The splitting algorithm (Algorithm 2) creates a $.java$ file for each specified split. Split class file constructions are done in the following three passes:

Pass 1: Writing specified methods into the splits This pass begins by including all the `import package` statements. It modifies the *Name of Class* and includes the class declaration constructs. These translations are done as indicated in the Table 2. It then includes all the methods that are specified in the configuration file into the respective split class files, referring to the Table 2 for appropriate translations of method declaration constructs. Next it includes all the methods of the original class, which fall in the call chain of an included method M.

As the final step in this pass, it changes the *access* specifiers of all the included methods that are not part of the split-specifications to `private`. This is done to maintain interface consistency.

Pass 2: Writing fields and constructors into the splits The algorithm notes all the fields referred by a method M in a split class. If a field is referred only by the methods of one split, i.e., if it is an *independent* field, it is included in that particular split. If it is a *shared field*, i.e., it is referred by the methods of more than one split, it is written into a sep-

Input:
- BoBClass File `A.bob`
- Split Configuration File A_{cfg}

Output: Split Class Files
`Asplit_1.java,...,Asplit_k.java &`
`Asplit_AUX.java`

forall *splits specified in* A_{cfg} **do**
  Create a class file (splitclass) ;
  Name it as: `ASplit _k.java`,where $k = split \quad number$;
  foreach *import statement* ($import_{stm}$) *in the original class A* **do**
    splitclass $\xleftarrow{write}$ $import_{stm}$
  end
  splitclass $\xleftarrow{write}$ `ASplit_k` + translated class constructs {refer Table 2}
  splitclass $\xleftarrow{write}$ ' { '
  foreach *method* m *in original class A* **do**
    if *method* m *specified for* this *split in the* A_{cfg} *file* **then**
      MethodInclude(m);
    end
  end
  foreach *method* m *in* splitclass **do**
    if *method* m *NOT specified for* this *split in the* A_{cfg} **then**
      if *access* $\neq$ `private` **then**
        Make (access = `private`)
      end
    end
  end
end
forall *splits specified in* A_{cfg} **do**
  foreach *method* m *in* splitclass **do**
    Refer BoB Internal Dependency Graph. Check all outgoing edges from the given method node m ;
    foreach *outgoing edge* **do**
      if *referred node is field* f **then**
        FieldInclude(f)
      end
    end
  end
  foreach *constructor* $A(exps)$ *in original class A* **do**
    ConstructorInclude($A(exps)$);
  end
  splitclass $\xleftarrow{write}$ ' } '
end
forall *splits specified in* A_{cfg} **do**
  foreach *method* m *in* splitclass **do**
    foreach *field reference* $ref.f$ *in* m **do**
      if *field* $f \notin$ this *split* **then**
        if *field* f *is static* **then**
          $f \xrightarrow{convert} AUX.(get/set)f$
        end
        else
          $f \xrightarrow{convert} refAUX.(get/set)f$
        end
      end
    end
  end
end

Algorithm 2: Generating BoB class splits

Procedure `MethodInclude(`*M*`)`

Input: Method Name M

if *M not already included* **then**
  splitclass $\xleftarrow{write}$ M ;
  Refer BoB IDG. Check all outgoing edges from method node M ;
  foreach *outgoing edge* **do**
    if *referred node is method* M **then**
      MethodInclude(M)

Procedure `FieldInclude(`F`)`

Input: Field Name F

if *F not already included* **then**
  if *F = constant field* **then**
    splitclass $\xleftarrow{write}$ F ;
  else if *F = variable field* **then**
    Refer BoB IDG. Check all incoming edges to field node F;
    if $\exists$ *incoming edges from methods in other splits* **then**
      AUXclass $\xleftarrow{write}$ F ;
      AUXclass $\xleftarrow{write}$ getter and setter for F;
      if *F is non static* **then**
        splitclass $\xleftarrow{write}$ $refAUX$;
    else
      splitclass $\xleftarrow{write}$ F ;

Procedure `ConstructorInclude(`*A (exps)*`)`

Input: Constructor Name $A(exps)$

forall *(f* $\in$ *original class)* $\wedge$ *(f referred in A(exps))* **do**
  if $f \notin$ *this split* **then**
    Declare f local in beginning of constructor body with same initial value as in original class ;

if *exist AUX class* **then**
  Include AUX in argument list $A(exps)$;
  Initialize $refAUX$ with passed value in $A(exps)$;

splitclass $\xleftarrow{write}$ $Asplit\_k(exps)$;

arate auxiliary (AUX) split class file. *Constant* fields are replicated into all the splits that use them.

We use only simple constructors for constructing BoBs. It is assumed that BoB constructors will not affect the state of objects external to that BoB. For each *constructor* specified in the original BoBClass, the algorithm includes a corresponding modified constructor in each split.

Pass 3: Transforming the shared field references In this pass, the algorithm iterates over each split class file and its method bodies to see if there are any class or object references to a shared field in the $AUX - Split$ file. Every such occurrence is transformed into a reference through $AUX - split$ class or an object of this class.

Handling Shared Fields Some fields of a BoB are shared between two or more splits. The splitting engine places all the shared fields as private fields of an *auxiliary* class. The access to these shared fields is provided with the help of *getters* and *setters*. At a later stage, the option is given to the user, to merge these shared variables with any of the split classes or keep them in separate node-specific auxiliary classes.

Figure 4 shows the splits performed on `BoBFolder`. It also shows the redeployments done on `BoBFolder` splits. The AUX class obtained after splitting is further split into $AUX1$ and $AUX2$ for the purpose of redeployment.

BoB Class Construct	Translation in splits
`public, final, class` *NameOfBoB*	Apply `public, final, class` respectively to all split class declarations Modified in the split classes as: `NameOfBoBSplit_1` `NameOfBoBSplit_2` `. . .` `NameOfBoBSplit_k`
`implements` *Interface*	Apply `implements` *Interface* to the corresponding split
BoB Field Construct	**Translation in respective split fields**
`private, static, final, volatile`	Apply `private, static, final, volatile` respectively
Type and *Name*	*Type* and *Name* remain same as before
BoB Method Construct	**Translation in respective split methods**
`public`	Change to `private` if split interface does not contain this method; otherwise, apply `public`
`private, static, final, synchronized, throws`	Apply `private, static, final, synchronized, throws` respectively
Type, Name, Arg List	*Type, Name, Arg List* remain same

Table 2. Construct Translations in Splits

4.5. Modification of the Client Program

All the classes that refer to a BoB class or object in a program, constitute *clients* with respect to the *server* BoB class. The clients can access these classes or objects in a number of *forms*. The algorithm considers the various scenarios and the corresponding modifications that are done in client codes. **Table 3** shows the various transformations that occur on client statements that refer to BoB classes or objects. We discuss them briefly below:

- **Variable declaration** (Transformation **T-1**): Variables for each split are declared.
- **Object creation** (Transformation **T-2**): Argument types for the constructor that is being invoked are noted. Split-objects are created by invoking the same signature constructor for all the split-classes.
- **Method invocation** (Transformations **T-3 and T-4**): The reference is changed to the split-class/object to which this method belongs and the corresponding method is invoked.
- **Variable assignment, argument passing, or aliasing** (Transformations **T-5 and T-6**): Wherever the split variable is being assigned a value, being passed as an argument or being aliased, it is replaced by corresponding assignment, method argument passing, and aliasing for each of the split variables.

Tfm	Code form in $\mathbb{P}$	Code translation in $\mathbb{P}'$
T-1	BoBTypeA x;	BoBTypeA_Split1 x_split1; … BoBTypeA_Splitk x_splitk; BoBTypeA_AUX x_aux;
T-2	`new` BoBTypeA ($exps$);	`new` BoBTypeA_Split1($exps$); … `new` BoBTypeA_Splitk($exps$); `new` BoBTypeA_AUX($exps$);
T-3	BoBClassA.m ($exps$)	BoBClassA_Split1.m($exps$) $\vee$ … $\vee$ BoBClassA_Splitk.m($exps$)
T-4	ref. BoBClassA/m ($exps$)	ref. BoBClassA_Split1/m($exps$) $\vee$ … $\vee$ ref. BoBClassA_Splitk/m($exps$)
T-5	BoBTypeA x = refBoBTypeA;	BoBTypeA_Split1 x_split1 = refBoBTypeA _split1; … BoBTypeA_Splitk x_splitk = refBoBTypeA _splitk; BoBTypeA_AUX x _aux = refBoBTypeA_aux;
T-6	Method (Type1 arg1, …, BoBTypeA x,…, Typen argn)	Method (Type1 arg1, …, BoBTypeA_Split1 x_split1, BoBTypeA_Split2 x_split2, … BoBTypeA_Splitk x_splitk, BoBTypeA_AUX x_aux, …, Typen argn)

Table 3. Client Transformations

It is to be noted that, access to a BoB is only through the public methods exported by it, and hence for client-transformations we need not consider references made to class or object fields. One of the consequences of splitting is that it might create some splits that remain unutilized in the application. Static optimizations can be applied, if required, to the program at a later stage to remove such split classes/objects. Also, some cases present in the Java language are presently not considered in the reorganizer. For example, at present we do not consider *reflection* based class references.

All the transformations (BoB splitting and client-reorganization) are semantics preserving. A framework for evaluating the *operational equivalence* of the split and original BoB programs is provided in [6] .

5. Related Work and Discussion

We discuss here the related work in application partitioning and object refactoring areas and also show how our efforts get inspired by or compliment these works.

5.1. Application Partitioning

Application partitioning, has been active area of research in the last decade. For example, J-orchestra [14], Pangaea [12] , Addistant [13] try to automate application partitioning of arbitrary Java programs, Coign [5] does partitioning of COM based applications. Work on the application partitioning has so far focused mainly on finding optimal ways to partition an application among different nodes, and component conversions into distributed components. Our focus is: (i) to have an entity which is more suitable for such partitioning (ii) create mechanisms or a process by which partitioning goals are externally specified (in manual or semi-automated way) and actual partitioning is automated and transparent. This makes our approach a *purely declarative way* of application partitioning. Additionally the granularity level of partitioning in these systems is objects or components, while in our case, granularity level is finer and is related to the methods of a BoB.

5.2. Class Refactoring

Different methods of refactoring have been proposed in literature [11] [3]. Mens and Tourwe [9] do a comprehensive survey and elaborate these refactoring techniques. For comparison, class refactoring method *Extract Class*[3] provides a means to create new class by moving the relevant fields (*Move Filed*) and methods (*Move Method*) from the old class into a new class. The main intent here is to improve the code design by splitting bloated classes and creating new crisper classes. However, no comprehensive techniques exist to provide refactoring of application classes for functional parititioning as discussed in this paper.

6. Conclusions

Direct refactoring of an application from one deployment scenario to another is difficult. We need to create structuring mechanisms by which functionality of an application can be easily partitioned. Toward this end, we use a structurally simplified model of objects called Breakable Objects. The main motivation is to write the application once and then, as far as possible, generate a scenario specific version by automated refactoring of BoBs.

We have defined a programming model for BoBs in Java, which requires the introduction of only one new construct in the language. The splitting engine generates the class definitions in Java for the BoB splits. This implies that we can use a BoB-based program in conjunction with the existing JVMs without any need to modify the latter.

Splitting engine provides an architecture for BoB splitting and program reorganizations. By keeping the configuration specifications external to BoB, it helps to separate out the partitioning concerns from the component implementation issues and paves the way for a declarative approach to application partitioning. We provided algorithms for BoB splitting and client reorganizations. Most importantly, we have have proved that BoB refactoring is operational semantics preserving.

Once an application has been refactored as above, components can be prepared for the new distributed environments through source or binary level transformations, as applicable in systems like Pangaea [12] and J-orchestra [14].

References

[1] Open source mail clients in java. At: `java-source.net/open-source/mail-clients`.

[2] G. Agha. *ACTORS : A model of Concurrent computations in Distributed Systems*. The MIT Press, Cambridge, Mass., 1990.

[3] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: improving the design of existing code*. Object Technology Series. Addison-Wesley, 1999.

[4] J. Gosling, B. Joy, G. L. Steele, and B. Bracha. *The Java language specification*. Java series. Addison- Wesley, Reading, MA, USA, second edition, 2000.

[5] G. C. Hunt. *Automatic distributed partitioning of component-based applications*. PhD thesis, University of Rochester. Dept. of Computer Science, 1998.

[6] V. Jamwal and S. Iyer. BoB Based Programming and Formalizations. Technical Report KReSIT, IIT Bombay, India, 2005. Available at: `www.it.iitb.ac.in/~vikram/bob-formal.pdf`.

[7] R. E. Johnson and W. F. Opdyke. Refactoring and Aggregation. In *Object Technologies for Advanced Software*, volume 742 of *Lecture Notes in Computer Science*, pages 264–278. First JSSST International Symposium, Nov. 1993.

[8] R. G. Lavender and D. C. Schmidt. Active object: an object behavioral pattern for concurrent programming. *Proc.Pattern Languages of Programs,*, Sept. 1995.

[9] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, Feb. 2004.

[10] M. Mikic-Rakic. *Software Architectural Support for Disconnected Operation in Distributed Environments*. PhD thesis.

[11] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, Urbana-Champaign , IL , USA, 1992.

[12] A. Spiegel. Pangaea: An automatic distribution front-end for java. In J. D. P. R. et. al., editor, *IPPS/SPDP Workshops*, pages 93–99, 1999.

[13] M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A bytecode translator for distributed execution of "legacy" java software. In J. L. Knudsen, editor, *ECOOP*, volume 2072 of *Lecture Notes in Computer Science*, pages 236–255. Springer, 2001.

[14] E. Tilevich and Y. Smaragdakis. J-orchestra: Automatic java application partitioning. In B. Magnusson, editor, *ECOOP*, volume 2374 of *Lecture Notes in Computer Science*, pages 178–204. Springer, 2002.