BoBs : Breakable Objects

Building Blocks For Flexible Application Architectures

Thesis

Submitted in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

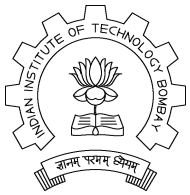
by

Vikram Jamwal

Roll No. 00429402

Supervisor

Prof. Sridhar Iyer



K.R. SCHOOL OF INFORMATION TECHNOLOGY
INDIAN INSTITUTE OF TECHNOLOGY - BOMBAY
MUMBAI - 400 076

(15^{th} August, 2006)

Dedicated to the memory of Late Dr. Pijush K. Ghosh,
and, To my Mother, Father and Sister.

APPROVAL SHEET

Thesis entitled "**BoBs: Breakable Objects - Building Blocks for Flexible Application Architectures**" by **Vikram Jamwal** is approved for the degree of **DOCTOR OF PHILOSOPHY**.

Examiners

_____

_____

_____

Supervisor

_____

Chairman

_____

Date: _____

Place: _____

INDIAN INSTITUTE OF TECHNOLOGY, BOMBAY, INDIA

CERTIFICATE OF COURSE WORK

This is to certify that Mr. Vikram Jamwal was admitted to the candidacy of the Ph.D. Degree on January 16, 2001 after successfully completing all the courses required for the Ph.D. Degree programme. The details of the course work done are given below.

Sr. No.	**Course Code**	**Course Name**	**Credits**
1.	IT 690	Mini Project	10
2.	IT 642	Data Warehousing and Data Mining	6
3.	IT 620	Seminar	4

I.I.T Bombay
Date:

Dy. Registrar (Academic)

Abstract

Computing systems are becoming increasingly varied in terms of computing and networking capabilities. A given application may often need to be deployed in diverse scenarios. Additionally, a software needs to evolve continuously, sometimes in disparate directions, to suit a diverse portfolio of user requirements.

However, given an application defined for one scenario, it may not be possible to simply refactor the application for a *direct reuse* or an *easy adaptation* to another scenario. The application itself may need to be redesigned to enable such flexibility. Hence a pertinent problem is - *How do we design and implement a software system such that it has a lesser compulsion to be redesigned, and has a greater flexibility to adaptation?*

In this thesis we propose a novel way of constructing such applications - wherein an application is built using Breakable Objects or **BoBs**. BoBs are similar to the traditional objects or components, but have the property that they can be readily broken into sub-objects or sub-components. As such, BoBs naturally form the building blocks for flexible application architectures. We term this architecture as **BODA** - Breakable Object Driven Architecture. We claim that: (i) BODA provides an *architecturally robust* mechanism for flexible fine grained reuse, and (ii) BODA *greatly facilitates* automatic application translations for various deployment scenarios.

We apply BODA in the context of object-oriented programming systems. We present a programming model for BoBs in Java called Java$_{BoB}$. Java$_{BoB}$ is a subset of Java language specification, and contains some small extensions to the present language. We also present *mechanisms* by which BoBs can be composed, decomposed and used in an application. Furthermore, the BODA process is illustrated and evaluated in the thesis using three main case-studies and some distilled-data from similar other application experiences .

Contents

List of Figures

List of Tables

The sort of poetry I seek resides in objects man can't touch.

E.M. Forster

Chapter 1

Introduction

Software systems need to *evolve* continuously - to incorporate a new user requirement, to address a new application concern, to adapt to a new environment, or simply to improve the architecture and implementation of existing systems [Leh98]. *Flexibility* is an important software quality and is determined by the capability of a software to respond to a *new requirement*. IEEE Standard Glossary of Software Engineering Terminology (IEEE Std. 610.12-1990) defines flexibility as:

> Flexibility is the ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed.

To facilitate easy adaptation to a new requirement, it is important to design a software in a manner that at a later stage, it is easy to *disassemble or separate* the given *software implementation* into *smaller* components [Par85]. These constituent components then become the units for required modification, replacement and(or)recomposition. Modifications to these *reusable artifacts* can be - black-box, white-box or glass-box [Koj06]. In the *black-box reuse*, the component is reused unchanged. In *white-box reuse*, the component is modified to fit the target product. *Glass-box reuse* allows for inspection of implementation of the component, but the resumable component itself is not modified.

In traditional object-oriented systems, classes/objects[1] are considered as the units of composition and decomposition. In its most basic form an object is composed primitively from methods and fields. Additionally, mechanisms like inheritance and aggregation, allow for its glass box and black-box reuse respectively.

[1]Since, objects are instantiations of classes, the terms *object* and *class* are used interchangibly in this discussion.

However, there are situations, as we show in the following example, where it makes good sense not to consider objects as atomic units (In Chapter 2 of this thesis, we provide a more detailed discussion on this issue). Instead, it is desirable to *extract* a sub-functionality from an object and consider this sub-functionality as a *separate artifact* for reuse and adaptation.

There are few difficulties, though, that might arise when we consider the reuse of sub-functionality of an object:

- Traditional composition mechanisms might compose an object in a manner that it is difficult to separate out its sub-functionality.
- We need mechanisms to:
  - extract the sub-functionality from an object, and
  - reuse the sub-functionality as a unit for composition and modification .

The motivating example in the following section highlights some of these issues.

1.1 Motivating Example

Distributed systems have grown from having nodes with uniform computing and communication capabilities, to having nodes with widely varying capabilities. The underlying communication networks also have become more complex and heterogeneous in nature. The same application may need to be run in different deployment scenarios [2]. But an application designed for one scenario may not be amenable to direct redeployment in another scenario. It may require significant modifications in terms of structuring and placement of its components [MR02].

Consider, for example, an e-mail client application (Figure 1.1). 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 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 a (thin)client manipulates them remotely using an appropriate interface,

(ii) *Offline mode* in which the (thick)client fetches the messages to the local machine and the messages are deleted from the server, and

[2] Different deployment scenarios arise as result of difference in: number of processing nodes involved; underlying network characteristics; communication, processing, power capabilities of nodes etc.

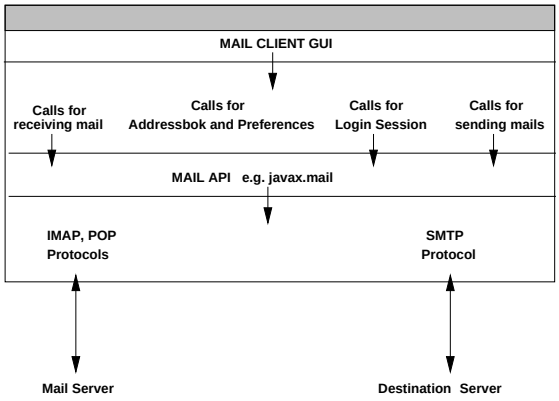


Figure 1.1: *E-mail application*

(iii) *Disconnected mode* in which the (medium-sized)client fetches the messages to the local machine and the messages are also retained at the server.

Although the functionality of the application essentially remains the same, the amount of functionality that is implemented at the server or the client varies, depending upon the mode. Our experience with e-mail applications (icemail, jwma, pooka, popmail [3]) has shown us that refactoring an existing e-mail application to operate in different modes is extremely difficult. For example, in an email implementation (Figure 1.2), the `Store` class may be fully on the client or server, or even 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 the client. Hence we need to automatically refactor the `Folder` class for transforming the application from one mode into another.

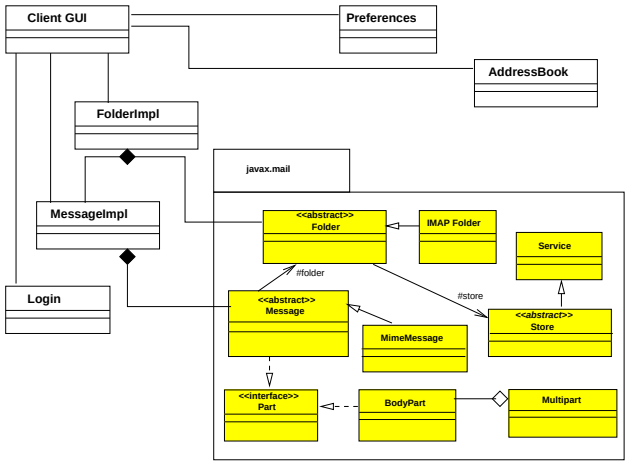


Figure 1.2: *E-mail implementation model*

[3] `java-source.net/open-source/mail-clients/`

To enable automatic refactoring of `Folder`, we need it in a form that can be easily partitioned into sub-entities. But *traditional objects* are *not always suitable* for such partitioning. Consider for example, the folder class implementation, `JwmaFolderImpl` (Figure 1.3), as done in a real-life application, viz., *jwma WebMail*. Corresponding source snippet of this class are shown in the Listing 1.1. Implementation features of `jwmaFolderImp` class like multiple interface inheritance, and embedded functionality in form of enclosed objects (`m_Folder`) make the functionality partitioning of this class very difficult. We discuss this further in detail in Chapter 2.

This points out to the fact that we need to *reformulate* and also perhaps *redesign* our entity, `jwmaFolder`, in a way so as to make it more amenable to such functionality refactorings.

We shall use this example as an illustrating example in the rest of the thesis. It also forms one of our case-studies in Chapter 8. In the next section, we state our problem statement.

Listing 1.1: *Source snippet of Folder implementation in Jwma*

```
public class JwmaFolderImpl    implements JwmaFolder, JwmaTrashInfo, JwmaInboxInfo {

 //associations
  protected JwmaStoreImpl m_Store;
  protected Folder m_Folder;

  //instance attributes
  protected JwmaFolderList m_Subfolders;
  protected JwmaMessageInfoListImpl m_MessageInfoList;
  protected JwmaMessage m_ActualMessage;
  . . .

  public int[] getReadMessages()
      throws JwmaException {
    . . .
    try {
      m_Folder.open(Folder.READ_ONLY);
      Message[] messages = m_Folder.getMessages();
      . . .
    }
  }
  . . .
  public static JwmaFolderImpl createJwmaFolderImpl(JwmaStoreImpl store, String fullname)
      throws JwmaException {

      JwmaFolderImpl folder = new JwmaFolderImpl(store.getFolder(fullname), store);
      folder.prepare();
      return folder;
  }
  . . .
}
```

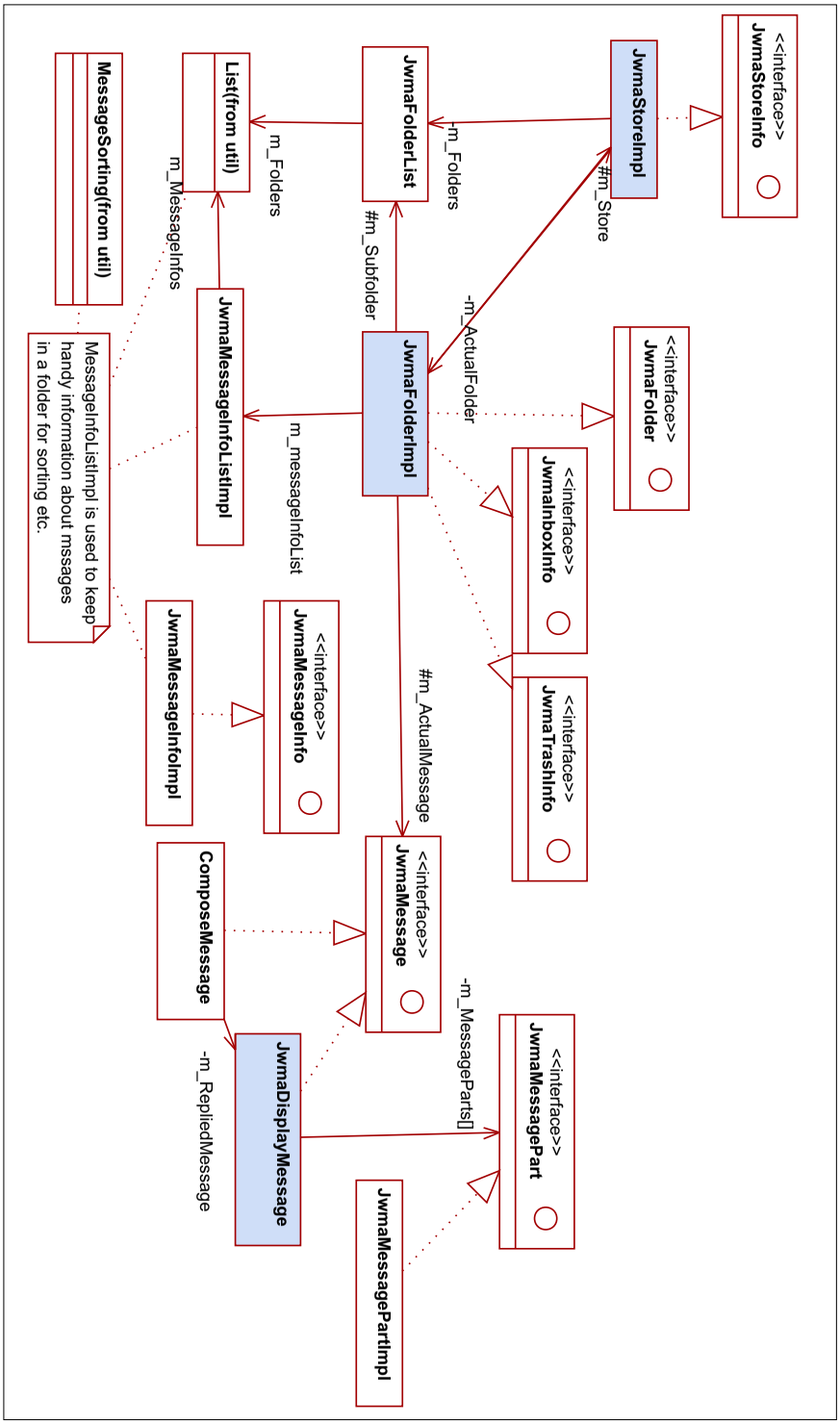


Figure 1.3: *Core classes in the jwma e-mail implementation*

1.2 Problem Statement

Software systems evolution is inevitable [Leh98]. Software reuse is considered to be an important approach to manage evolutions. It is the process of creating software system from existing software rather than building software systems from scratch [Kru92]. An application adaptation, $\mathsf{A} \rightarrow \mathsf{A}'$, might require some refactoring[Fow99] or even a redesign as a pre-step for adaptation (Figure 1.4). Hence, for effective reuse, we need to design and implement a software application in a manner that it has a greater flexibility for direct adaptation and there is lesser need to redesign or refactor.

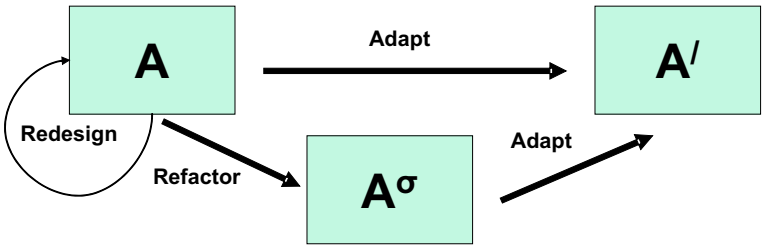


Figure 1.4: *Application adaptation*

With this background, we state our **broad problem** as:

> *Design and implement a software such that, given an application designed for one scenario*[4] *, we can easily adapt the application (through automatic transformations) to a new scenario.*

We believe that, for achieving the above, it is imperative to have (or design) the software base in form such that it can be easily and effectively decomposed into optimally-grained reusable components. Object-oriented and component-oriented programming approaches are a step in that direction. However, the granularity of reuse is fixed : an *object* in the first case, and a *component* in the latter case.

It is desirable to have an approach that allows for the part functionality of an object or a component to be utilized in an optimal and flexible manner.

Now, we state the **more specific problem** that we are trying to address:

> *Design and implement an object-oriented software such that when the software base is decomposed into constituent objects, the functionality of* **some select objects** *is further* **factorable**.

[4]Here *scenario* is used in a much more generic sense; scenario = a feature, a requirement, a use-case, or a deployment set-up

Furthermore, we should be able to factor these select objects in a manner that:

- *units of desired granularity levels can be created,*
- *the units can be easily extracted, and*
- *the units are reusable (i.e. they can be modified, replaced, and recomposed)*

1.3 Solution Outline

We propose the concept of *Breakable Object* (or *BoB* in short), as one effective solution to the above problem.

A BoB is similar to an object in an object-oriented system, but has a simplified structure to allow easy refactoring of its functionality. It is factored on the basis of *interface methods*, resulting in split-fragments which are themselves objects. An added construct, *together*, is used to denote the methods that are designated *inseparable* by the designer of the BoB.

A BoB can be factored in a flexible manner, depending upon the application's factoring requirements. The *splits or fragments* in turn become new artifacts for a finer-grained *reuse*.

1.4 This Thesis

1.4.1 Goal

The aim of our work is two fold:

1. To thoroughly explore the notion of Breakable Object (BoB).
2. To provide automatic tools and language mechanisms for usage of breakable objects in object-oriented softwares. Particularly, we look at two aspects of this usage:
   (a) **Refactoring:** In this, we consider the BoB based program transformations which are behavior-preserving.
   (b) **Extensibility and Composition:** In this, we consider the usage of BoBs which enhances the functionality of a given application.

1.4.2 Contributions of the Thesis

This thesis contributes in the following ways:

1. **The Concepts of Breakable Object (BoB)**: Herein, we identify the need of *viewing* and *treating* some of the application objects or components, as factorable units instead of atomic or rigid units. We highlight various software architectural situations where it is imperative or useful to have such a flexibility.

2. **BODA (Breakable Object Driven Architecture):** We layout the process of constructing applications using BoBs.

3. **Programming Model for Breakable Objects in Java - Java$_{BoB}$:**. We devise a programming model for BoBs. This is a mostly as subset and a minor extension of Java programming language. There is added construct `together` to designate inseparable methods and few compositional operators for BoB class compositions. We also exclude some features of Java language, such as reflection, that hinder easy breakability of a BoB.

4. **Automated refactoring of BoBs - Splitting and Merging of BoBs, Client reorganization techniques:** A software implemented using BoBs undergoes large scale source code level transformations whenever adaptation to a new deployment scenario is required. We provide algorithms and implementation for these automated refactorings in the form of splitting and merging engines.

5. **ASM Models and Proofs:** We extend the definition of *contextual equivalence* and introduce a new notion of *extended contextual equivalence*. We also develop a methodology for proving extended contextual equivalence in programs. We use the formal execution semantics of Java$_{BoB}$ provided by Abstract State Machine(ASM) models to prove that the transformations by splitting and the client-reorganization algorithm are valid.

6. **BoB Composition Mechanisms:** We extend the basic operations of Split and Merge to lay the foundations for BoB compositions mechanisms. We illustrate how operations like Add, Subtract and Replace allow a systematic and easy composition of a BoB from other BoBs or BoB split-fragments.

7. **Applicability of BoBs:** We identify areas where BoBs can be employed. We provide three case studies to illustrate the usage of BoBs in application partitioning and application product families scenarios. Lastly, we also provide guidelines for designing effective and useful BoBs.

1.5 Organization of the Thesis

The remainder of the thesis is organized as follows:

In **Chapter 2** we provide the additional motivations for BoBs and discuss these in the background of related works. We discuss the features of BoB in more detail in **Chapter 3**. Once an application has been developed using BoBs, we can more readily refactor the application for various deployment configurations. Also a BoB-based application design facilitates fine grained evolution of applications. In the latter part of Chapter 3 , we highlight the BoB Driven Architecture (BODA) process. **Chapter 4** specifies the programming model for BoBs in Java. In **Chapter 5**, we show how a splitting engine performs compile-time refactoring of a BoB-based program for various deployment configurations. The engine takes as input the BoB-based program along with a *Split-Configuration* (which specifies the lines along which the BoBs are to be split) and produces a scenario-specific Java program as the output. We use the distributed e-mail application outlined above as an illustrative example. **Chapter 6** provides the equivalence proofs for split and non-split versions of the program.

In **Chapter 7**, we present how the BoBs splits can be merged together, and how BoBs can be deployed in the distributed scenarios. We also describe the implementations of our splitting and merging engines. In **Chapter 8**, we present the case-study of a multi-mode e-mail client that highlights the scenarios in which BoB can be employed and the methodology of implementing an application using BODA. We also demonstrate how BoBs help to reduce the application size when such a requirement is critical, e.g. in a small-devices. In **Chapter 9** of this thesis, we build a formal model for BoBs and discuss BoB composition mechanisms. **Chapter 10** presents additional case-studies using both BoB composition mechanisms and different aspects of BODA. **Chapter 11** we compare BoBs and BODA with the related works. We also discuss the new paradigm that BoBs introduce to the design of distributed systems and compare it to the other distributed design paradigms. We conclude our discussion and indicate the future directions and possibilities that our work with BoBs create in **Chapter 12**.

Appendix A, presents a formal definition of BoB class based on Java_{BoB} specifications and **Appendix B**, contains the Abstract State Machine based execution rules for JavaBoB. **Appendix C** lists the publications related to this thesis work.

Most of the problems of education are problems of motivation.

Jean-Jacques Rousseau

Chapter 2

Motivation and Background

In this chapter, we first show through three software engineering problems, the need of composition and decomposition at class/object fragment level. We start our discussion from a larger context and then focus on the object breakability requirements and issues. In the rest of the chapter we discuss the the background contexts where Breakable Objects would prove to be useful. These are techniques for application partitioning and multidimensional separation of concerns, application designs which support change, and mechanisms which provide variability in software product lines. Finally, we summarize the three motivation scenarios.

First the motivational scenarios:

2.1 Motivation-I: Functionality Partitioning

Distributed systems have grown from having nodes with uniform computing and communication capabilities, to having nodes with widely varying capabilities. The underlying communication networks also have become more complex and heterogeneous in nature.

The same application may need to be run in different deployment scenarios[1]. For example, as discussed in the previous chapter, 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. But an application designed for one scenario may not be amenable to direct redeployment in another scenario. It may require significant modifications in terms of structuring and placement of its components [MR02]

[1]The term *scenario* here has been used in the context of *deployment scenarios* Different deployment scenarios arise as result of difference in: number of processing nodes involved; underlying network characteristics; communication, processing, power capabilities of nodes etc.

Ideally, given an application designed for one scenario, one should be able to automate these modifications i.e., one should be able to generate applications for a new scenario through an automated refactoring process[Men04]. However, this is extremely difficult in practice for the following reasons:

(i) Functionality partitioning: This involves apportioning application functionality into component sub-sets *suitable* for redeployment in new scenarios. This implies satisfying the functional constraints of the applications[Til02], i.e. making a component available at a physical location where it is required. It also includes satisfying the resource constraints of application components [MR04], e.g., CPU requirements of a component, number of run-time threads required to execute the component, sizes of communication events etc.

(ii) Component distribution: This involves distributing an application's component across different nodes and making them work as distributed components. This might involve modifying application's source code, application's binaries prior to execution components, or manipulating application's execution through run-time interventions [Hun98].

(iii) Environmental heterogeneity This involves dealing with differences in environments encountered due to hardware and software constraints on the target environment, e.g., CPU speed, link characteristics, battery power, available system software, operating systems, run time libraries etc. [MR04].

Of these, component distribution and environmental heterogeneity (points (ii) and (iii)) have been areas of active research for quite some time. For example, component heterogeneity may be addressed by having similar run-time environments on all the nodes and component distribution may be addressed by having a middleware framework for translating between local and remote references. However, *functionality partitioning of application* (point (i)) remains an important problem, which still requires much attention.

In order to achieve and possibly automate (i), the main requirement is:

Application functionality needs to be cleanly separated such that the components can be grouped into deployment-specific subsets.

This is hard to achieve in practice as we may not be able to draw clean lines of separation through an application. *Some functionality may span across multiple components, or a single component may include parts of multiple functionality.* This was illustrated by the example in the introduction Chapter 1.

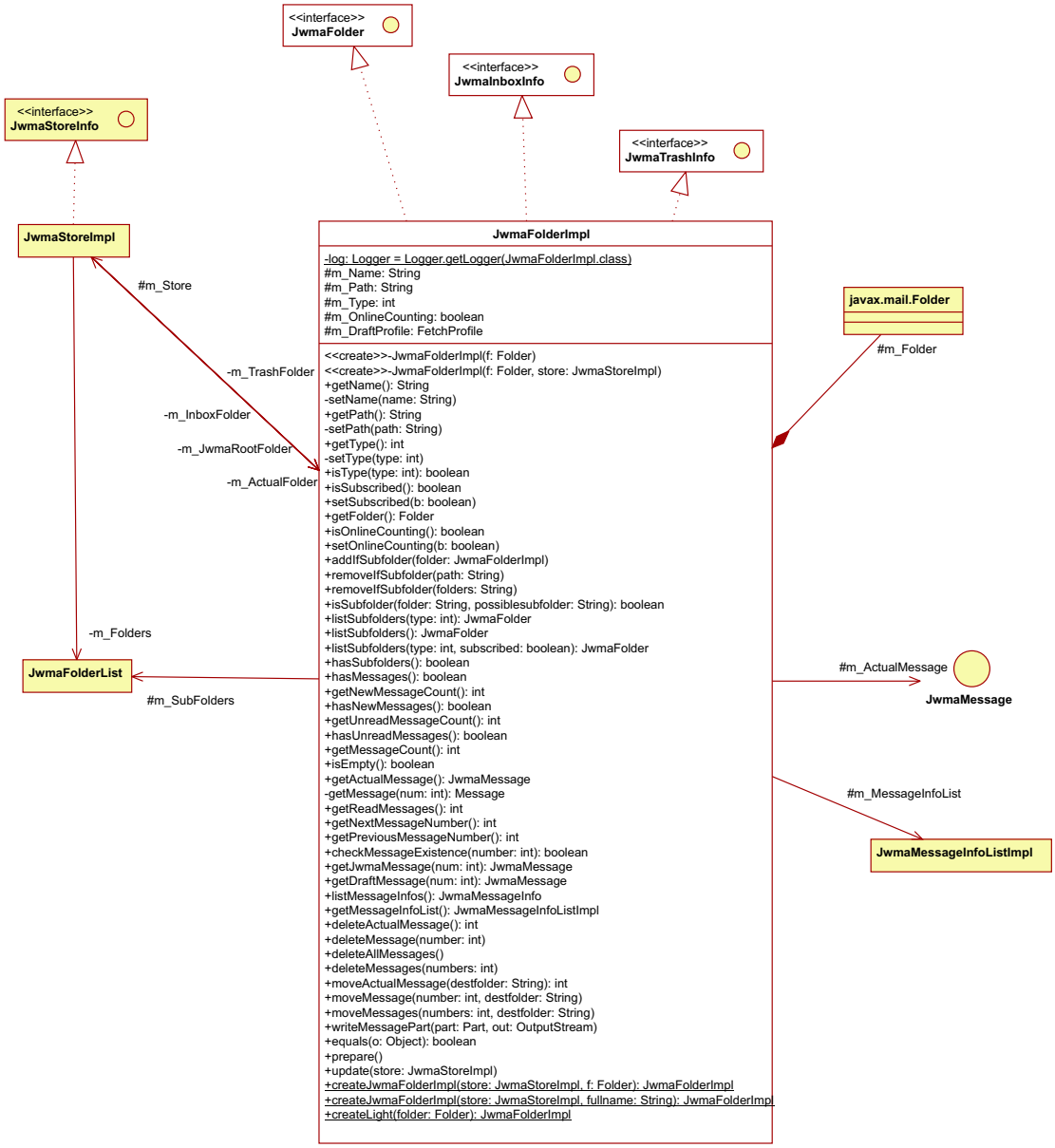


Figure 2.1: *Folder class in the jwma e-mail implementation*

Figure 2.1 shows how the folder class implementation `JwmaFolderImpl`, is arranged in the application (*jwma WebMail*).

There are few things to be noted here: the composition objects (aggregations, associations, attributes), the public interface methods of `jwmaFolderImpl` class, and the constructors.

A deeper look at the source code (see also source code listing 1.1) and design documentation reveals that there are two associations. The first one is `m_Store` is actually a backward reference to the aggregator object `jwmaStoreImpl`. The second one `m_Folder` is a strong aggregation on the *abstract* class `abstract Folder` of `javax.mail` package. The class `jwmaFolder` also acts as a wrapper to `Folder`.

The instance attributes of interest to us are: `m_FolderList` and `m_MessageInfoList` represent the list of sub-folders and the list containing brief message informations respectively. Some methods implementation too have been shown in the source code listing 1.1 to highlight how these methods are using the class and instance fields.

Now, if we want to factor out this `jwmaFolderImpl` class, for the three scenarios described above, we find that it is quite cumbersome and not at all straightforward. The main difficulties that arise are listed below:

- The functionality of the `jwmaFolderImpl` class in the form of methods, is not arranged in a manner so that it is easy to factor or extract out the sub-functionalities. This is understandable because that was never the aim in this implementation.
- The `jwmaFolderImpl` has various aggregate objects, and the functionality is further embedded in these objects.
- The `jwmaFolderImpl` implements three interfaces `JwmaFolder, JwmaTrashInfo`, and `JwmaInboxInfo` and in case we are using polymorphic messages in the application, we cannot *subgroup* the methods from these interfaces.
- The constructors initialize the contained objects, and we need to rewrite them when we factor out the functionality.
- The aggregated `Folder` class is abstract, and it is not clear what should we refactor: abstract class, its implementation, or the full inheritance chain.

These implementation features of `jwmaFolderImp` class make the partitioning of this class very difficult.

This points out to the fact that we need to *reformulate* and also *redesign* our entity, `jwmaFolder`, in a way so as to make it more amenable to such refactorings. In essence, we seek to make it *breakable*.

2.2 Motivation-II: Fine Grained Reuse

Lehman and Ramil [Leh01] describe software evolution as: *All programming activity that is intended to generate a new software version from an earlier operative one.* But new version of a software usually tends to be a bulkier or more comples; quoting Walter Bender, Executive Director of the MIT media lab.

> "I don't think I've ever seen a piece of commercial software where the next version is simpler rather than more complex."

Though the need for designing software for expansion as well as contraction has long been advocated, e.g. in [Par78], the contraction part has been largely been overlooked. We don't have program entities or composition mechanisms which encourage this. In object-oriented domain, techniques like *inheritance* and *aggregation* have proved to be a useful over the years. Other techniques like *multiple inheritance* and *mixins* help to create new classes by combining the implementations (behaviors) or two or more classes, or a class and a mixin.

However, these techniques only allow the *whole* to be added on to the new class. There might be cases where only a part and not the whole is required to be present in the newly created class. Techniques like *multiple selective inheritance* [Dor94] address some of these concerns.

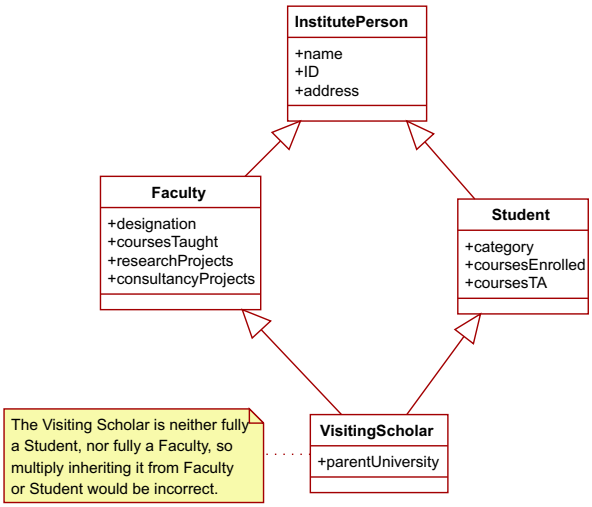


Figure 2.2: *Example of how normal extensibility proves insufficient*

Consider the following example shown in Figure 2.2. From the two classes faculty and student, we need to take out the fragment functionality which we require in the

visiting scholar. Simple extensibility techniques prove in sufficient. Figure 2.3 shows the compositions when we consider part-functionality. In short, we need mechanisms to allow part-functionality extensibility both for contraction, as well as expansion.

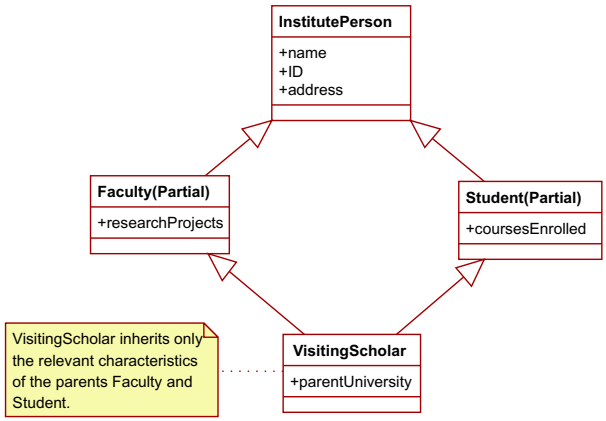


Figure 2.3: *Example for optimal or selective extensibility*

2.3 Motivation-III: Variability Support

Product families requires different products to support variations of some features (Section 2.7 provides a more detailed discussion on the variability issues). A feature variation at product level in an object oriented software can affect an individual class or a series of related classes. Many approaches, including feature oriented programming FOP and role based programming [Che05] [Ste00] have been proposed for achieving this kind of variability.

In this section we shall discuss an example variability scenario, to illustrate the fact that while supporting variability at class-level, the normal extensibility mechanisms like static inheritance chains, may sometimes prove to be inadequate. This example is somewhat similar to the one discussed by Valecia in smalltalk tutorial[2] and also presented by Mezini in [Mez02] [Mez97].

Consider a case, where we want to model and implement Bank Accounts. Many variations that can exist are described below:

[2]http://www.gnu.org/software/smalltalk/gst-manual/gst.html

Single person account Individual accounts which are most common personal investment accounts.

Shared account Joint accounts are set up by more than one person.

Corporate account There are opened by institutions, companies, partnerships, trust and non-profit organizations. Require higher minimum investment.

Express account Designed for people who prefer to bank by ATM, telephone or personal computer, this account usually boasts unlimited check writing, low minimum balance requirements, and low or no monthly fees.

Checking account This account is for the customer who uses a checking account for little more than bill-paying and daily expenses, and does not maintain a high balance.

Saving or Interest-bearing Usually requires a minimum balance to open, with an even higher balance to maintain in order to avoid fees.Interest is paid monthly, at the conclusion of your statement cycle.

Senior/student checking Many institutions offer special checking deals if you are a student or age 55 or over. The perks vary from bank to bank, but may include freebies on checks, cashiers and traveler's checks, ATM use, better rates on loans and credit cards, or discounts on everything from travel to prescriptions.

Money market This account combines checking with savings and/or investment opportunities to help you pursue higher earnings. Requires high minimum deposit to open.

Life-Line These "no-frills" accounts for low-income consumers are typically products with very low monthly fees . They require a low, if any, minimum deposit and balance, and allot a certain number of checks per month.

In Figure 2.4 we try to build a hierarchical class taxonomy of different types of accounts. Although this example does not consider *dynamic* or *conditional* variations such as running account v/s blocked account, or pessimistic v/s optimistic transaction algorithms [Mez02] [Mez97], still we soon encounter many difficulties. Some of these are described below:

- When there many variations of an entity, a single strict hierarchical layout might not work. To include every feature combination, the hierarchy would grow gigantic in size.

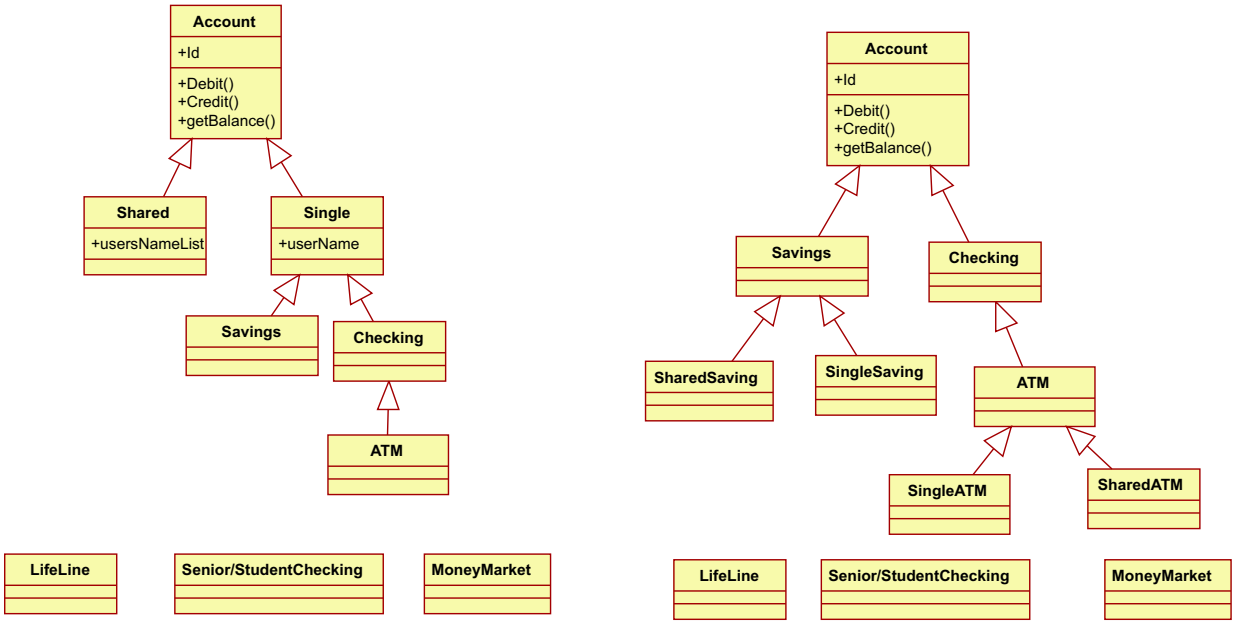


Figure 2.4: *Possible hierarchies for accounts example*

- Multiple inheritance is required and helps to reduce the hieararchy tree-size, but again it has its own set of problems like name conflicts.
- If the feature plans change dynamically or over a period of time, the old hierarchical structure may not remain relevant.

If classes or objects can be viewed as made up of multiple parts, the variation in the different versions has been provided by adding, removing or specializing a certain functionality existing in a class. The main point that we emphasize is that different object incarnations require different specialization. It is difficult to support all the variations through normal inheritance mechanisms. An approach which would allow specializations at *fragment level* is desirable.

In the rest of this chapter, we discuss in detail the software engineering areas that form the background of these motivations.

2.4 Notion of Breakability

We believe that an important property of flexible software is *breakability*, which has previously not been adequately applied at the architectural and design stages of software development. We define it here, informally as:

> *Breakability is ability of the software which has been already designed and/or implemented, to be split into smaller constituent components, which can then become new and independent units for modification or replacement.*

In other words, breakability is the capability of a software base to *disassemble effectively*. This tearing apart of the application and extracting useful and reusable components in a systematic manner, in our view, leads to more managed evolution of the softwares.

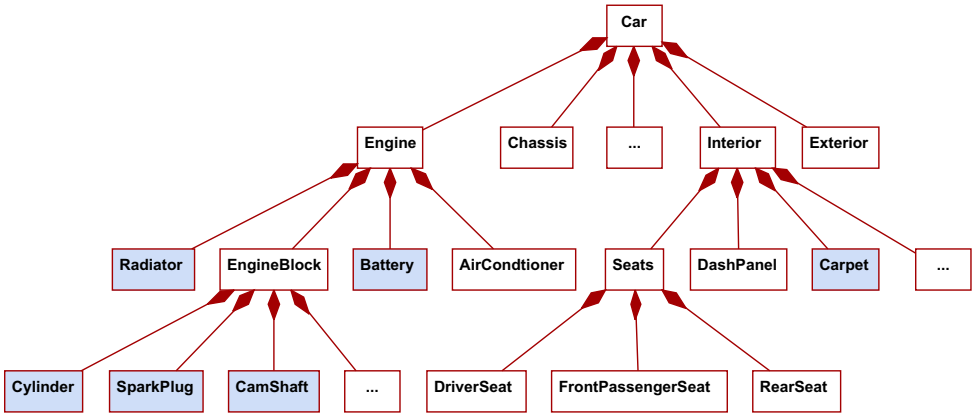


Figure 2.5: *Components of a car-assembly*

The examples of breakability abound in the real life. Consider a *car object* in Figure 2.5. It is made up of a number of components which can be separated or taken apart (shown in white). Some of these components are *unbreakable* (shown in light grey), by which we mean that if we try break (or decompose) them further, we loose their usefulness in the considered context. Such component are, for example, the tyres of a car, the front-glass, hood etc. There are other components which are *breakable*, for example, the engine, which can be further broken down into cylinder, piston etc. The flexibility in the construction of cars, has not only come from the property that these components are replaceable but also from the fact, that the main car object is breakable. This means that we have *ability to extract* these constituent parts. Please note that breakability is also an abstraction and depends upon the view under consideration. It is quite possible that breakable object in once scenario, might be considered breakable in another. Figure 2.6 shows the assembly of a radio control helicopter, which has various levels of breakability. Another example that we mention here, before we consider breakability with respect to software artifacts, is that of a *graphics drawing application*. Consider the *clip-art object* shown in the Figure 2.7. This object can be ungrouped (or split), to yield constituting graphics objects. The

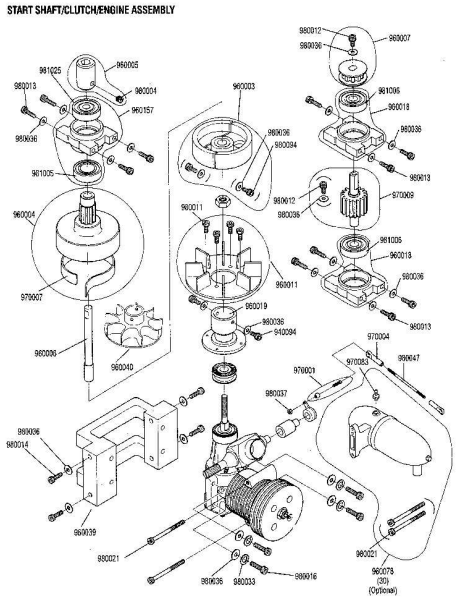


Figure 2.6: *Objects in a Ergo 30/46 CCPM radio control helicopter assembly*

breaking down process can continue until we reach objects which are unbreakable. These applications provide useful operations of the form `group` or `ungroup`. We can group the split sub-objects into new groups and use them in entirely new graphical contexts, thus enhancing the overall flexibility of the drawing application.

In software context, we deal with many forms of decompositions and at various stages of software development. At the *architectural and design stage* we decompose the functionality of requirements into modules and components/objects. We argue, that this does not guarantee or ensure breakability of artifacts once they are transformed into implementation units. Hence, it is equally important how we compose these artifacts. However, there are already few prevailing architectural principles that when applied enhance the breakability of software. For example, *encapsulation* or *designing to an interface* principles enable the implementations to be separated from the interface interactions. These allow implementations to be easily replaced without the need to change the client software.

In object oriented systems, the objects become the atomic units of functionality. The objects can be composed to form other objects. Once composed, however, the composed units become a composite whole, and viewed as singular units. We believe that object themselves, should be considered breakable. This would greatly enhance the *breakability* of an object-oriented software, in turn making it more flexible. In this thesis we look at these issues and consider the implications of BoB splitting

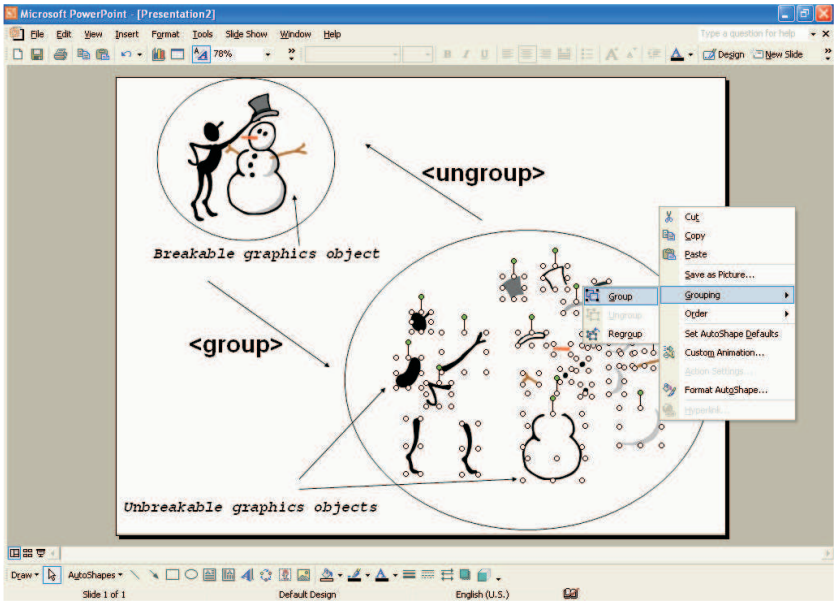


Figure 2.7: *Graphic objects in a PowerPoint application*

transformations - behavior-preserving as well as those that extend functionality of the given application.

2.5 Automatic Application Partitioning

Application partitioning refers to, breaking up of the application into components while preserving the semantics of original application. The source application might be designed, implemented and debugged to run on a stand-alone system. The resulting components, however, are distributed so as to take advantage of distributed setting. *Automatic partitioning*, hence, is described as the process of adding distribution capabilities to an existing centralized application without needing to rewrite the application's source code or needing to modify existing runtime systems. [Lio04]

Partitioning of object-oriented programs is motivated by the following reasons :

Distribution of program components Consider a large-scale, multi-user application using a wide variety of resources in a network. For example, a large banking application has its databases distributed over geographically separated areas. A properly distributed application, would have components distributed all over the network in an optimal manner, reducing amount of remote communication overheads; thereby optimizing the program execution. In contrast, a stand-alone system based application might be situated at some central node

making expensive network calls. In addition to this, a distributed program allows *load-balancing*, where idle hosts can share the load of a busy system by hosting some of the components.

Abstraction over the middleware Remote Method Invocation (RMI), DCOM, CORBA provide mechanisms for developing remotely invocable components in languages like Java, C++, C # etc. However, these middleware technologies can be quite complicated to code. They have special coding constructs, and programmer has to *think about distribution mechanisms* while designing the classes. Java RMI, for example, needs each remote class to be specifically defined to extend a special interface. Creating a remote object and getting its access on remote node is a tedious job.

Abstraction over deployment scenarios Primarily been designed to cater to client-server systems, current middleware technologies also fail to provide enough abstraction for efficient placement of components. The programmer has to *think and consider the deployment aspects* while designing the application. Finally, these constraints prevent the program to be re-deployed efficiently in different distributed settings.

Summing up, automatic application partitioning systems help to *separate distribution concerns from application logic*.

2.5.1 A Taxonomy for Automatic Application Partitioning

We present here a brief discussion for automatic application partitioning techniques. The taxonomy is based on the ones discussed in [Ved06] and [Spi02].

Explicit and Implicit Approaches

Application partitioning approaches, as described in the previous sections, work from within the programming language, rather than modifying execution environments. These are called *explicit* approaches[Spi02]. They work by forming a layer above the standard middleware of the corresponding programming language.

Partitioning can be also achieved by mechanisms that work by modifying the execution environment of an application. These are called as *implicit* approaches. These systems do not work on the programming language level. Instead, a Distributed Shared Memory(DSM) environment is used. DSM mechanisms involve *caching* and *replication*, which can be done at different levels like *page* and *object*. Though they make things more transparent, the efficiency of these a systems is usually lesser than

the explicit approaches[Spi02]. Also, since these are often non-standard mechanisms, they cannot be extended to different machine architectures.

Implicit approaches are beyond the scope of concern of this thesis and we shall not discuss them further. In the rest of this section we discuss with various aspects of explicit approaches.

Distribution Infrastructures

A partitioning framework or middleware works on top of a distribution middleware (DCOM, CORBA, RMI, etc.). It provides an *analysis component* which creates distribution policy, and a *translation component* which processes the distribution policy and transforms the source code into remote-enabled code accordingly. Another major task is to handle the various issues related to the middleware. There are a number of systems that provide such infrastructure support. J-orchestra [Til02], Pangaea [Spi99] , Addistant [Tat01] try to automate application partitioning of arbitrary Java programs, Coign [Hun98] does partitioning of COM based applications

The distributed infrastructures expect a detailed distribution specification about component. Specification (e.g, as in [Til02]) is of the form:

- Location of component, (*class or object* location).
- Parameter passing mode (*visit, move, reference, copy*) for formal arguments for each method.
- Migration policy (*fixed or migrable*) for classes.

Further we need a way to specify distribution policy in the system. Mechanisms for specifying the distributions are of the following form:

- **Additional Constructs** Additional keywords and constructs extending Java language are provided to declare remotely accessible classes. E.g., as in Java Party[Phi97]
- **Comments** Comments written above the classes in a specified format. E.g., as in Doorastha[Dah00]
- **Confiuration File** A configuration file per class. E.g. J-Orchestra[Til02] uses an XML configuration file, Addistant[Tat01] uses a special configuration file, indexed by hosts for the same.

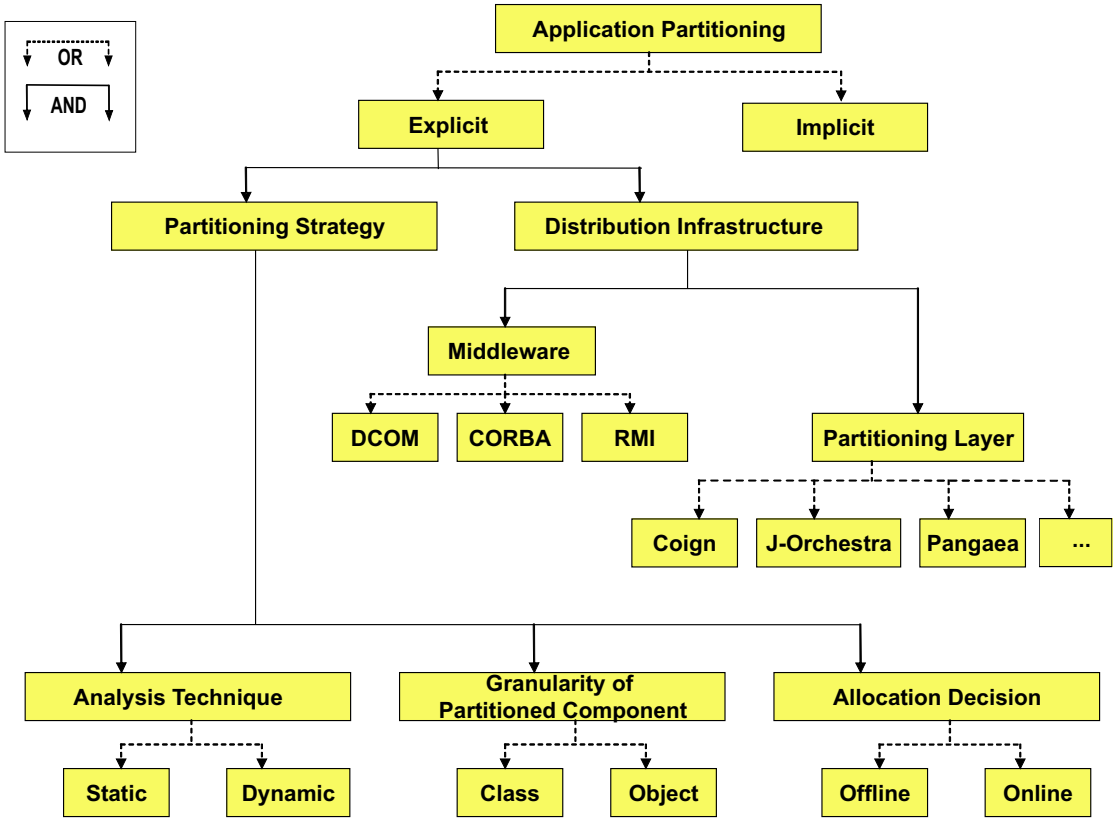


Figure 2.8: *A taxonomy for Automatic Application Partitioning techniques*

Distribution of components causes some interactions to be local while increasing the remote communication overhead for others. A partitioning strategy should tries to balance these two opposing factors. In the next three sections we discuss various features of a partitioning scheme.

Analysis Techniques

The partitioning scheme needs to identify closely interacting components so that they can be co-located. This involves program analysis to identify dependency relationships between the components. The analysis can be done on source code, or on the binary or byte code. Further, the analysis can be *static* or *dynamic*.

- **Static Analysis** Involves techniques like, call-graph creation and doing dependency analysis based on it. Available libraries for code analysis can be used. Source-code level analysis are easier to comprehend, whereas byte-code level analysis techniques have the advantage that they can include analysis of system classes.

- **Dynamic Analysis** Involves instrumentation of the existing code, to profile application runs. This profiling again can be of two types: on the standalone code itself, or distribution-enabled code. Profiling can be used, to study the component interactions and subsequent logs be then analyzed and classified.

Granularity of Distribution Component

Since an object-oriented program comprises of classes and objects, granularity of partitioning component can be class or an object.

- **Class-Level Partitioning** This kind of partitioning aims at grouping closely interacting classes together. The source code comprising of different classes can be divided across the hosts. This kind of partitioning is based on the assumption, that objects of a single class will exhibit similar behavior, use same resources and hence should be placed on the same host. Further, classes can be clustered into groups.

- **Object-Level Partitioning** Object level partitioning are useful where the above assumption does not hold good and objects from the same class may be operating in different contexts and interact with different objects. Allocating all objects of one class to one node can lead to inefficient partitioning. Hence, in object level partitioning, the primarily aim in not the division of source classes, but that of instantiated objects across different hosts.

Allocation Decision

This feature is based on the time of actual allocation the deployment decision.

- **Offline** Component allocation is decided before execution and is specified in the form of annotations within the program or through a configuration file.

- **Online** Component allocation decision is postponed till run-time.

Most the application partitioning systems work by pre-deciding the component allocation before the actual execution of program. Veda [Ved06] discusses a mechanism for online component allocation.

We state the problem that we are trying to solve:

In the next section...

2.6 Separation of Concerns and Modularization

A concern is a piece of interest or focus in a program. Separation of concern is a fundamental software engineering principle which states that a given problem involves different kind of concerns, which should be identified and separated to cope with complexity . This is done to achieve the required quality factors such as robustness, adaptability and re-usability [Aks01].

2.6.1 Modularization

The separation of concerns principles states that each concern of a given software design problem should be preferably mapped to one module in the system. The advantage of this is that concerns are localized and as such can be easier understood, extended, reused, adapted etc. Figure 2.9 shows that the design problem is decomposed into various concerns C1 to Cn and each of these concerns is mapped to a separate module M.

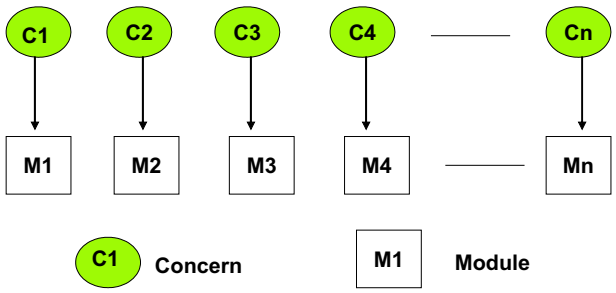


Figure 2.9: *Concerns (C) and Modules (M)*

The term *separation of concern* was first used by Edsger W. Dijkstra [Dij76]. It was expressed in other idioms like *information hiding* by Parnas in [Par72] [Par85], though the term was not used directly. Parnas emphasizes the concept of information hiding modules and discusses the criteria to be used in decomposing system into modules - identifying design decisions that are likely to change and isolating them into separate modules (separation of concerns). He also points out that different design decisions might require different decompositions.

A module is an abstraction of a modular unit in a given design language (class, function, procedure, etc). For example, in structural methods, concerns are represented as *procedures*. In object-oriented programming, the separated concerns are

modeled as *classes or objects* , which are generally derived from the entities in the requirement specification and use cases [Aks01]. Hence, object-oriented decompositions allow problem domain concepts to be directly reflected in the implementation concepts. Further, we can say that the *classes* are the *primary* decomposition mechanisms and the *methods* are the *secondary* decomposition mechanism.

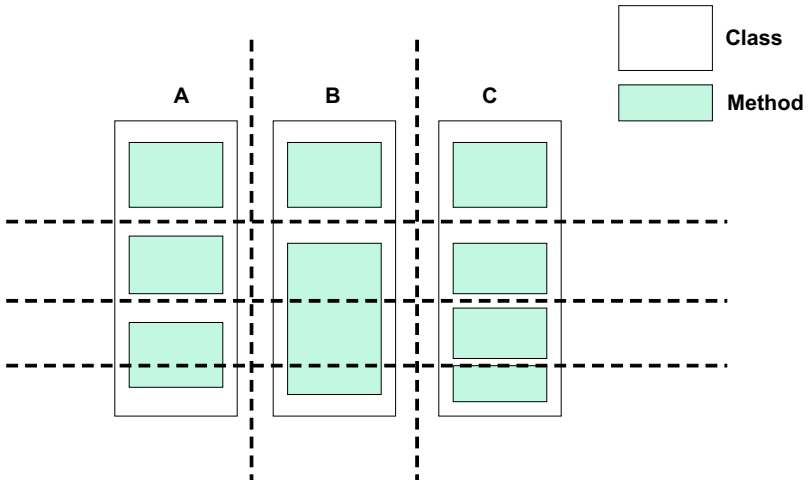


Figure 2.10: *Modularization of concerns in a Class*

Figure 2.10 shows how concern space is divided into classes and methods.

2.6.2 Cross-Cutting Concerns

Many concerns can indeed be mapped to single modules. Some concerns, however, cannot be easily separated, and given the design language we are forced to map such concerns over many modules. This is called *crosscutting*.

In Figure 2.11, for example, concern C4 is mapped to the modules M1, M2, M3, and M4. We say that C4 is a crosscutting concern or an *aspect*. Examples of aspects are monitoring, logging, synchronization, load balancing etc. Aspects are not the result of a bad design but have more inherent reasons. A bad design including mixed concerns over the modules could be refactored to a neat design in which each module only addresses a single concern. However, if we are dealing with these crosscutting concerns this is in principle not possible, that is, each refactoring attempt will fail and the crosscutting will remain. A crosscutting concern is a serious problem since it is harder to understand, reuse, extend, adapt and maintain the concern because it is spread over so many places. Finding the places where the crosscutting occurs is the first problem, adapting the concern appropriately (without unforeseen effects) is

another problem.

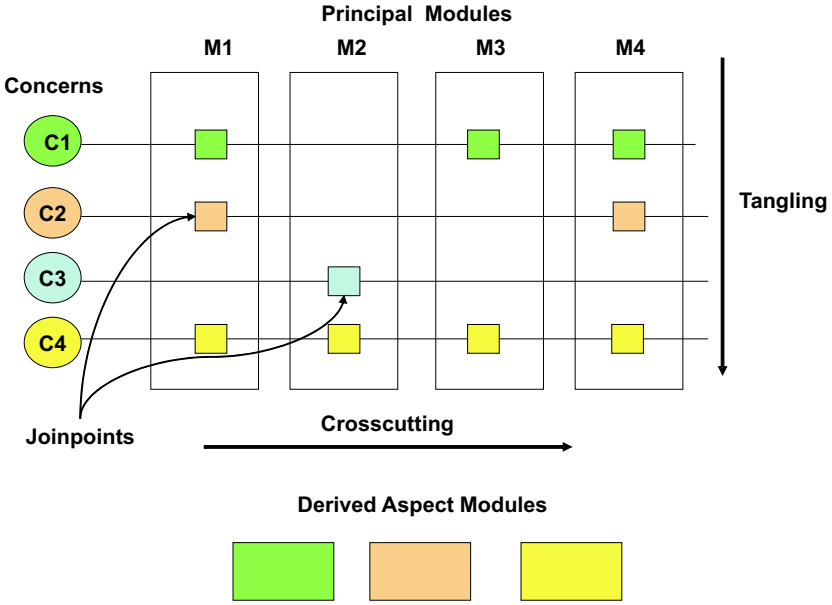


Figure 2.11: *Cross-Cutting and Tangled Concerns*

Tangled Concerns

Since we cannot easily localize and separate crosscutting concerns several modules will include more than one concern. We say that the concerns are tangled in the corresponding module. For example in Figure 2.11, the concerns C1, C2 and C4 are tangled in the module M1 and M4. Note that concern C3 is not crosscutting.

Joinpoints

In Figure 2.11 the modules have been aligned over the horizontal axis and the concerns over the vertical axis. The squares represents the places where the concerns crosscut a module. These are called *joinpoints*. Joinpoints can be at the level of a module (class) or be more refined and deal with sub-parts of the module (attribute, operation) etc. Crosscutting can be easily identified if we follow a concern in a horizontal direction (multiple joinpoints). Tangling can be detected if we follow each module in the vertical direction (multiple joinpoints).

2.6.3 Aspect Decomposition

Conventional abstraction mechanisms may fail to appropriately cope with these crosscutting concerns. Aspect approach provides explicit abstractions for representing crosscutting concerns. As such, a given design problem is decomposed into concerns that can be localized into separate modules and concerns that tend to crosscut over a set of modules. *Pointcuts* specify the points at which the aspects crosscuts. To each poincut and *advice* can be attached, which specifies the behaviour that is needed at those pointcuts.

2.6.4 Multidimensional separation of concerns

The concept of multi-dimensional separation of concerns(MDSOC) was proposed in [Tar99] and the key idea is to *simultaneously* support various concern encapsulations in a software system. This includes overlapping and interacting concerns. Done well, a multidimensional separation of concerns, enables *on-demand remodularizations*, thus ameliorating many of the limitations of the *dominant* initial modularization. The many advantages that are sought through MDSOC are improved comprehension, better reuse, resilience to impact of change, and improvement is ease of maintenance, evolution and traceability of the programs.

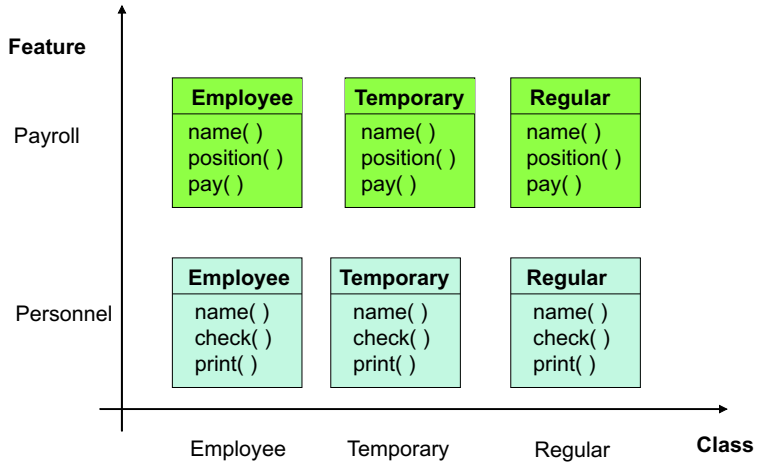


Figure 2.12: *Hypersplices in payroll application*

Once such approach is *hyperspaces* [Oss01a]. In it, based on a concern a set of units can be selected to from modules called *hyperslices*. Thus a hyperslice encapsulates concerns. Relationships between hyperslices can be specified, and they can be used to control flexible composition of hyperslices into *hypermodules*. Figure 2.12 shows

the hyperslices along the two features - payroll and and personnel. Each hyperslice separates these concerns in the classes `Employee`, `Temporary`, adn `Regular`. Sets of hyperslices thus represent different decompositions of the software and by composing them in a desired manner, components and systems can be built.

In the next section we consider various mechanism that exsit for achieving variability in software product lines.

2.7 Variability in Software Product Lines

A software product family is a collection of systems sharing a managed set of features derived from a common set of core software assets [Cle01].

Software product families are required, when there is a need to create different (but related) software products in order to better service the various market segments, and the amount of software necessary for individual products is substantial. It becomes important to exploit the commonality between different products and to implement the differences between the products as *variability* in the software artifacts.

Software variability is then the ability of a software system or artefact to be efficiently extended, changed, customized or configured for use in a particular context [Rie03]. It can also be defined as the ability of a system, an asset, or a development environment to support the production of a set of artifacts that differ from each other in a pre-planned fashion [Bac05].

2.7.1 Features

It is important to identify where the variability is needed in the software product family. Variability is more easily identified when a system is modeled using *features*. A feature is a logical unit of behavior that is specified by a set of functional and quality requirements [Gur01]. Features are used to differentiate various products in a product family. The process of identifying variability, hence consists of listing features which may vary between products. The features can be classified as:

Mandatory These are the features that must be present in all the variants. In essence, they identify the product.

Variant A variant is an abstraction for a set of related features. In *XOR* variants, only one of the variants is selected and in *OR* variant, more than one variant may be selected.

Optional These features have the option to be selected or rejected for a product. When selected they add some value to the core features.

External These are the features, which result from the capabilities of the deployment platform and are external to the main requirement specification.

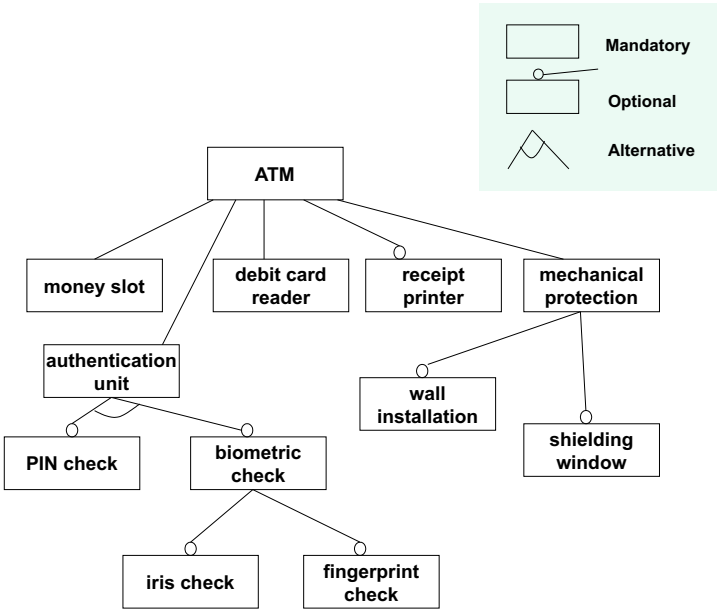


Figure 2.13: *ATM Feature Model*

Figure 2.16 shows an example for an ATM product line with a feature ATM as concept feature using Feature Oriented Domain Analysis(FODA) method [Kan90]. The feature debit card reader is a mandatory feature, stating that it is common to all instances of the domain, because every ATM has a reader for debit cards. The feature receipt printer is marked as optional by an empty bullet, because there are ATMs in this product line example without a printing device.

Feature models help software architects and software developers [Rie03] in the development of a product line by:

- defining reusable components and separating them according to the Separation of Concerns principle
- assigning reusable components to variable features
- describing dependencies and constraints between components and features
- controlling the configuration of products out of the reusable components

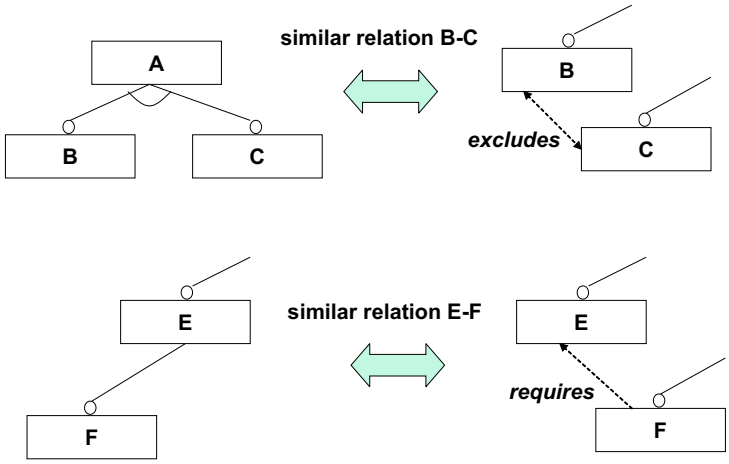


Figure 2.14: *Similar semantics of hierarchical relation and dependency relation*

FODA also introduces some composition rules like *requires* and *mutex-with*. These rules control the selection of variable features in addition to hierarchical relations. If, for example as shown in Figure 2.14, a feature F is selected and there is relations F requires E, then E has to be selected as well. Oppositely, if there is a relation $\mathsf{B\ mutex-with\ C}$, then if B is selected, C has to be unselected.

Thus, a feature model provides an overview of the requirements, and it models the variability of a product line. It is used for the derivation of the costumer's desired product and provides a hierarchical structure of features according to the decisions associated with them.

By linking features to requirements, detailed information from the problem domain is reachable. By linking features to elements of design and implementation, additional information about details of the solution domain are provided. These links are built using traceability links as shown in ERM diagram Figure in 2.15.

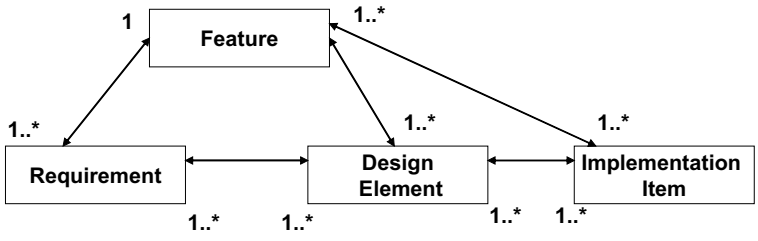


Figure 2.15: *References between features, requirements, design and implementation*

2.7.2 Realizing Variability

The product family architecture and shared product family components must be designed in such a way that the different products can be easily and effectively supported. These related products might be achieved by replacing components, extensions to the architecture or particular configurations of the software components.

Additionally, the software product family must also incorporate variability to support likely future changes in requirements and future generations of products. This means that when designing the commonalities of a software product line, not all decisions can be taken. Instead, design decisions are left open and determined at a later stage, e.g. when constructing a particular product or during runtime of a particular product.

Nature of Variant

A variant can be implemented in many ways, for example as a component, a set of classes, a single class, a few lines of code, or a combination of all of these. Table 2.7.2 shows different software entities are most likely to be in focus during the different stages of variability implementation: architecture design, detailed design, implementation, compilation and linking.

Table 2.1: *Entities involved in different development stages [Sva05]*

Development activities	Software entities in focus
Architecture design	Components ,Frameworks
Detailed design	Framework implementations, Sets of classes
Implementation	Individual classes, Lines of code
Compilation	Lines of code
Linking	Binary components

Variant Introduction

When a variant feature is introduced into the software product family (by a domain engineer) it takes the form of a set of variation points. The variation points are used to connect the software entities constituting the variants with the rest of the software product family. The decision on when to introduce a variant feature is governed by a number of things, such as:

- the size of the involved software entities;

- the number of resulting variation points;
- the cost of maintaining the variant feature.

Variant Population

During this step the software entities of the variants are created such that they fit together with the variation points introduced in the previous step. After this, the variation points are instrumented to be made aware of each new variant. Depending on how a variation point is implemented, the population is either done *implicitly* or *explicitly*.

If the population is *implicit* the variation point itself has no knowledge of the available variants and the list of available variants is not represented as such in the system. With an *explicit* population the list of available variants is manifested in the software system. This means that the software system is aware of all of the possible variants, can add to the list of available variants and possesses the ability to discern between them and by itself select a suitable variant during runtime.

Variant Binding

The main purpose of introducing a variant feature is to delay a decision, but at some time there must be a choice between the variants and a single variant will be selected and used. This is referred to as *binding* of a variant. Binding can be done at several stages during the development and also as a system is being run. Consequently binding can happen during any of these phases: (i) Product architecture derivation, (ii) Compilation, (iii) Linking, or (iv) Runtime.

The other aspect of binding is to decide whether or not the binding should be done *internally or externally*. Internal binding implies that the system contains the functionality to bind to a particular variant. This is typically true for the binding that is done during a system runtime. External binding implies that there is a person or a tool external to the system that performs the actual binding. This is typically true for the binding that is done during product architecture derivation, compilation and linking, where tools such as configuration management tools, compilers and linkers perform the actual binding.

Variability Mechanisms

There are various mechanisms [Jac97] [Gac01] for achieving architecture variability:

- component replacement, omission, replication
- parameterization (including macros, templates)

- compile-time selection of different implementations (e.g., #ifdef)
- OO techniques: inheritance, specialization, and delegation
- application frameworks
- configuration and module interconnection languages
- generation and generators
- aspect-oriented programming - an approach for modularizing system properties that otherwise would be distributed across modules

In the next section we shall discuss the various contexts in which these variability mechanisms are used.

2.7.3 Variability Realization Techniques

These provide different ways to implement the variation points for a variant feature. A taxonomy of variability realization techniques is presented in [Sva05]. We discuss these techniques briefly with respect to their *intent* and *variability solution mechanisms*, below:

Architecture reorganization Supports several product-specific architectures by reorganizing (i.e. changing the architectural structure of components and their relations) the overall product family architecture.

In this realization technique, the components are represented as subsystems controlled by *configuration management tools* or, *Architecture Description Languages (ADLs)*. Examples are Koala [Omm02], XVCL [Zha04], or different ADLs [Med00]). Which variants are included in a system is determined by the configuration management tools. Some variation points may be resolved at this level, as the selected components may impose a certain architecture structure. Typically this technique also requires variation points that are in focus during later stages of the development cycle in order to work.

Variant architecture component Support several, differing, architecture components representing the same conceptual entity.

The solution is to support a set of components, each implementing one variant of the variant feature. The selection of which component to use at any given moment is then delegated to the *configuration management tools* that select what component to include in the system.

Optional architecture component Provide support for a component that may, or may not, be present in the system. That is, it may be present in one product and not the other.

There are two ways of solving this problem. If we want to implement the solution on the *calling side*, the solution is simply delegated to variability realization techniques introduced during later development phases. To implement the solution on the called side (convenient, but less efficient), we can create a null component. This is a component that has the correct interface, but replies with dummy values.

Binary replacement - linker directives Provide the system with alternative implementations of underlying libraries.

The solution is to represent the variants as *stand-alone library files*, and instruct the *linker* which file to link with the system.

Binary replacement - physical Facilitate the modification of software after delivery.

In order to introduce a new variation point after delivery, the software binary must be altered. The easiest way of doing this is to replace an entire file with a new copy. To facilitate this replacement the system should thus be organized as a number of relatively small binary files, to localize the impact of replacing a file. Furthermore, the system can be altered in two ways: either the new binary completely covers the functionality of the old one or the new binary provides additional functionality in the form of, for example, a new variant feature using other variability realization techniques.

Infrastructure Centered Architecture Make connections as first-class entity.

The solution is to convert the connectors into first-class entities, so the components are no longer connected to each other but are rather connected to the infrastructure, i.e. the connectors. This infrastructure is then responsible for matching the required interface of one component with the provided interface of one or more other components. The infrastructure can either be an *existing standard*, such as *COM* or *CORBA* [Szy02], or it can be an *in-house developed* standard such as *Koala* [Omm02]. The infrastructure may also be a *scripting language*, in which the connectors are represented as snippets of code that are responsible for binding the components together in an architecture. These code snippets can either be done in the same programming language as the rest of the system, or they can be done using a specialized scripting language [Ous98].

Variant Component Selection Adjust component implementation to product architecture.

Solution is to separate the interfacing parts into separate classes that can decide the best way to interact with the other component. Let the configuration management tool decide what classes to include at the same time as it is decided what variant of the interfaced component to include in the product architecture.

Optional Component Specializations Include or exclude part of component implementations.

One solution is to separate the optional behavior into a separate class and create a null class that can act as a placeholder when the behavior is to be excluded. Let the configuration management tools decide which of these two classes to include in the system. Alternatively, surround the optional behavior with compile-time flags to exclude it from the compiled binary.

Runtime Variant Component Specialization Support the existence and selection between various specializations inside a component implementation.

There are several *Design Patterns* [Gam00] that are applicable here, for example *Strategy, Template Method* and *Abstract Factory*. Alternating behavior is collected into separate classes, and mechanisms are introduced to select, at runtime, between these classes. Design Patterns use language constructs such as *inheritance* and *polymorphism* to implement the variability. Alternatively, generative programming solutions such as *C++ templates* may also be used [Cza00].

Variant Component Implementations Support several coexisting implementations of one architecture component so thateach of the implementations can be chosen at any given moment.

Solution is to implement several component implementations adhering to the same interface and make these component implementations tangible entities in the system architecture. There are a number of *Design Patterns* [Gam00] that facilitate this process. For example, the *Strategy* pattern is on a lower level a solution to the issue of having several implementations present simultaneously. Using the *Broker* pattern is one way of assuring that the correct implementation gets the data, as are patterns like *Abstract Factory* and *Builder*.

Condition on Constant Support several ways to perform an operation, of which only one will be used in any given system.

As a solution we can use two different types of conditional statements. One form includes pre-processor directives such as C++ *#IFDEFs*, and the other is the traditional *if-statements* in a programming language. If the former is used

it can be used to alter the architecture of the system, for example by opting to include one file over another or using another class or component, whereas the latter can only work within the frame of a given system structure. Another way to implement this variability is by means of *C++ templates* which are handled similarly to pre-processor directives by most compilers.

Condition on Variable Support several ways to perform an operation of which only one will be used at any given moment but allow the choice to be rebound during execution.

The solution is to replace the constant used in *condition on constant* with a *variable* and provide functionality for changing this variable. This technique cannot use any compiler directives but is rather a pure programming language construct.

Code Fragment Superimposition Introduce new considerations into a system without directly affecting the source code.

The solution to this is to develop the software to function generically and then superimpose the product-specific concerns at a stage when the work with the source code is completed. There are a number of tools for this, for example *Aspect Oriented Programming* [Kic97], where different concerns are weaved into the source code just before the software is passed to the compiler and, *superimposition* [Bos99], where additional behavior is wrapped around existing behavior.

Table 2.7.3 presents a detailed snapshot of various variability realization techniques.

Table 2.2: *Snapshot of variability realization techniques*

SNo.	Name	Involved Entities	Variant Introduction	Open for adding variants	Binding times	Variants Collection	Functionality for binding
1	Architecture reorganization	Co,Fr	AD	AD	PAD	Imp	Ext
2	Variant architecture component	Co,Fr	AD	AD, DD	PAD	Imp	Ext
3	Optional architecture component	Co, Fr	AD	AD	PAD	Imp	Ext
4	Binary replacement linker directives	Co, Fr, CoI,FrI,Cl	AD	L	L	Imp/Exp	Ext/Int
5	Binary replacement physical	Co, Fr, CoI, FrI, Cl	AD	AC	BR	Imp	Ext
6	Infra-structure centered	Co, Fr	AD	AD,L,R	C,R	Exp	Int
7	Variant component specializations	CoI,FrI,Cl	DD	DD	PAD	Imp	Ext
8	Optional component specializations	CoI,FrI,Cl	DD	DD	PAD	Imp	Ext
9	Runtime variant component specializations	CoI,FrI,Cl	DD	DD	R	Exp	Int
10	Variant component implementations	CoI,FrI,Cl	AD	DD	R	Exp	Int
11	Condition on constant	LoC	I	I	C	Imp	Int/Ext
12	Condition on variable	LoC	I	I	R	Imp/Exp	Int
13	Code fragment super-imposition	CoI,FrI,Cl, LoC	C	C	C/R	Imp	Ext

Co: Components, **Fr**: Frameworks, **CoI**: Component Implementation, **FrI**: Framework Implementation, **Cl**: Classes, **LoC**: Lines of Code,

AD: Architecture Design, **DD**: Detailed Design, **PAD**: Product Architecture Derivation, **AC**: After Compilation, **I**: Implementation, **C**: Compilation, **L**: Linking, **BR**: Before Runtime **R** : Runtime,

Imp: Implicit, **Exp**: Explicit, **Int**: Internal, **Ext**: External.

Table 2.3: *Motivational Scenarios*

Motivation	Larger Context	Particular Context
Ex I	Application Partitioning and MD-SOC	Class Partitioning
Ex II	Software reuse	Partial class extensibility and composition
Ex III	Variability in Product Lines	Class functionality level variability

2.8 Summary of Example Scenarios

There is need to support constant evolution of softwares. Such a need arises due to changes in software deployment environment or new user requirements in the form of *new features* and *new concerns*. It is important that software is designed and implemented in a manner so that it can easily be evolved for the above scenarios.

We observed that once decomposed into modules, classes/objects can become entangled in the software base in a manner that it is difficult or cumbersome to disassemble them for modification, replacement or reuse.

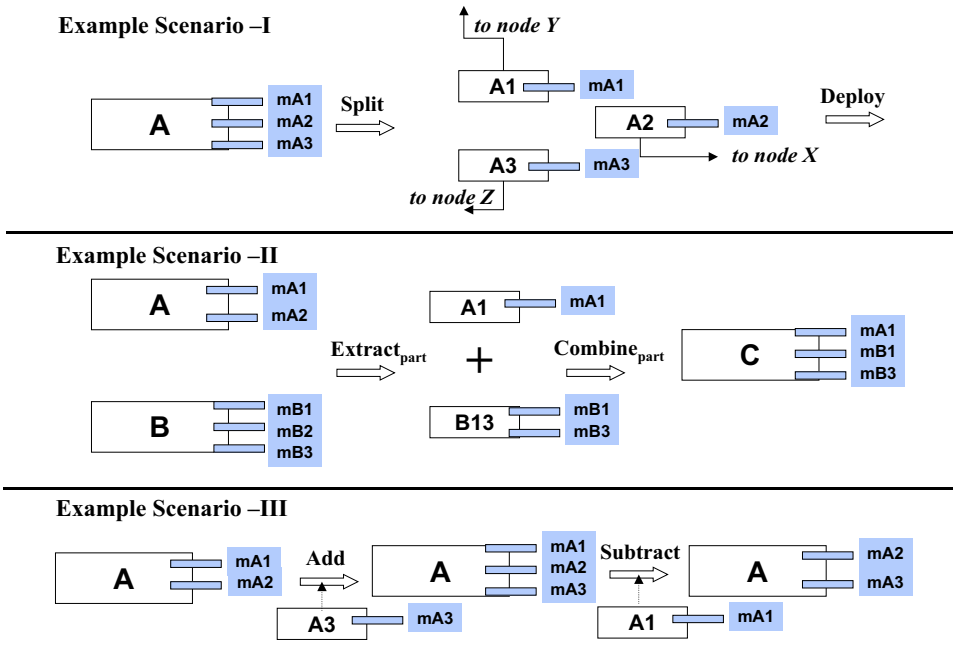


Figure 2.16: *Summary of partial functionality usages*

We need our software in the form so that we can easily extract its functionality. Additionally, it is desirable to have mechanisms which allow part functionality of a

class to be exploited. We summarize the motivational scenarios presented in this chapter in Table 2.3, which shows the larger contexts in which they occur and the particular aspects that are of interest to us. Figure 2.16 summarizes the three ways of partial-functionality usage for a component that results from these motivational scenarios.

In the next chapter, we introduce Breakable Object (BoB) and Brekable Object Driven Architecture (BODA).

To create architecture is to put in order. Put what in order?
Function and objects.

Le Corbusier

Chapter 3

BoB and BoB Driven Architecture

This chapter provides the definition for Breakable Object(BoB) and illustrates the programming process of applying BoBs in an application - BoB driven architecture(BODA).

3.1 BoB

An informal definition of BoB is as follows:

Definition 3.1.1 (BoB) *A BoB is an entity (class/object) 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.*

BoB supports an interface and the partitioning is done on the basis of this interface. It has an added construct - **together** - *to specify the inseparable groupings on its interface methods.*

A more precise definition is dependent on the specific programming language model being used. A BoB may be - a class/object in class based systems, an object in object based systems, a component in component driven systems, or a service in service oriented programming systems.

In this thesis we concern ourselves with *class-based* programming language models only. Definition for Java BoB is given in chapter **??**

3.1.1 Features of a BoB

The various features of a BoB are listed below:

BoB Name : It is the name of the class that implements the BoB.

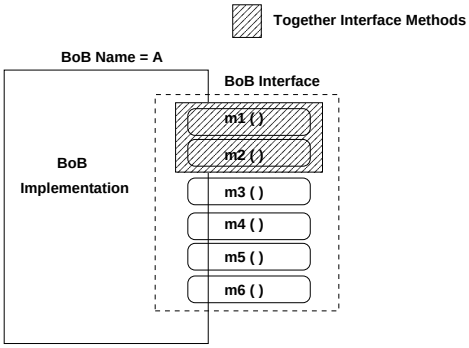


Figure 3.1: *BoB*

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.
- *Inseparable 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.

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 chapters.

Figure 3.1 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};`

3.1.2 Splitting Specifications

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

For example, consider the BoB `A` (Figure 3.1).

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

Given split-configuration:

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

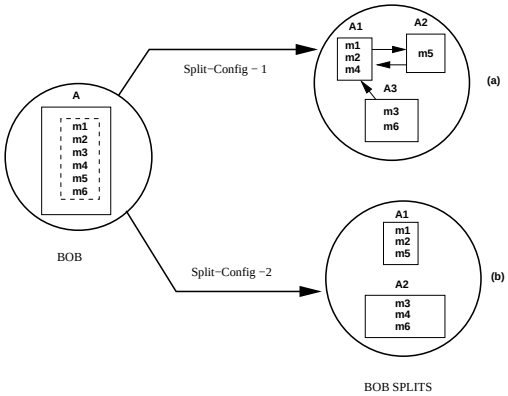


Figure 3.2: *BoB splits generation*

After the splitting process (Figure 3.2 a) , `A` is split into sub-classes $\mathtt{A}_1, \mathtt{A}_2, \mathtt{A}_3$ supporting the interfaces:
$\mathtt{I}_\mathtt{A}^1 = \{\mathtt{together}(\mathtt{m1}, \mathtt{m2}), \mathtt{m4}\}$,
$\mathtt{I}_\mathtt{A}^2 = \{\mathtt{m5}\}$, and
$\mathtt{I}_\mathtt{A}^3 = \{\mathtt{m3}, \mathtt{m6}\}$ 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: $\mathtt{A} \xrightarrow[\mathtt{Split}_{\mathtt{cfg}}]{\otimes} \mathtt{A}_1 + \mathtt{A}_2 + \mathtt{A}_3$

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

3.2 Nature of Splits

We now discuss some of the properties of the splits:

3.2.1 Multi-form splits

The way in which a BoB can be split into sub-object sets is *multi-form*. That is, both, the *number* of splits that are obtained by splitting a BoB, and the *form* of these splits (determined by the interface methods supported by it) is variable. For example, in the Figure 3.2, the BoB after splitting can acquire any of the two different split forms - (a) or (b), depending upon whether split-config 1 or 2 is used.

3.2.2 Exclusive Splits and Overlapping Splits

If a BoB is split in a such a way that one interface method can belong to only one split, the splits are said to be *exclusive*. The case, where a method can belong to more

than one split, the splitting is said to be *overlapping*. In this thesis we use exclusive splitting to build our partitioning models. The overlapping splitting can then easily be incorporated as a simple extension of exclusive splitting.

3.2.3 Independent and Dependent Splits

The splits generated can be independent or dependent. A split is called an *independent split* if does not refer to and is not referred by any other split from the same BoB. Otherwise it is called a *dependent split*. The split in Figure 3.2(a) yields dependent splits, while the one in Figure 3.2(b) yields two independent splits. Independent splits may lead to ease of redeployment.

3.2.4 Types Implications of Splitting a BoB

Splitting a BoB introduces new classes or *types* into the system. The new classes can either entirely replace the principal class in the system or they can co-exist with the principal class. We shall discuss the implications of these replacements, when we discuss splitting-engine details in Chapter 10.6. In the present model, the replacement occurs universally at the class level for every object instance of the principal BoB's type.

3.2.5 Levels of Splitting

Since a BoB can contain another BoBs, the splitting can be applied at any level. If a contained BoB is shared between more than two splits, it can be further split between them. The level of splitting required, depends on the context of use and is an externally specified parameter.

3.2.6 Splitting Multiple BoBs

If there are more than one BoBs in a program that need to be split, the splitting is done iteratively. That is, we proceed by splitting one BoB, reorganizing the rest of program and then repeating the process for the next BoB, until we are split all the BoBs which were intended to be split. We shall discuss this in more detail in further sections.

In the next section we describe the methodology of applying BoBs in an application.

3.3 BoB Driven Architecture (BODA)

We call an application architecture based on BoBs as *BoB Driven Architecture (BODA)* and the resulting process for achieving relocation of an application from one scenario to another as *BODA process*. In general, we can say that a BoB driven architectural process supports two types of application translations:

1. BODA for Application Partitioning: In this, the new concern that we want to address involves partitioning of an application into different partition spaces. For example, such a need arises when do automatic transformation of a legacy code into a distributed application deployed on multiple nodes.

2. BODA for Software Evolution: In this, the new concern or requirement that we address, modifies the basic functionality of the application. Hence, there is a need to provide for effective mechanisms for extension and contraction of the software system. In fact, the ideal that we want is: *the new increment (or decrement) should be in such a manner that architectural quality of the system is preserved*. BODA tries to achieve this by using composition mechanisms which, not only provide fine-grained reuse solutions, but also provide an *architecture sensitive* approach and methodology.

In practice, both these translations need not be always mutually exclusive. For example, we might light to partition and application on some concern and then consider evolution of that particular partition. Similarly, an effect evolution might need partitioning or re-modularization of certain parts of the software system.

The next section detail these two BoB driven architectural processes.

3.4 BODA - Application Partitioning

We discuss briefly the 3-stage BODA process for application partitioning:

3.4.1 Stage 1: Design and implementation

In this stage (steps 1-3, Figure 3.3), a *deployment and context independent* version of the application implementation is prepared. We proceed with the analysis, design and implementation phases of software development similar to those for normal object-oriented programs. We also take the following additional steps:

- The program designer divides the application functionality into a set of objects and BoBs. The choice as to which class should be defined as a simple class and

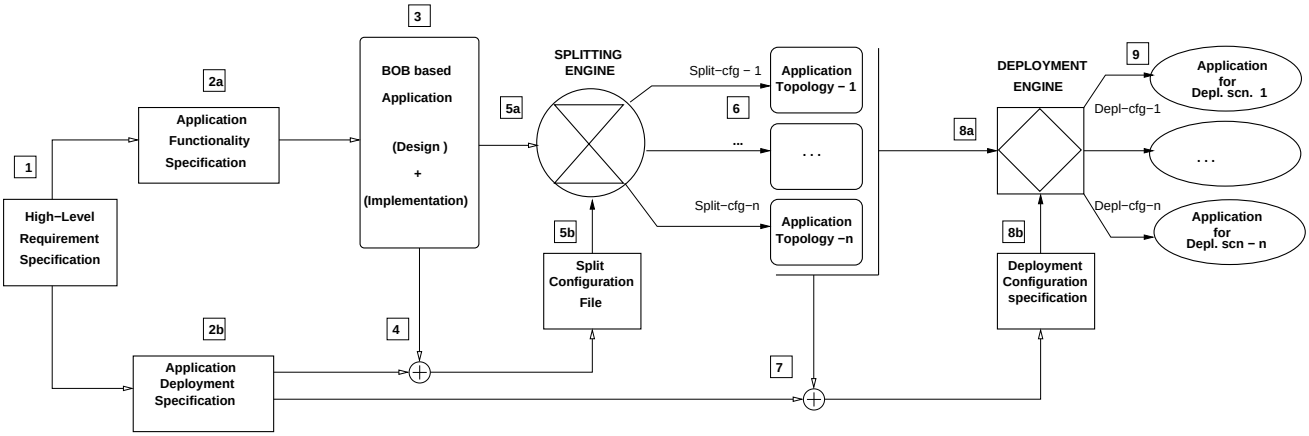


Figure 3.3: *BODA process stages for Automated Application Partitioning*

which should be defined as a breakable, is application specific and is based on designers understanding of the future deployment scenarios.

- For each BoB class an interface is defined. It consists of the public methods of the class and specification of *together* methods.

Further than this a BoB is treated as just another object in the program. Thus a *base-implementation* is done with no splitting on the BoBs.

3.4.2 Stage 2: Splitting and reorganization

In this stage (steps 4-6 , Figure 3.3), application functionality is factored into *specific subsets of objects*.

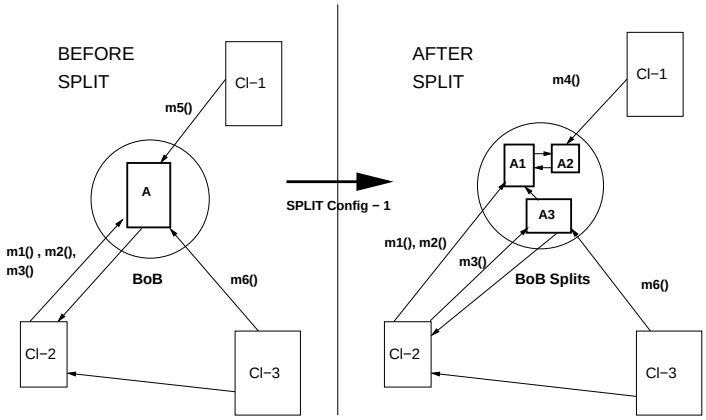


Figure 3.4: *Splitting and Client Reorganization*

- The functionality required on each node of the new scenario is determined. This functionality is mapped onto - (i) the normal objects in the application, and (ii) the interface methods of a BoB.
- For the given deployment scenario, the *BoB split-configurations* are worked out and specified for all the relevant BoBs.
- The BoBs are automatically split based on these specifications. The rest of program is reorganized to transform the original BoB references to references to BoB-splits.

For example, consider we have a program P with two BoB A, and B, and four other client classes $\mathsf{CL}-1, \mathsf{CL}-2,$ $\mathsf{CL}-3,$ and $\mathsf{CL}-4$, out of which $\mathsf{CL}-1, \mathsf{CL}-2,$ $\mathsf{CL}-3$ refer to interface methods of A (Figure 3.4). Suppose, we are required to split A based on a split-configuration a, we proceed as follows:

$$\mathsf{P} = \{\mathsf{A}, \mathsf{B}, \mathsf{CL}-1, \mathsf{CI}-2, \mathsf{CI}-3\}$$

Splitting BoB A:

$$\mathsf{A} \xrightarrow[(\mathsf{Split-Configuration-a})]{\otimes} \mathsf{A}_1 + \mathsf{A}_2 + \mathsf{A}_3$$

Client Reorganization

$$\mathsf{Reorganize} \rightarrow \{\mathsf{CL}-1, \mathsf{CL}-2, \mathsf{CL}-3\}$$

New program

$$\mathsf{P}' = \{\mathsf{A}_1, \mathsf{A}_3, \mathsf{B}, \mathsf{CL}-1', \mathsf{CL}-2', \mathsf{CL}-3', \mathsf{CL}-4\}$$

where, $\mathsf{CL}-1'$, $\mathsf{CL}-2'$, and $\mathsf{CL}-3'$ refer to the new reorganized clients, having the all references pertaining to A being translated to references to the splits of A.

Now, if we are required to split B too:

$$\mathsf{B} \xrightarrow[(\mathsf{Split-Configuration-b})]{\otimes} \mathsf{B}_1 + \mathsf{B}_2 + \cdots + \mathsf{B}_\mathsf{l}$$

Client Reorganization: Assuming $\mathsf{A}_2, \mathsf{CL}-2',$ and $\mathsf{CL}-4$ refers to B.

$$\mathsf{Reorganize} \rightarrow \{\mathsf{A}_2, \mathsf{CL}-2', \mathsf{CL}-4\}$$

New program

$$\mathsf{P}'' = \{\mathsf{A}_1, \mathsf{A}_2', \mathsf{A}_3, \cdots, \mathsf{A}_\mathsf{k}, \mathsf{B}_1, \mathsf{B}_2, \cdots, \mathsf{B}_\mathsf{l}, \mathsf{CL}-1', \mathsf{CL}-2'', \mathsf{CL}-3', \mathsf{CL}-4'\}$$

The final program transformations are:

$$\mathsf{P} \xrightarrow{\mathsf{splitA},\ \mathsf{reorg}} \mathsf{P}' \xrightarrow{\mathsf{splitB},\ \mathsf{reorg}} \mathsf{P}''$$

The final program obtained after a series of these transformations, is independent of the order in which BoBs are split. For example, in the above program if we had split B first and then A, i.e.,

$$\mathsf{P} \xrightarrow{\mathsf{splitB},\ \mathsf{reorg}} \mathsf{P}^{\sim} \xrightarrow{\mathsf{splitA},\ \mathsf{reorg}} \mathsf{P}^{\approx}$$

then following holds good :

$$\mathsf{P}'' = \mathsf{P}^{\approx}$$

3.4.3 Stage 3: Partitioning and Deployment Specific Transformations

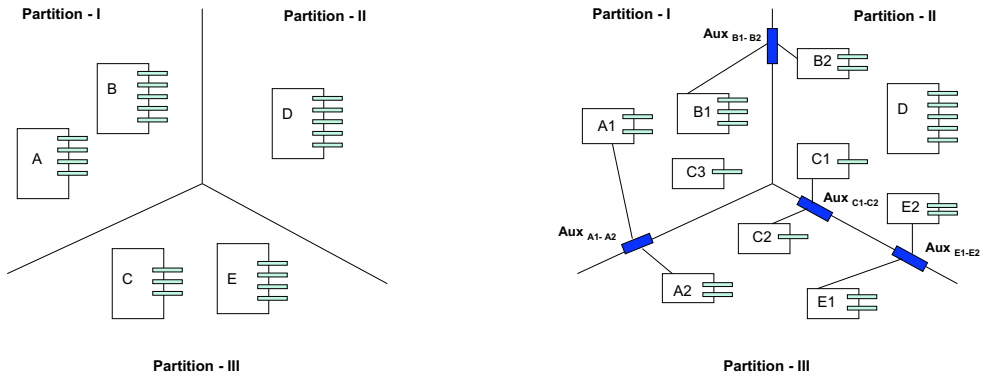
In stage 3 (steps 7-9, Figure 3.3), the application is *partitioned* and *deployable distributed components* for the new scenario are generated, distributed and deployed.

- **Partitioning**: Each application object is mapped to a node of the application deployment setup (*deployment-configuration* generation). A non-split program can be partitioned on the basis of individual BoBs. A further, fine grained partitioning is possible with BoBs. Wherever required, the split-fragments are partitioned across different partition spaces. These partitions can be exclusive or overlapping. In *exclusive* partitioning, split-fragments from a BoB do not have overlapping interface methods. This restriction is done away with in overlapping partitioning. For the purpose of this thesis, we consider only exclusive partitioning. The overlapping partitioning is a simple extension of exclusive partitioning process.

- **Deployment Specific Transformation** Components are prepared for these new distributed environments by doing source/binary level transformations on them.

- **Distribution** The resulting application components are distributed and deployed across these nodes.

For the application described in the figure 1.2, we construct two BoB classes: viz. `BoBFolder` and `BoBMessage`. Figure 8.2 shows the interface and split-configuration specifications for `BoBFolder`. It also shows the resulting node-specific splits for the



(a) **(b)**

Figure 3.5: *(a)BoB Partitioning, (b) BoB Split-fragment Partitioning*

disconnected deployment scenario. Figure 8.3 shows the distributions for the three scenarios of e-mail application. The example is further detailed in chapter **??**.

The main advantage that BODA offers is the separating out of *partitioning concerns* from the application design and implementation concerns. It achieves this through split and deployment configuration specifications, which are external to the objects and through automation of object refactorings.

3.5 BODA for Software Evolution

We would like software applications to be available to us in a manner that they can be easily decomposed, and the resulting artifacts *recombined* in new ways. For this purpose we reformulated the basic structuring entities in the form of BoBs. A BoB can be considered as an object (or a component), and hence provides us all the advantages of object-oriented (or component oriented) software development and reuse. Additionally, since a BoB functionality can be split into multiple forms though automated splitting, we can further use these split-fragments are new artifacts (or units) for reuse. This gives us the power of a flexible and fine-grained evolution of software systems.

We shall describe three ways in which a BoB based architecture supports program evolutions:

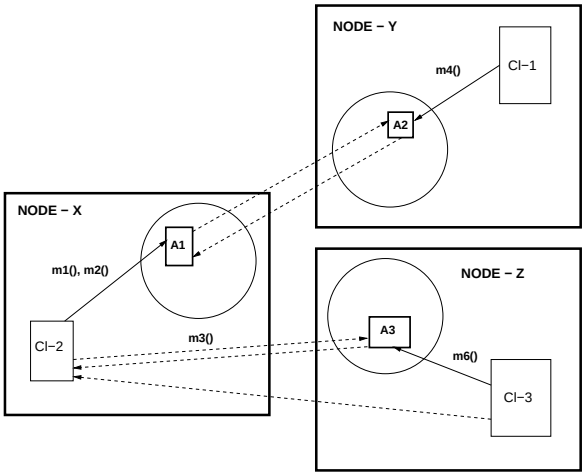


Figure 3.6: *Application redeployment*

3.5.1 BoB Program Modifications

The new requirements (or new concerns) for a software can come from a variety of sources like support for new concerns in the form of a new use-case (or scenario), a new *feature* or a new *task*. [Loh03].

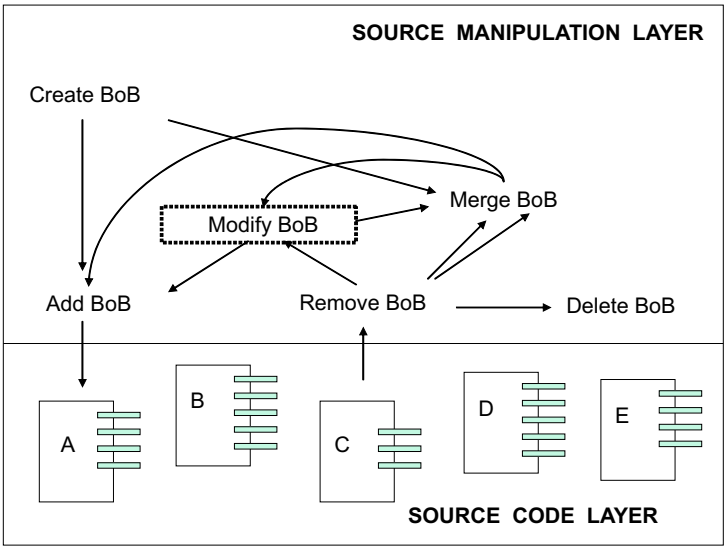


Figure 3.7: *BODA- BoB Program Modifications*

Whatever be the source, what is common is that a new requirement will require a change and modification to the existing software base. Whenever a new adaptation has to be made to a software base, the first task is to map these requirements to the artifacts that we use in a program. In a BoB based program, the primary artifacts are BoBs. So any new requirement will be mapped to:

- Addition of a component into the system.
- Deletion of component from the system.
- Modification of a component in the system.

Figure 3.7 describes this process. However, since BoB can be further decomposed, the paradigms supports two new mechanisms for program evolutions. We describe these in the next two sections.

3.5.2 Individual BoB Modifications

Instead of doing a white-box modification by making *invasive* changes to a BoB as a whole, BoB support a methodology for a more fine-grained black-box modification. Figure 3.8 explains this in detail. The process is two-three step:

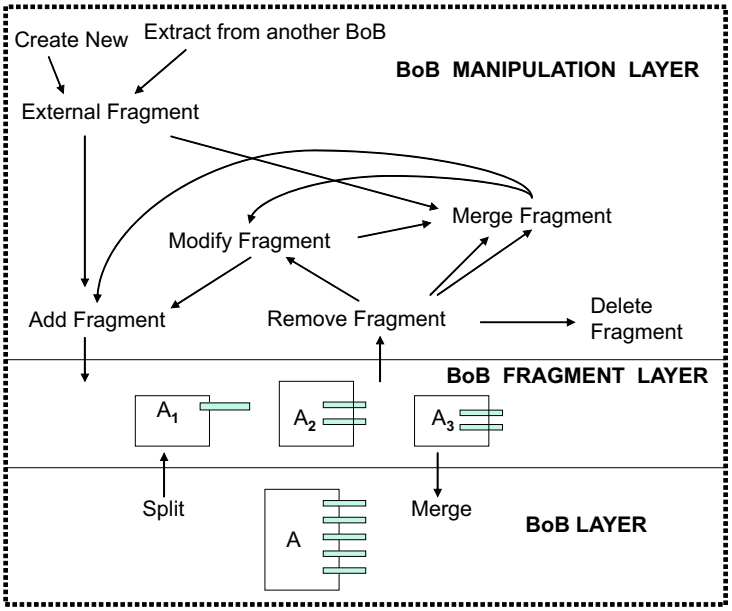


Figure 3.8: *BODA- Individual BoB Modifications*

1. Split the BoB on the desired lines.
2. Adapt fragments individually. Each fragment can now be considered as an artifact for modification and recombination. A new fragment can be added, and the old one deleted or modified. Similarly a fragment from one BoB can provide an element for reuse in an entirely different BoB.
3. Merge the new combination back into a BoB.

3.5.3 BoB Slice Modification

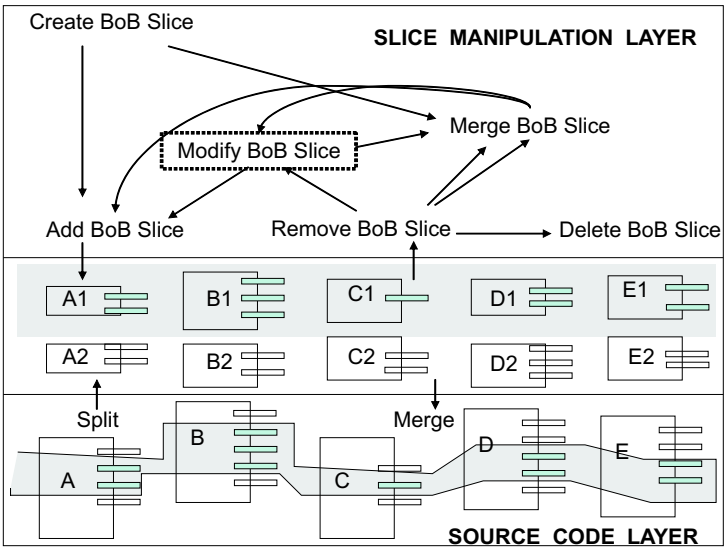


Figure 3.9: *BODA- Collaborating BoBs Slice Modifications*

The process of modification is similar to the ones described in previous two sections. The main difference is that instead of concerning ourselves with individual BoBs and BoB fragments, we take can take a whole *slice* across some BoBs and the whole of slice-fragment can now be considered as an element for adaptation and reuse Figure 3.9 . Such an approach is particularly useful when we are considering a *role-based collaboration* of classes [Van96b].

Summarizing, we can say that a BoB driven architecture of a software system, provides a fine-grained and systematic manner for partitioning and evolving the applications for adaptation to different concerns. In the rest of thesis, we elaborate on these mechanisms are provide case-study examples on their usage in object-oriented applications. The next chapter describes a model of BoBs based on object-oriented programming language Java. We shall use this model as a basis for developing our application-partitioning, BoB based composition and decomposition mechanisms.

I can't work without a model.

VINCENT VAN GOGH

Chapter 4

Programming Model

This chapter presents the preliminary model for Breakable Objects. We first build a very basic programming model, so that we can illustrate various BoB mechanism. More sophisticated features, will be added to this model, as we build BoB related concepts in the subsequent chapters .

We had the following three design guidelines for the BoB programming model: (a) it should match closely to a widely deployed language platform, (b) it should be based on a language which makes distributed computing relatively easier, and (c) it should minimize the inclusion of new constructs into the language.

To this effect, we chose Java as the base programming model. We introduce **only one new construct** into the language. The splitting engine generates the Java class definition files (.java files) from the BoB class definition files (.bob files). This enables BoBs to be used with existing Java Virtual Machines(JVMs) without any modification to the latter.

4.0.4 Programming Model

The programming model for **BoBs**, called Java$_{BoB}$ is based on the object oriented language Java. Table 4.1 presents the status of various constructs used in 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 clubbed methods. Figure 4.1 provides the schematics of a BoB class. Appendix A presents the formal description of a BoB Class.

Clubbed Methods

The designer of a BoB might intend that some of the methods should not be separated from each other during future splittings. We introduce a new programming

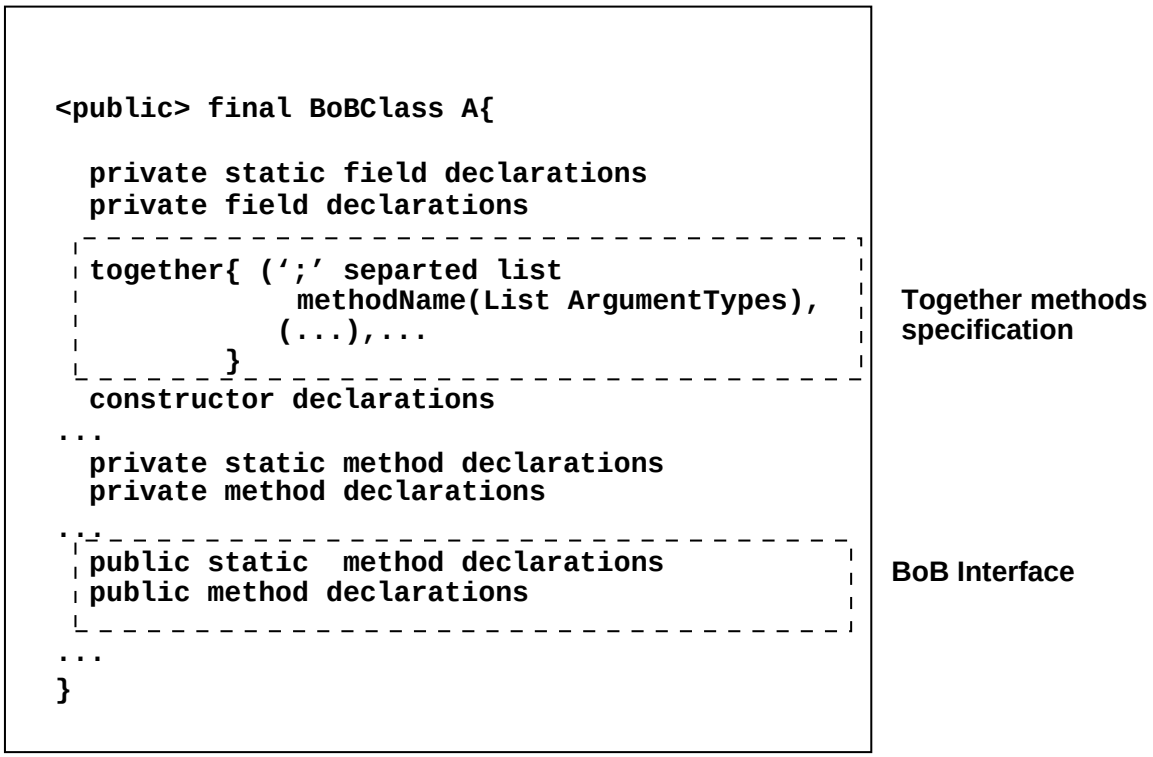


Figure 4.1: *Schematic of BoB class - A.bob*

construct named `together` to specify methods in a class that are inseparable. This is applied only to the interface methods. Also a number of such groupings can be done on the methods. For example, consider we have six interface methods `m1, m2, m3, m4, m5,` and `m6`. The designer might decide to create more than one clubbing, say `together` {`m1; m2`}, `together` {`m4; m5`}. If a method is part of more than one clubbing, the *union* of two sets will form a clubbed methods set.

The reason for clubbing methods together can by anything and is prerogative of the designer. For example, consider two synchronized methods in a BoB that access a shared field inside it. An inadvertent splitting might put these two methods in different splits leading to possible race conditions. Clubbing them together can prevent this.

Concepts Not Considered in this Model

Some of the constructs in present language have been disallowed[1] for the sake of simplicity while others have been removed because they pose difficulties in the construction of relevant BoBs. We discuss the implications of some of these restrictions

[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 disallowed features.

in later part of the thesis. The Java$_B$ differs from the Java as defined in language reference [Gos96] [Gos00] 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.

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. The interface inheritance can be used with one constraint that all the methods of a particular interface are designated as *together*. This to prevent the breaking of polymorphic references to the object after splitting.

Also each BoB is a *final* class. This means that it cannot be further extended.

Threading BoB are not active object [Agh90] [Lav95], that is they cannot run as separate threads. Also, since we do not allow inheritance, Java BoBs 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.

4.0.5 Format of the Configuration File

The format of configuration file is simple. It specifies the number of splits required and the signatures of the `public` methods of a BoB.

A valid configuration file satisfies the following properties:

- Only public methods are specified.
- Every public method in each BoB has to be specified as part of some split.
- A method cannot belong to more than one split.
- Clubbed methods (identified by the `together` construct) cannot be split.

The next chapter provides the details of the splitting engine.

Table 4.1: *Constructs for BoB Model - $Java_{BoB}$*

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

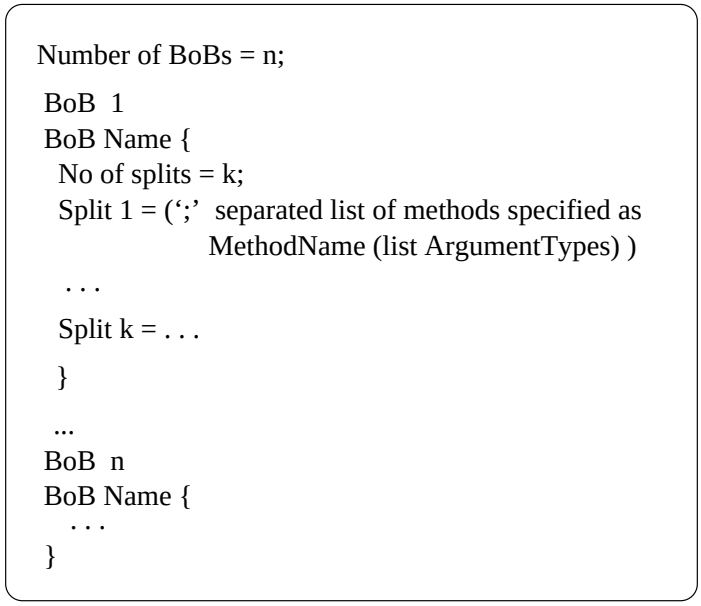


Figure 4.2: *Configuration file - split.conf*

Consciousness, we shall find, is reducible to relation between objects, and objects we shall find to be reducible to different states of consciousness..

T.S. (Thomas Sterns) Eliot

Chapter 5

Splitting Process

In this chapter, we present the algorithms used by the splitting engine (refer Figure 5.1). 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.

5.1 Main Algorithm

The splitting engine takes a specified BoB class, splits it and performs client reorganizations. For the example shown in Figure 5.1, it takes the BoB class `A.bob`, the client classes `Cl-1.java, Cl-2.java, Cl-3.java` and the `split.conf` file as input. It creates a internal dependency graph of the BoB class `A.bob` to capture the various interdependencies among the methods and fields of `A.bob`. Based on the specifications in the `split.conf` file and the internal dependency graph, it generates Java class definitions for the new splits `A1.java, A2.java, A3.java`. It 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`, as appropriate. This process is then repeated for all the specified BoB classes. This algorithm is described in Algo.1

5.1.1 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. Algo.2 constructs

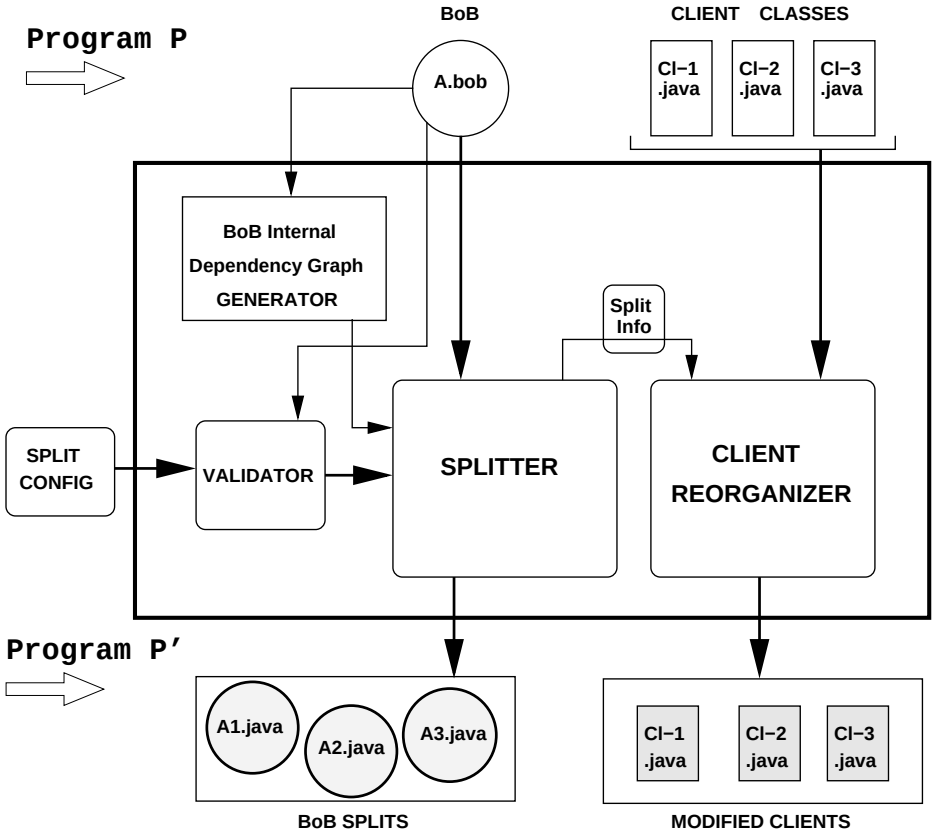


Figure 5.1: *Mechanisms of Splitting Engine*

Input: BoB and Java class files for $\mathbb{P}$, Split-Config file
Output: Java class files for $\mathbb{P}'$

```
1  foreach BoB class A ∈ Split-Config do
2      if (valid A ∧ valid split_cfg A) then
3          Prepare IDG for A {using Algo. 2 } ;
4          Split A {using Algo. 8 } ;
5          forall classes in P that refer A do
6              Reorganize {using Algo. 7};
7          end
8          Delete A;
9      end
10     else
11         report 'invalid splits'
12     end
13 end
```

1: **Splitting engine main algorithm**

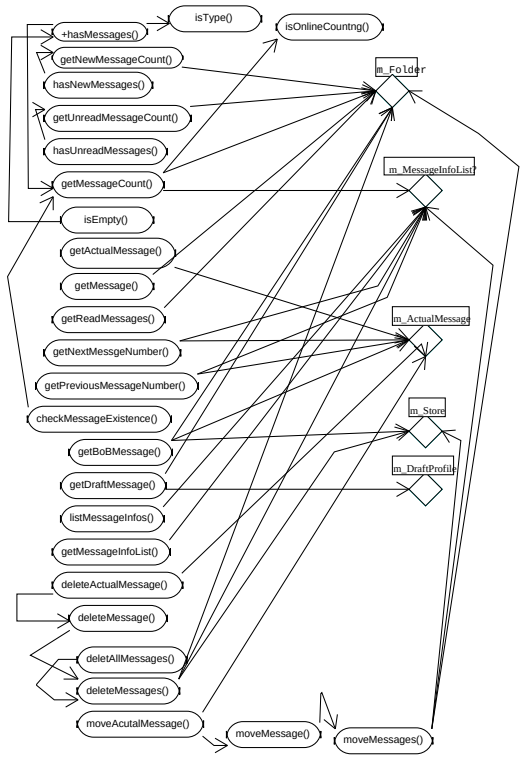


Figure 5.2: *IDG for BoBFolder (message handling portion)*

the BoB dependency graph. It is a directed graph. It has two types of nodes: field (F) nodes and method (M) nodes. We need to consider only those references, in a method, that form the fields of that class. Figure 8.1 presents the IDG for `BoBFolder`.

5.2 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 (Algo. 8) 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 5.1. It then includes all the methods that are specified in the configuration file into the respective split class files, referring to the Table 5.1 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.

Input: BoB class file A $\in \mathbb{P}$
Output: dependency-graph for A

1 Construct a graph $G = (V, E)$ where $v \in V$ is one of the following types:

- Field F Nodes {$\diamond$ nodes}
- Method M Nodes {$\circ$ nodes}

and $e(v_1, v_2) \in E$ is an directed edge $v_1 \rightarrow v_2$, where $v_1, v_2 \in V$;

2 **forall** *method m in the BoBClass C* **do**
3 | **foreach** *reference to field* $f \in C$ **do**
4 | | Construct a directional edge between m and f
5 | **end**
6 | **foreach** *invocation reference to method* m_{next} **do**
7 | | Construct an edge to referred node m_{next}
8 | **end**
9 | **foreach** *reference to class/object* $\notin$ *this class/object.field* **do**
10 | | Ignore
11 | **end**
12 **end**

2: **BoB internal dependency graph generation**

Table 5.1: *Construct Translations in Split Classfiles*

BoB construct	
Class	**Translation in splits**
`public`	apply `public` to all split class declarations
`final`	apply `final` to all split class declarations
`class`	apply `class` to all split class declarations

NameOfBoB	modified in the split classes in the following way:
	`NameOfBoBSplit_1`
	`NameOfBoBSplit_2`
	`. . .`
	`NameOfBoBSplit_k`

Field	**Translation in respective split fields**
`private`	apply `private`
`static`	apply `private`
`final`	apply `final`
`volatile`	apply `volatile`
Type	*Type* remains same as before
Name	*Name* remains same as before
Method	**Translation in respective split methods**
`public`	change to `private` if split interface does not contain this method; otherwise, apply `public`
`private`	apply `private`
`static`	apply `static`
`final`	apply `final`
`synchronized`	apply `synchronized`
`throws`	apply `throws`
Type	*Type* remains same as before
Name	*Name* remains same as before
Arg list	*Arg List* remains same as before

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 5.1}
  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

3: **Generating BoB class splits**

Procedure `MethodInclude(`M`)`

Input: Method Name M

```
50 if M not already included then
51     splitclass <-write- M ;
52     Refer BoB IDG. Check all outgoing edges from method node M ;
53     foreach outgoing edge do
54         if referred node is method M then
55             MethodInclude(M)
```

Procedure `FieldInclude(`F`)`

Input: Field Name F

```
56 if F not already included then
57     if F = constant field then
58         splitclass <-write- F ;
59     else if F = variable field then
60         Refer BoB IDG. Check all incoming edges to field node F;
61         if ∃ incoming edges from methods in other splits then
62             AUXclass <-write- F ;
63             AUXclass <-write- getter and setter for F;
64             if F is non static then
65                 splitclass <-write- refAUX ;
66
67
68         else
69             splitclass <-write- F ;
```

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

Input: Constructor Name $A(exps)$

```
70 forall (f ∈ original class) ∧ (f referred in A(exps)) do
71     if f ∉ this split then
72         Declare f local in beginning of constructor body with same initial value as
           in original class ;
73 if exist AUX class then
74     Include AUX in argument list A(exps);
75     Initialize refAUX with passed value in A(exps);
76 splitclass <-write- Asplit_k(exps) ;
```

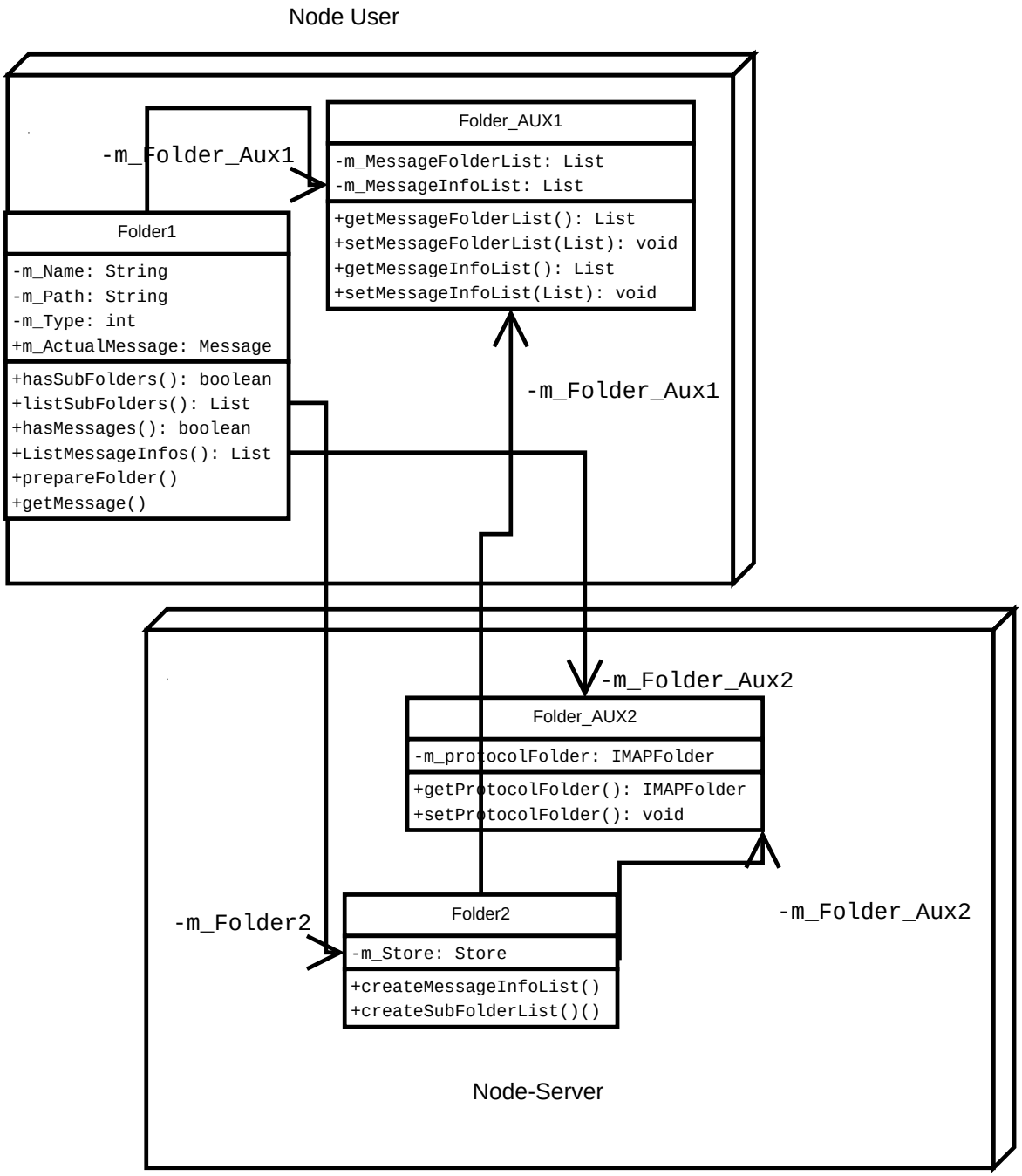


Figure 5.3: *Splitting and redeployment for* `BoBFolder`

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 separate auxiliary (AUX) split class file. *Constant* fields are replicated into all the splits that use them. Table 5.2 indicates various cases and the corresponding adopted strategies for the fields of a BoB class.

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.

Example: Figure 8.2 shows the splits performed on `BoBFolder` using FolderSplit$_{cfg}$ - 2. The figure 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 redeployments.

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 (as illustrated in the `BoBFolder` split example).

Table 5.2: *Effect of Splitting on BoB Fields*

	Updation ⟶	Constant				Variable			
	Scope ⟶	Static		Instance		Static		Instance	
	Visibility ⟶	Public	Private	Public	Private	Public	Private	Public	Private
Split-Field ↓	Independent	NA	SS	NA	SS	NA	SS	NA	SS
	Shared	NA	Repl	NA	Repl	NA	Sh Class	NA	Sh Obj

Repl: *Replicated* in all sharing splits, **Sh Class**: Present in a *shared class* accessible to all sharing splits,
Sh Obj: Present in a *shared object* accessible to all sharing splits, **SS** : Present only in a *single split* class/object,
NA: Case in not applicable to BoBs

5.3 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*. Algorithm (Algo. 7) considers the various scenarios and the corresponding modifications that are done in client codes. **Table 5.3** 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 Transformation **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 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.

It is to be noted that, access to a BoB is only through the public methods exported by it, and hence we need not consider references made to class or object fields. Also

some cases present in the Java language, are not considered in the reorganizer. For example, *polymorphic* references do not apply in case of BoBs (BoBs do not inherit or implement) and we do not consider *reflection* based class references.

Input: Class files Cl which refers to BoB
Output: Modified Class file Cl' which refers to BoB splits

```
1  switch FORM of statement do
2      case FORM = object declaration
3          – Declare variable for each split
4      case FORM = object creation
5          – Note constructor that is being called (compare argument types)
6          – Create all split objects by calling corresponding (similar argument types)
           constructor for each split
7          – If AUX split is present, then create AUX split first and pass its reference
           to other splits
8      case FORM = class method invocation
9          – Note M that object refers to
10         – Locate split class to which this M belongs
11         – Change class name to corresponding split class
12     case FORM = object method invocation
13         – Note m that object refers to
14         – Locate split class to which this m belongs
15         – Change object name to corresponding split object
16     case FORM = object assignment and aliasing
17         – For each variable assignment (or aliasing) of BoB object, expand to do the
           assignments (or aliasing) for all BoB splits
18         – In method arguments, for each passed parameter of BoB class, expand to
           pass all BoB splits
19     otherwise
20         – Report ERROR - outofscope -
```

7: **Client reorganization algorithm**

5.4 Restrictions on the Client-models

1. Return values - Side-effect free methods, or pass only as method parameters.

5.5 Restrctions for BoBs

1. Reflection based references. 2. this. reference.

Table 5.3: *Client Transformations*

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 = ref-BoBTypeA;	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)

5.6 A simplified illustrating example

Let us consider a simple case where we have a program consisting of two classes namely CalculatorServer and CalculatorClient. The server provides four arithmetic operations (addition, subtraction, multiplication and division) of integer numbers. The client instantiates an object of this server and then access its methods to perform the arithmetic operations

```
Server class = CalculatorSever.java

Class CalculatorServer {

    int ADD (int x, int y);
    int SUB (int x, int y);
    int MUL (int x, int y);
    int DIV (int x, int y);
}

Client class = CalculatorClient.java

Class CalculatorClient {
    CalculatorServer cs = null;

    CalculatorClient() {
        cs = new CaluclatorServer();
    }

   public static void main() {
        cs.MUL(4, 5);
        cs.SUB(3, 9);
   }
}
```

We provide a configuration file Calculator.splitconf which specifies the lines of split i.e. the combination of fields and methods that are required in the corresponding splits.

```
Configuration file: Calculator.splitconf

{
        No of splits = 4;

 Split 1 = Field (nil);
    Method (int Add(int, int));

 Split 2 = Field (nil);
    Method (int Sub(int, int));

 Split 3 = Field (nil);
   Method (int Mul(int, int));

 Split 4 = Field (nil);
   Method (int Div(int, int));
}
```

This implies that the Server should be split into four servers with each split serving a single arithmetic operation. The split server class files and the modified client class are shown below:

```
Split Class files:

Server Class 1 = CalculatorServerSplit_1.java

Class CalculatorServerSplit_1 {
   int ADD (int x, int y);
}

Server Class 2 = CalculatorServerSplit_2.java

Class CalculatorServerSplit_2 {
    int SUB  (int x, int y);
}

Server Class 3 = CalculatorServerSplit_3.java

Class CalculatorServerSplit_3 {
    int MUL (int x, int y);
}

Server Class 4 = CalculatorServerSplit_4.java

Class CalculatorServerSplit_4 {
    int DIV  (int x, int y);
}



Modified Client after split:

Class CalculatorClient {

    CalculatorServerSplit1 cs1 = null;
    CalculatorServerSplit1 cs2 = null;
    CalculatorServerSplit1 cs3 = null;
    CalculatorServerSplit1 cs4 = null;

    CalculatorClient() {
       CalculatorServerSplit1() cs1 =
                                    new CaluclatorSeverSplit1();

       CalculatorServerSplit1() cs2 =
                                    new CaluclatorSeverSplit1();

       CalculatorServerSplit1() cs3 =
                                   new CaluclatorSeverSplit1();

       CalculatorServerSplit1() cs4 =
                                   new CaluclatorSeverSplit1();
    }
```

```
    public static void main() {
              cs3.MUL(4, 5);
              cs2.SUB(3, 9);
    }


}
```

In the next chapter, we provide proofs for correctness of splitting process.

We cannot believe by proof; but could we believe without?

A.C.(Algernon Charles) Swinburne

Chapter 6

Proofs

We need to prove:

> Given a program $\mathbb{P}$ with a *BoBclass* C, if we replace C with splits $C_1, C_2, \cdots, C_k$, and reorganize client accesses from the rest of program to now refer to these splits, the new program $\mathbb{P}'$ so obtained, is equivalent to $\mathbb{P}$.

In this chapter we first evolve a definition of equivalence relevant to parallel lock-step execution of $\mathbb{P}$ and its split version $\mathbb{P}'$. Next we develop a proof methodology to prove this equivalence for all the relevant statement types in the program. In the later part, we devise these proofs based on the execution semantics of a Java$_{BoB}$ run.

We use the following definition (6.0.1) of *contextual equivalence* as described in [Pit00].This kind of program equivalence is also known as *operational* or *observational* equivalence.

Definition 6.0.1 (Contextual equivalence) *Two phrases of a programming language are* contextually equivalent *if any occurrences of the first phrase in a* ***complete program*** *can be replaced by the second phrase without affecting the* ***observable results*** *of executing the program.*

6.1 Equivalence of Split and Non-Split programs

Let us consider a program $\mathbb{P}$ having a set of classes and a BoB class as defined below:.

$$\mathbb{P} = C_1, C_2, ..., C_n,\ C_{BoB}$$

After being supplied with a configuration file C_{cfg} specifying the lines of split, we split C_{BoB} into k classes. Let $\mathbf{\Sigma}\ C_{BoBSplit_k}$ be the generated split-classes. We now do

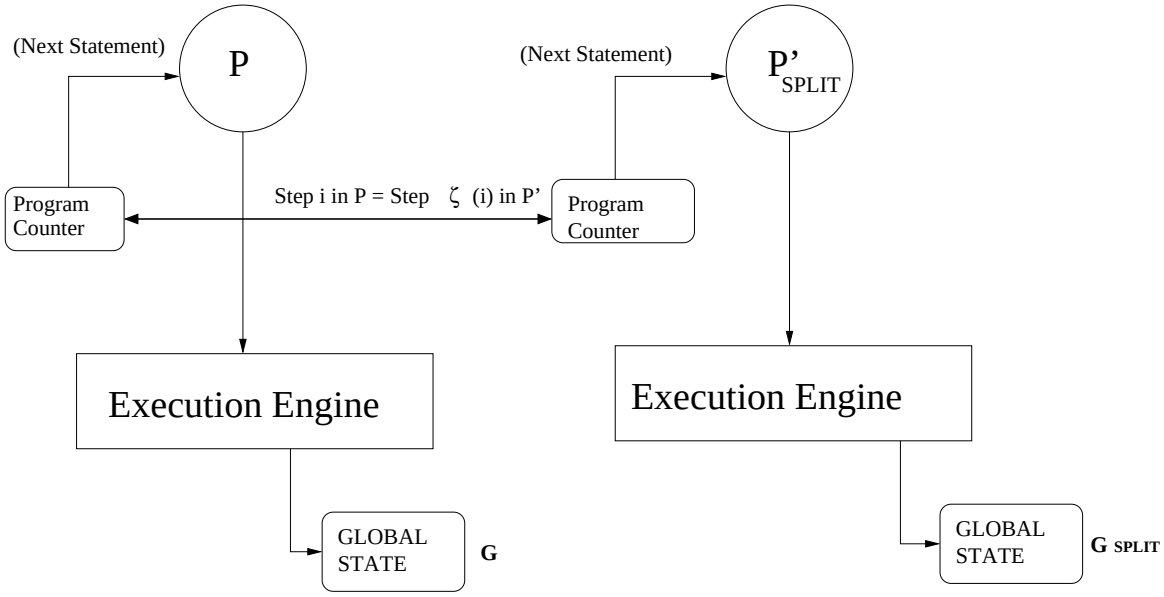


Figure 6.1: *Parallel lock-step runs of P and P'*

the client reorganizations to produce a transformed version for each client class file $C_1', C_2', ..., C_n'$. The new program $\mathbb{P}'$ thus obtained is defined as follows:

$$\mathbb{P}' = C_1', C_2', ..., C_n', \ \mathbf{\Sigma}\ C_{BoBSplit_k}$$

Consider an abstract execution engine which takes one program statement (or phrase) at a time and then executes it. The statement executions cause changes in the *global state* of the program. We specify the global state of the program as the *intended observed results* in the execution of a program.

We create a mechanism (refer Figure 6.1) for parallel lock-step runs of programs $\mathbb{P}$ and $\mathbb{P}'$. We use the mapping $n \mapsto \zeta(n)$ to denote the *corresponding steps* in the two runs, i.e., if n is the n^{th} step of execution in $\mathbb{P}$, then $\zeta(n)$ is the corresponding step in $\mathbb{P}'$. This mapping is not monotonic, i.e., one step in $\mathbb{P}$ may result in zero, one, or more steps in $\mathbb{P}'$. Following properties hold good: if $m \leq n$, then $\zeta(m) \leq \zeta(n)$.

Extended contextual equivalence

We extend the definition of *contextual equivalence*, as described in the previous section, to get a definition for describing the equivalence between pre-split ($\mathbb{P}$) and post-split ($\mathbb{P}'$) versions of the program. We term this new notion of equivalence as *extended contextual equivalence*.

Definition 6.1.1 (Extended contextual equivalence) *Two phrases of a program-*

ming language are extended contextually equivalent *if any occurrences of the first phrase in a* ***complete program*** $\mathbb{P}$ *can be replaced by the corresponding split-phrase in the* ***complete program*** $\mathbb{P}'$*, without affecting the* ***observable results*** *in the parallel runs of the programs.*

In order to show the extended contextual equivalence of $\mathbb{P}$ and $\mathbb{P}'$, we now define a transform τ on the global state.

6.2 The τ Transform

The τ transform is a function of the form $f : G \rightarrow G$ where G is the global state(refer 6.4.1) of a program. It takes the global state of an unsplit program, and with the help of the configuration file, transforms it into an equivalent global state of the split version of the program.

$\tau : G_{BoB} \xrightarrow{C_{cfg},\ Split-Algo} G_{BoBSplits}$

Methodology for τ transformation

- In global state of $\mathbb{P}$, we refer:
  each static field as:
  $ClassName \cdot FieldName$
  each object field as:
  $ClassName : ObjectName.FieldName$

- For a BoB class (with the help of split configuration file and split-algorithm) determine the corresponding fields and the splits to which they belong.

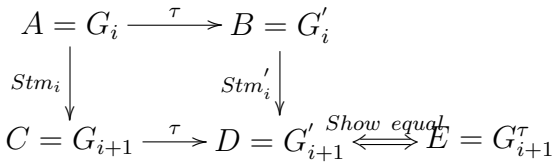
- In the global state of $\mathbb{P}$, replace each $ClassName$ of a field by its corresponding $SplitClassName$ and each $ObjectName$ of a field by its corresponding $SplitObjectName$ to get global state for $\mathbb{P}'$.

6.3 Methodology for Equivalence Proofs

The general methodology for equivalence proofs is *commutative diagram based* and is indicated in Figure 6.2. The steps involved in the proof process are described below:

1. We capture the global state G_i of the program $\mathbb{P}$ in the i^{th} step of its execution.

Figure 6.2: *Equivalence proof methodology*



2. We assume that the program $\mathbb{P}'$ has reached the corresponding step $\zeta(i)$ in a parallel run, and has a corresponding equivalent global state G_i'. This can be obtained by applying a transform τ on G_i.

3. By the execution of a program statement (Stm_i) in $\mathbb{P}$, we generate a new global state G_{i+1}.

4. A corresponding version of the Stm_i for $\zeta(i)$ step in $\mathbb{P}'$, Stm_i', is obtained by applying the appropriate split-transformations as specified in Table 5.3.

5. By the execution of program statement (Stm_i') in $\mathbb{P}'$, we generate a new global state G_{i+1}'.

6. From G_{i+1}, by applying the τ transform, we can generate G_{i+1}^{τ}, equivalent global state for $\mathbb{P}'$ in $\zeta(i+1)$.

7. Now, if we can show that $G_{i+1}' = G_{i+1}^{\tau}$, we have proved that the two program statements Stm_i in $\mathbb{P}$ and Stm_i' in $\mathbb{P}'$ are *extended contextually equivalent.*

The formal proof is based on the abstract execution model described in the next section.

6.4 Abstract Execution Model

We base our abstract machine model for execution of the Java$_{BoB}$ on the formalisms presented in [Bor98] [Stä01] which provides a system and machine independent definition of the semantics of the full programming language Java, as it is seen by the Java programmer. The definition is modular, given as a series of refined ASMs, dealing in succession with Java's imperative core, its object oriented features, exceptions and threads. For the sake of simplicity and without loss of any generality, we consider only the imperative (Java$_I$), class (Java$_C$) and object (Java$_O$) modules of the language for proof purposes [**?**].

6.4.1 Dynamic Global State

The Global State G of a program comprises the state of: (i) all classes (value of static fields), (ii) objects (value of instance fields) that exist at the time of state capture, and (iii) method call stack resulting from method-invocations during program executions (includes values of all local variables of the methods called).

Global State is defined for Java$_{BoB}$ ASM model as *dynamic state* of the program $\mathbb{P}$ given by the following functions: *pos, restbody, locals, meth, frames, classState, globals, heap* [**?**].

6.5 Main Theorem

Theorem 6.5.1 *Given a program $\mathbb{P}$ having a BoB class C and a split-configuration C_{cfg}, if the class C is split into $C_1, \cdots, C_k$, the new program $\mathbb{P}'$ is equivalent to $\mathbb{P}$.*

Before we go on to prove this theorem, we present some definitions and some lemmas which are used in the proof. we provide detailed proofs of these lemmas in the next section.

Definition 6.5.2 (Equivalent Methods) *Two methods are said to be equivalent* **iff** *(i) they have same signature, (ii) on being supplied same values in arguments, return the same value (iii) on being supplied same values in arguments, create the same side-effects on the rest of program.*

Lemma 6.5.3 *For every* **static** *method invocation m of a BoBClass C in program $\mathbb{P}$, there is always a corresponding* **static** *method invocation m' in program $\mathbb{P}'$ such that m and m' are equivalent methods.*

Lemma 6.5.4 *For every* **instance** *method invocation m of a BoBClass C in program $\mathbb{P}$, there is always a corresponding* **instance** *method invocation m' in program $\mathbb{P}'$ such that m and m' are equivalent methods.*

Lemma 6.5.5 *For every object O instantiated from BOBClass C in program $\mathbb{P}$, there is always a set of split-objects $\{O_{split\_1}, O_{split\_2}, \ldots, O_{split\_k}, \wedge O_{split\ \_AUX}\}$ instantiated in $\mathbb{P}'$ such that for every field in O, $O.f$, there is a corresponding equivalent variable in one of the splits $\{O_{split\_1} \vee O_{split\_2} \vee \ldots \vee O_{split\_k} \vee O_{split\ \_AUX}\}$.*

Proof for Main Theorem

PROOF SKETCH: We have to prove that by applying any statement Stm in $\mathbb{P}$ and applying an equivalent statement $Stm^{'}$ in split version of the program $\mathbb{P}^{'}$, the post-conditions obtained on the *GlobalState* are equivalent. For this we divide the programming statements into Java_I, Java_C, and Java_O.
PROVE: True

⟨1⟩1. for all Java_I Stm $\mathbb{P} \equiv \mathbb{P}^{'}$

PROOF: No splitting for Java_I.

⟨1⟩2. for all Java_C Stm $\mathbb{P} \equiv \mathbb{P}^{'}$

PROOF: by Lemma 6.5.3

⟨1⟩3. for all Java_O Stm $\mathbb{P} \equiv \mathbb{P}^{'}$

PROOF: by Lemmas 6.5.4, and 6.5.5

⟨1⟩4. Q.E.D.

PROOF: by ⟨1⟩1, ⟨1⟩2, ⟨1⟩3

□.

6.6 Lemma Proof Details

Lemma 6.6.1 *For every* **static** *method invocation m of a BoBClass C in program* $\mathbb{P}$*, there is always a corresponding* **static** *method invocation* $m^{'}$ *in program* $\mathbb{P}^{'}$ *such that m and* $m^{'}$ *are equivalent methods.*

PROVE: True. OR. methods return the same value, and produce the same side-affects

⟨1⟩1. m and $m^{'}$ have same arguments

PROOF: By Split-Algo

⟨1⟩2. m and $m^{'}$ have same return type

PROOF: By Split-Algo

⟨1⟩3. m and $m^{'}$ have same method body statements $\forall$ $Java_I$ statements

PROOF: By Split-Algo

⟨1⟩4. m and $m^{'}$ have same method body statements for class field references and class method invocations for classes outside the splits

PROOF: By Split-Algo

⟨1⟩5. references in m and $m^{'}$ to the fields within the splits produce similar results

PROVE: references are equivalent variable references

The variables referred by $m^{'}$ are either in the split class containing $m^{'}$ or in the AUX split class.

CASE: Variables within the split class containing $m^{'}$

References remain unchanged. By Split-Algo Initial values are same (see next step) and references are from equivalent method body (equivalent invocations in same the same order).

CASE: Variable in the AUX

References are modified to $BoBClassC_{AUX}.f$ By Split-Algo Initial values are same (see next step) and references are from equivalent method body (equivalent invocations in same the same order).

$\langle 1 \rangle 6$. preconditions for m and $m^{'}$are same

By assumption that this is the first method invocation on split class. So the preceding program is either of type $Java_I$ or the method invocations and field references are the unsplit classes. For the rest of invocations, post-conditions for the first invocation become the pre-conditions for the rest of program)

$\langle 1 \rangle 7$. Q.E.D.

PROOF: $\langle 1 \rangle 1$, $\langle 1 \rangle 2$, $\langle 1 \rangle 3$, $\langle 1 \rangle 4$, and $\langle 1 \rangle 5$

□.

Lemma 6.6.2 *For every* **instance** *method invocation m of a BoBClass C in program $\mathbb{P}$, there is always a corresponding* **instance** *method invocation $m^{'}$ in program $\mathbb{P}^{'}$ such that m and $m^{'}$ are equivalent methods.*

PROOF: On same lines as Lemma 6.5.3, except that method invoked is an instance method as shown below:

$$
\begin{aligned}
{}^{\alpha}ref \cdot c/m^{\blacktriangleright}(vals) \quad & \rightarrow \textbf{if} \quad ref \neq null \quad \textbf{then} \\
& \quad \textbf{let} \quad c^{'} = \textbf{case} \\
& \quad callKind(up(pos)) \quad \textbf{of} \\
& invokeMethod(up(pos), c^{'}/m, \\
& \qquad [ref] \cdot vals)
\end{aligned}
$$

□.

Lemma 6.6.3 *For every object O instantiated from BOBClass C in program $\mathbb{P}$, there is always a set of split-objects $\{O_{split\_1}, O_{split\_2}, \ldots, O_{split\_k}, \wedge O_{split\ \_AUX}\}$ instantiated*

in $\mathbb{P}'$ such that for every field in O, O.f, there is a corresponding equivalent variable in one of the splits $\{O_{split\_1} \vee O_{split\_2} \vee \ldots \vee O_{split\_k} \vee O_{split\ \_AUX}\}$.

⟨1⟩1. Every BoB creation in $\mathbb{P}$ is translated to BoB split creation in $\mathbb{P}'$

PROOF: By Client-Reorg Algo and Transformation T2

⟨1⟩2. Each BoB field has a corresponding BoB split field

PROOF: By Split-Algo and Transformation T2

⟨1⟩3. Instance creation methods are treated like ordinary method invocations. An instance creation expression `new` *C(exps)* is transformed at compile time into the abstract form (`new` C) . C misg(exps), where *misg* is the name of a constructor of class C with name `<init>`. The abstract expression (`new` C) creates a new reference to an instance of class C which is the target reference of the invoked constructor

- for $\mathbb{P}$, we get:

$$
\begin{array}{lll}
\texttt{new} \quad c \rightarrow & \textbf{if} & initialized(c) \quad \textbf{then} \\
 & & \quad create \quad ref \\
 & & \quad heap(ref) := Object(c, \\
 & & \qquad \{(f, defaultVal(type(f))) \\
 & & \qquad | f \in instanceFields(c)\}) \\
 & & \quad yield(ref) \\
 & \textbf{else} & initialize(c)
\end{array}
$$

Difference $\mathfrak{U}$ of states =
$\{(c, f, defaultVal(type(f)) | f \in instanceFields(c))\}$

- for $\mathbb{P}'$, we get:

$$\texttt{new}\quad c_{splitk} \rightarrow \quad \textbf{if}\quad initialized(c_{splitk})\quad \textbf{then}$$
$$create\quad ref$$
$$heap(ref) := Object(c_{splitk},$$
$$\{(f, defaultVal(type(f)))$$
$$|f \in instanceFields(c_{splitk})\})$$
$$yield(ref)$$
$$\textbf{else}\quad initialize(c_{splitk})$$

Difference $\mathfrak{U}'$ of states =
$\{(c_{split_1}, f, defaultVal(type(f)))|$
$f \in instanceFields(c_{split_1})\}, \bigcup \ldots,$

$\{(c_{split_k}, f, defaultVal(type(f)))|$
$f \in instanceFields(c_{split_k})\}, \bigcup$

$\{(c_{split_{AUX}}, f, defaultVal(type(f)))|$
$f \in instanceFields(c_{split_{AUX}})\}$

⟨2⟩1. $\mathfrak{U}$ and $\mathfrak{U}'$ are **Isomorphic**

PROOF: Let $\theta : ClassName_{split\_number} \rightarrow ClassName$ be an isomorphism from $\mathfrak{U}'$ to $\mathfrak{U}$. Isomorphic states property is satisfied

⟨2⟩2. Constructors (`new` C) . `<init>` / misg(exps) and (`new` C_split_number). `<init>` / misg(exps) cause equivalent state changes

PROOF: By Split-Algo each constructor has same signature and method body except that the fields not present in that particular split are suppressed as local variables in the respective constructors. The updates will reflect as (`new` C) . C / f, value change
$defaultVal \rightarrow constructorVal$ and the corresponding (`new` C_split_split) . C_split_k/f, value change
$defaultVal \rightarrow constructorVal$.

⟨1⟩4. preconditions for m and m' are same

PROOF: Same as first leema $< 1 > 6$.

⟨1⟩5. Q.E.D.

PROOF: ⟨1⟩1, ⟨1⟩2, ⟨1⟩3, ⟨1⟩4 .

□.

In the next section, we extend the BoB program transformation primitives to include the *merging* of BoBs which have been split. We shall also discuss the implementation details of splitting and merging engines, and those of deployment engine (based on two existing partitioning engines). We also discuss issues related to inheritance in BoB application partitioning.

The need to become a separate self is as urgent as the yearning to merge forever.

JUDITH VIORST

Chapter 7

Merging, Redeployments, and Implementation Details

In this chapter, we present the algorithms used by the merging engine (refer Figure 7.1). A set of BoB fragment classes forms the input. The BoB class formed by merging these fragments is the output of merging engine.

7.1 Merging Algorithm

Merging algorithm is described in Algo.8.

Input: Split Class Files `A_split_1.java,...,A_split_k.java & A_split_AUX.java`

Output: BoBClass File `A.bob`

Create a class source file MergedClass; Name it as: `A.bob`;
MergedClass $\xleftarrow{write}$ $\bigcup$ { $import_{stm}$ $\in$ split class in $Merge_{cfg}$ }
MergedClass $\xleftarrow{write}$ class constructs in any split + "BoBClass" + A
MergedClass $\xleftarrow{write}$ ' { '
forall *splits specified in* $Merge_{cfg}$ **do**
  foreach *method m in SplitClass excluding AUXclass* **do**
    `MergeMethodInclude` (m) ;
  foreach *field f in SplitClass* **do**
    `MergeFieldInclude` (f);
  foreach *constructor c in SplitClass* **do**
    `MergeConstructorInclude` (c);
NewClass $\xleftarrow{write}$ ' } '

8: **Merging BoB class splits**

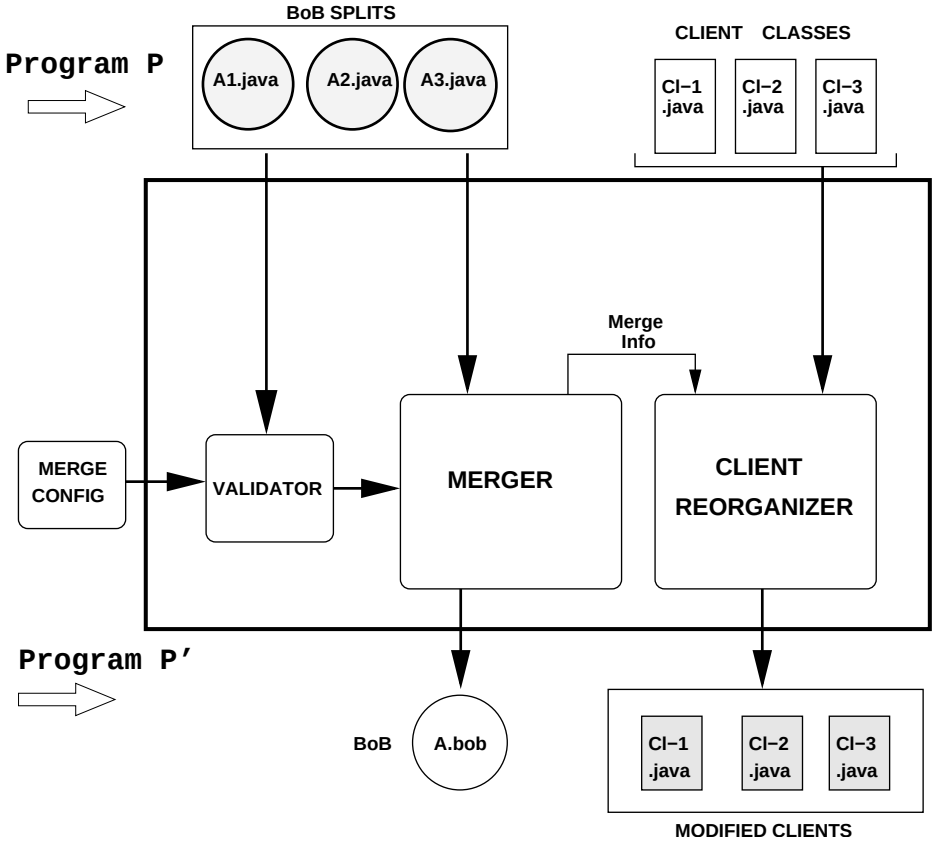


Figure 7.1: *Mechanisms of Merging Engine*

Procedure `MergeFieldInclude(`F`)`

Input: Field Name F

if *F not already included* **then**
  NewClass $\xleftarrow{write}$ F ;

Procedure `MergeMethodInclude(`M`)`

Input: Method Name M

if *M not already included* **then**
  NewClass $\xleftarrow{write}$ M ;
Refer split BoB IDG.
forall *outgoing edges from method node M* **do**
  if *referred node is AUXclass* **then**
    Change the reference to corresponding accessed field in this
if *method* $M \in \bigcup \{$ *interface methods* $\forall$ *splits in* $Merge_{cfg}$ $\}$ **then**
  if *access of* $M =$ `private` **then**
    Make (access = `public`)

Procedure `MergeConstructorInclude(`*C(exps)*`)`

Input: Constructor Name *C*

if *C (exps) not already included* **then**
- NewClass $\xleftarrow{write}$ C (exps) ;
- **forall** *outgoing edges from constructor C* **do**
  - **if** *refereed node is AUXclass* **then**
    - Change the reference to corresponding accessed field in this.
- ∀ local variables declared in the beginning of constructor (after super(exps) or this(exps) call), having the same name as the class fields, remove the local variable declarations.

7.1.1 Rules for merging fragments

- Validation - for Fields: Matching field names should have same type
- Validation - for Methods: Matching methods should have same method bodies (ignore references to AUX or convert AUX reference to field references)
- Validation - for Constructor : Constructors signatures should match. If optimization has taken place on the constructors, the signatures are expanded by concatenation

7.2 Handling Inheritance

Following sections illustrate the issues related to *BoB Class inheritance*

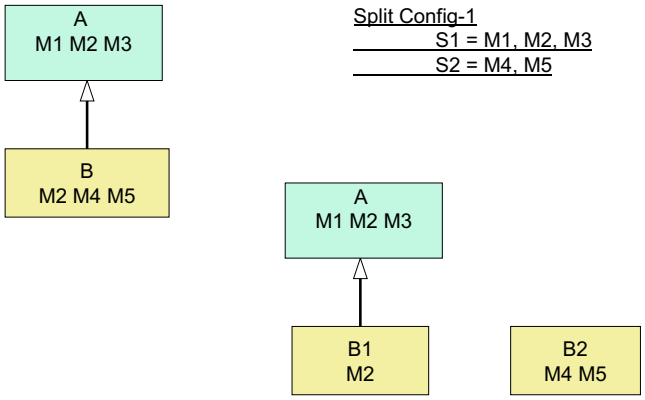


Figure 7.2: *Inheritance-issues (a)*

Consider a BoB A. Figure 7.2 shows the splitting for a given configuration. The parent is not affected by the partitioning. However, in Figure 7.2 we have a case where the parent is also partitioned. This can pose problems as we discuss in next section.

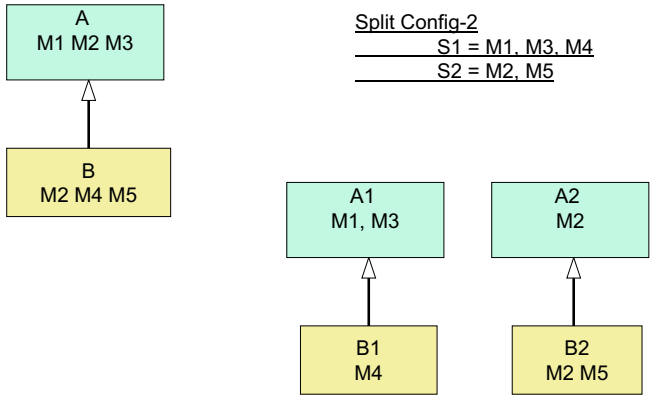


Figure 7.3: *Inheritance-issues (b)*

7.2.1 Issue: Retaining old type

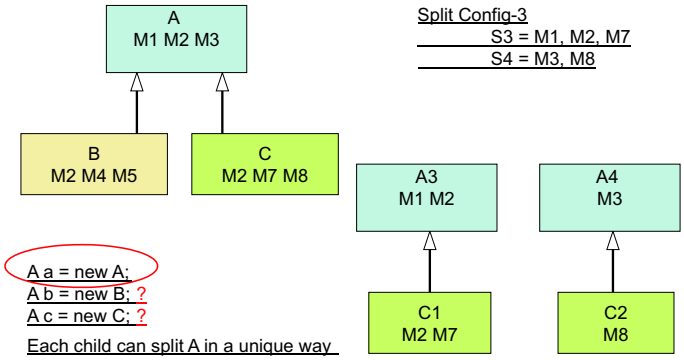


Figure 7.4: *Inheritance-issues-(c)*

Consider the splitting in Figure 7.4. When BoB C is partitioned, it partitions A in a different manner than that of BoB B partitioning in the previous example. This leads to problems as shown in the example.

So any partitioning introduces new types into the system. The issue is what should be done with the old type (A in this case). We have two options:

- Retain old type. That is, for Figure 7.2 A remains along with A1, A2.

- Delete old type. Only A1 and A2 remain.

We prefer case 2, and allow only interface inheritance with the condition that all the methods of an interface are designated `together`. W recommended aggregation (and delegation) as the principal composition mechanisms for BoBs when used in the context of partitioning. It leads to neater design and reduces complexity of restructuring.

7.2.2 Class-level/object level

We do class-level partitioning. Object level partitioning creates more complexity, particularly when used in the context of class based object oriented languages.

If we allow object-level partitioning, it implies that we allow a BoB to be split in more than one way in application. This lead to the same kind of problems as discussed in the above section.

Consider, for example, the assignment operation. We need to know the ¡type¿ of object on RHS and then convert it to the ¡type¿ being assigned (LHS)

A ax = new A();
A ay = new A();
ax split: ax1 = m1, m2, split ax2 = m3
ay split: ay1 = m2, split ay2 = m1, m3;

If we have following assignment operation, ax = ay it is cumbersome to do this translations once splitting is done .

We need run-time support for such an operation. Such a support is not available in the present languages. We use *class-level* BoB partitioning, and it is sufficient and meaningful for most applications.

7.3 Implementation - Splitting and Merging Engines

The implementation of the splitting and merging engines has been done using Inject/J [Gen03], which provides support for large-scale source meta-programming[Lud00]. It is based on Recorder/Java[1] transformation library. It can be used to transform Java program. The input is a Java program and the output is a modified Java program. Inject/J language is dynamically typed and provides transformation operators

[1] `http://recoder.sourceforge.net/index.html`

in terms of model entities. Its allows to navigate these model entities (e.g. a class, a method etc.), select model entities and perform transformational operations on them. Inject/J software then processes transformations scripts written in Inject/J software transformation language. Both the splitting and merging engines make heavy use of these scripts. There, are however, certain limitations in the language, e.g. it does not allow i/o operations, and at present can read in only Java source code files. This makes it difficult for us to specify e.g. a xml based configuration file. We circumvent this by using Java interface definitions for specifying the split and merge configurations. Extending the Inject/J language platform for our specific and customized use is an area for future work.

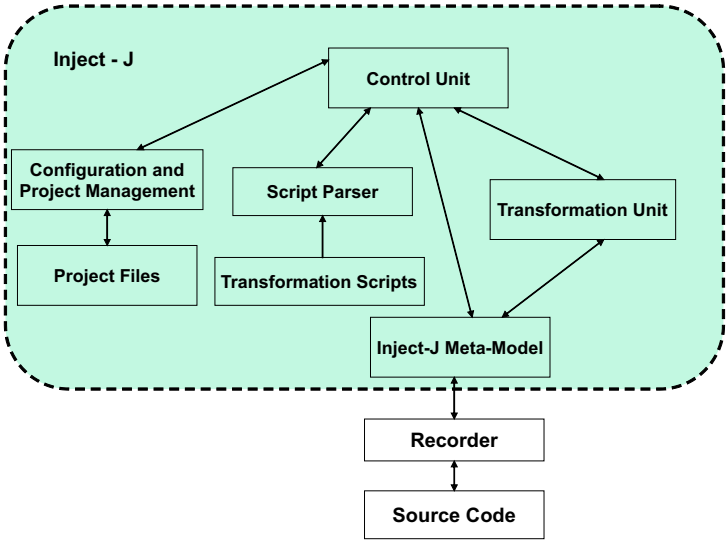


Figure 7.5: *Inject-J Architecture*

7.4 Deployment Architecture

This chapter provides the details of deployment Engine. The deployment engine currently uses mechanisms developed by Pangaea [Spi99] and J-orchestra [Til02] for BoB deployments. In the former case, the distribution transformations are done on the source code and the compilation is done at the subsequent stage, while in the latter case the distribution is done after compilation of the source code into the Java byte code. Below we explain the constitution of BoB-deployment engine based on both these techniques.

7.4.1 Deployment using Pangaea

Pangaea is a system that can distribute centralized Java programs, based on static source code analysis and using arbitrary distribution platforms as a back-end. It thus distributes programs automatically, in the following four senses:

- Pangaea automatically analyzes the source code of a program, estimating its run-time structure and determining distribution-relevant properties.

- Based on this information, the distribution strategy for the program can be decided upon. This involves where to place objects, and when to employ object migration. To specify the distribution strategy, the programmer is provided with a graphical view of the program, in which he may configure the program for distributed execution, assisted by automatic tools.

- After the program has been configured, the desired distribution strategy is implemented automatically by regenerating the program's source code for a chosen distribution platform.

- At run-time, the placement of some or all of the program's objects can be adjusted automatically based on their communication behavior. This is realized by an asynchronous migration facility supplied by Pangaea's run-time system.

Pangaea provides support for three back-end code generators, viz., CORBA, JavaParty, and Doorastha. For our purpose we have considered only Doorastha.

The various steps involved in distribution through Pangaea (see figure 7.6) system are enumerated below:

1. The first step is source code analysis. The source code analysis algorithm for Pangaea provides an object-oriented, macroscopic view of the program's run-time structure that is suitable for further distribution analysis.

2. Next step is specifying the distribution strategy. Pangaea's graphical user interface, provides a useful tool by which the programmer can specify distribution requirements. The act of specifying a distribution strategy for a program is called configuring the program. This involves where to place individual objects, which objects may be kept local on each machine, and for which objects asynchronous object migration should be employed. The programmer specifies the configuration in Pangaea's graphical user interface, assisted by automatic tools.

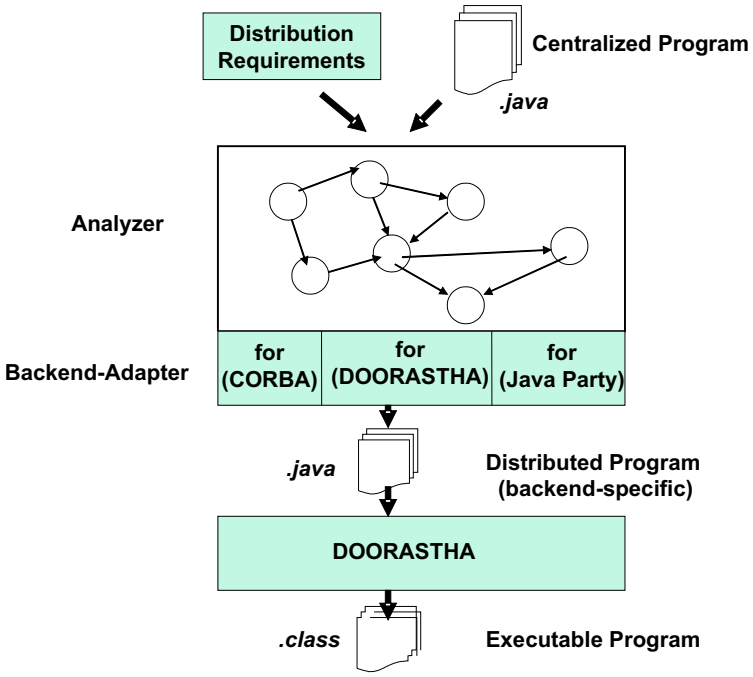


Figure 7.6: *Pangaea Architecture*

3. After the configuration is complete, the types of the program are classified, which means, for example, that some types must be remotely invokable, others migrable, etc. The classification of the program's types is an abstract way of describing the implementation of the chosen distribution strategy. Classification is carried out automatically by Pangaea and forms the basis for the subsequent code generation step.

4. Pangaea's back-end code generators regenerate the source code of the program, transforming it into a distributed program for the chosen distribution platform, while preserving the centralized program's semantics. This includes making some classes remotely invokable, turning some object allocation statements into remote allocations, and other transformations. There are back-end code generators for CORBA, JavaParty, and Doorastha.

5. Pangaea provides a run-time system in addition to that of the distribution platform. Its main purpose is to implement asynchronous object migration, which means that objects are monitored at run-time and moved to the partition from which they are most frequently accessed.

In addition, Pangaea provides a small Launcher utility that hides these differences and complexities of the distribution platforms , so that executing a distributed pro-

gram becomes very similar to the centralized case.

7.4.2 Deployment using J-orchestra

J-Orchestra operates at the Java bytecode level. It rewrites the application code to replace local data exchange (function calls, data sharing through pointers) with remote communication (remote function calls through Java RMI, indirect pointers to mobile objects).

J-Orchestra receives input from the user specifying the network locations of various hardware and software resources and the code using them directly. A separate profiling phase and static analysis are used to automatically compute a partitioning that minimizes network traffic. It specifies the deployment node for each component of the system. Figure 7.7 gives a high-level overview of the operation of J-Orchestra.

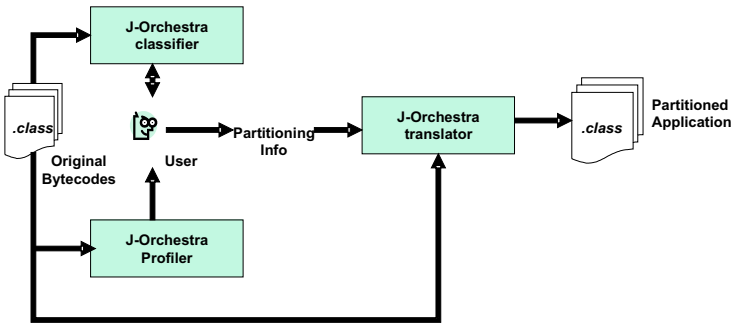


Figure 7.7: *Jorchestra Architecture*

The various features of the J-orchestra system are discussed below:

1. The user interaction with the J-Orchestra system consists of specifying the mobility properties and location of application objects.

2. J-Orchestra converts all objects of an application into remote-capable objects, i.e., objects that can be accessed from a remote site.

3. Remote-capable objects can be either anchored (i.e., they cannot move from their location) or mobile (i.e., they can migrate at will). For every class in the original application, or Java system class potentially used by application code, the user can specify whether the class instances will be mobile or anchored.

4. For mobile classes, the user needs to also describe a migration policy - specification of when the objects should migrate and how. For anchored classes, the user needs to specify their location.

5. Using this input, the J-Orchestra translator modifies the original application and system byte code, creates new binary packages, produces source code for helper classes (proxies, etc.), compiles that source code, and creates the final distributed application.

To ensure a correct and efficient partitioning, J-Orchestra offers two tools: a *profiler* and a *classifier*. The profiler reports to the user statistics on the interdependencies- of various classes based on (off-line) profiled runs of the application. With this information, the user can decide which classes should be anchored together and where. The J-Orchestra classification algorithm is responsible for ensuring the correctness of the user-chosen partitioning. The classifier analyzes classes to find any dependencies that can prevent them from being fully mobile. The classifier takes one or more classes and their desired locations as input and computes whether they can be mobile and, if not, whether the suggested locations are legal and what other classes should be co-anchored on the same sites. The user interacts with the classifier until all system classes have been anchored correctly.

7.5 Summary and Discussion

The main aim of providing distribution capability to a BoB is to facilitate optimal placement of distributed components. Optimal placement of BoB components is beyond the scope of this thesis. For our purposes we assume that a mechanism exists - manual or automatic, which determine the suitable location for each of the application components. Partitioning systems, as discussed above, provide provide these mechanisms. For BoBs, the information for placement of sub-components is specified externally and is assumed to be available.

We have mainly looked at compile time distribution of BoBs. Deployment time distributiondistribution of BoBs is an area of future research.

Always design a thing by considering it in its next larger context - a chair in a room, a room in a house, a house in an environment, an environment in a city plan.

Eliel Saarinen

Chapter 8

Case-Studies-1

8.1 Multi-mode E-mail Client Application

We describe here briefly the e-mail client application motivated in the introduction and which we has been designed as a BoB based application. Figure 1.2 shows the class layout view of the e-mail application. We mention here only those classes which are relevant for the purpose of illustrating the concepts and exclude all other details that are used in actual implementation such as the graphical user interfaces classes, session class, mime-helper class utilities, text processing utilities, etc. We had identified two objects for BoB implementation were, viz., BoBFolder and BoBMessage.

BoBFolder: `BoBFolder` builds as a container of `Folder` class provided by `javax.mail` package. All the connection and protocol handling part is done by the protocol specific implementation class (e.g. `IMAPFolder`). `BoBFolder` provides the various utilities to the user interface and the rest of the program through a series of public methods (refer figure 8.2). It also supports a mechanism to create a *cache* for the *sub-folders* and *message-infos* of the acquired messages.

BoBMessage: It carries the actual *e-mail message* which can be single or multi part. We only do simple splitting on `BoBMessage` to separate out the information, content and attached parts. Depending upon the need, the requisite part is fetched, e.g., if we are only displaying the list of mails in a folder, we create a list of message information parts.

We use the `BoBFolder` example for further illustration of the BDA process. Figure 8.1 presents the IDG for `BoBFolder`. Figure 8.2 shows the splits performed on `BoBFolder` using FolderSplit$_{cfg}$ - 2. The figure also shows the redeployment done

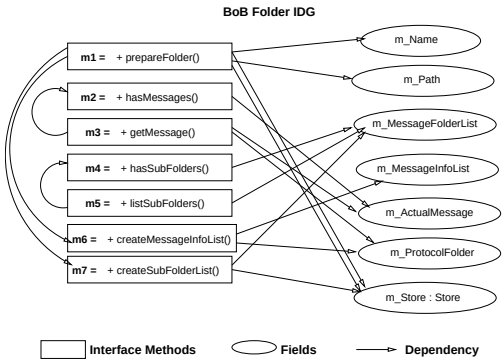


Figure 8.1: *IDG for BoBFolder*

on `BoBFolder` splits. The AUX class obtained after splitting is further split into $AUX1$ and $AUX2$ for the purpose of redeployment.

Figure 8.3 describes different configurations of principal classes for the different versions - *online, disconnected and offline* - obtained for the e-mail application, implemented using these BoBs.

8.2 Small (Mobile) Device Application Architectures

An important aspect of software is that they should small in size. This is particularly important when we have devices which are resource constrained.

Classes with contain a lot of functionality which lies untilized add to the overall bulk of program size. Consider the following overvation by a game developer Kyle Wislon:

> ...More troubling is all the functional code that's left lying around never to be executed. It bloats the executable, it makes it harder to find the function you're looking for, and it leaves everyone unsure when to invest effort in making sure something still works and when to just let it lie. Eventually, code will become so bloated and unwieldy that it's impossible to maintain, and everyone on the team will start talking about how much things need to be rewritten from scratch. ...

Application on mobile devices have to run in a limited environment. The primary limitation is amount of available memory to run and store applications. For example, in MIDP devices running J2ME applications, the application size is 50K or less. This

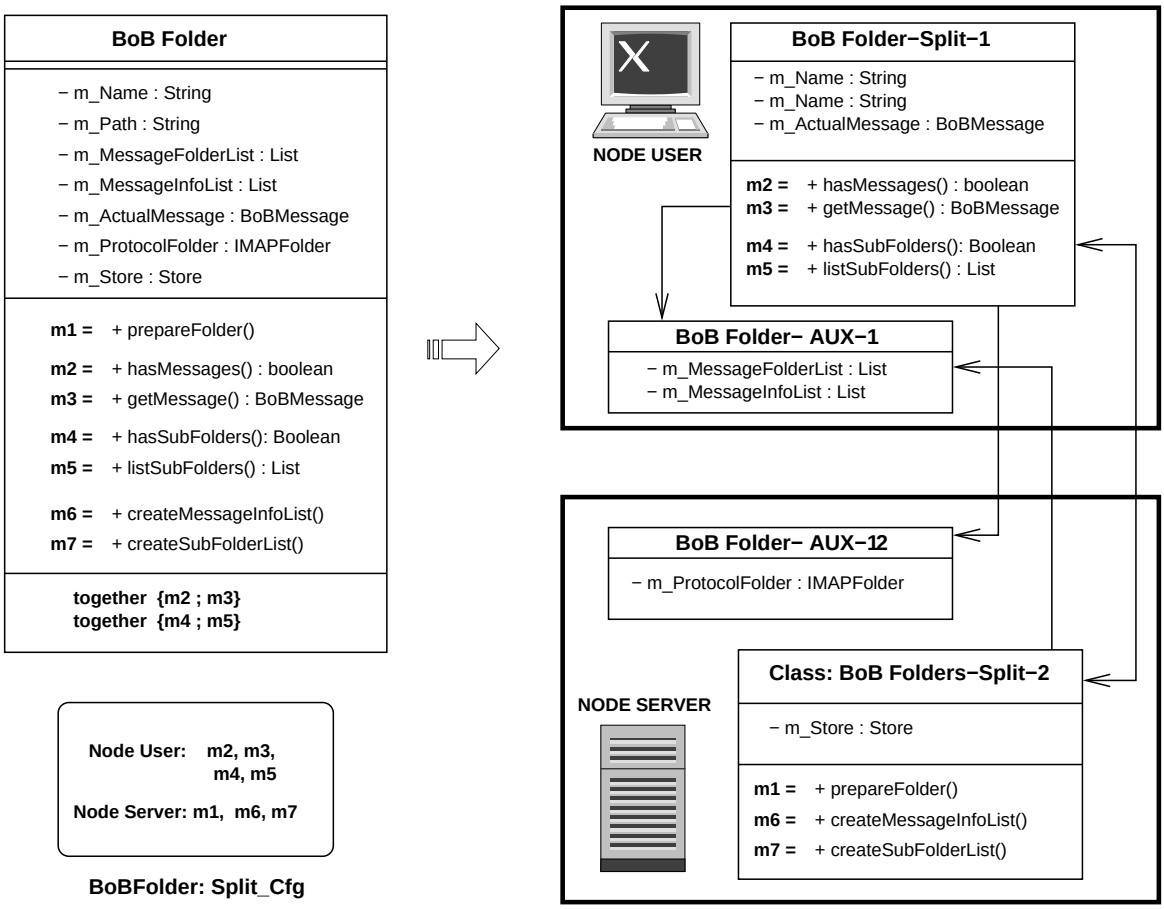


Figure 8.2: *Splitting and redeployment for* `BoBFolder`

is in contrast to multi-megabyte applications in server based environments. To reduce the application sizes, many optimization are advised [1]. Some of them are:

- Remove unnecessary classes by pruning the application's functionality. Build the most minimal version of the application.
- Examine and remove the inner classes if possible.
- Combine two or three classes as one multifunctional class and place all the code in one file.
- Maximize the use of pre-installed classes.
- Collapse application inheritance hierarchies.
- Shorten the names of packages, classes, methods and data members.

[1] `http://java.sun.com/developer/J2METechTips/2002/tt0226.html`

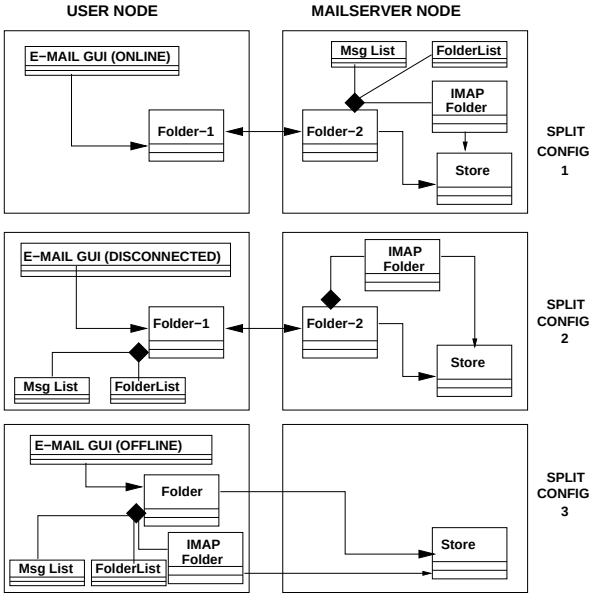


Figure 8.3: *BoB-based email application*

- Rexamine the use of array initializations.

In the section we discuss the example of a Stock Database application implemented on wireless device. We show how a BoB based approaches helps to reduce the overall size of application in most of the usage scenarios.

The MIDlet for this example does the following:

- Creates a record store (database).
- Adds new records (stocks) to the database.
- Views the stocks in the database.

To add a stock to the database, the user enters the stock symbol (such as, SUNW, IBM, IT, MS, GM, or Ford). The MIDlet retrieves the corresponding stock quote from the Yahoo Quote Server (http://quote.yahoo.com), constructs a record, and adds the record to the database. To view the stocks in the record store, the MIDlet iterates through the records in the record store and prints them on the display in a nice format.

8.2.1 The Implementation

The implementation of this MIDlet consists of the following three classes: Stock.java, StockDB.java, and QuotesMIDlet.java. Figure 8.4 shows the classes used in the used Record Management Store (RMS) package. This package forms the backend in this

MIDlet example. Figure 8.5 shows an implementation of Record Store class. We also show through RMS package, how it can optimized for different applet accesses, for example StockDB in this case.

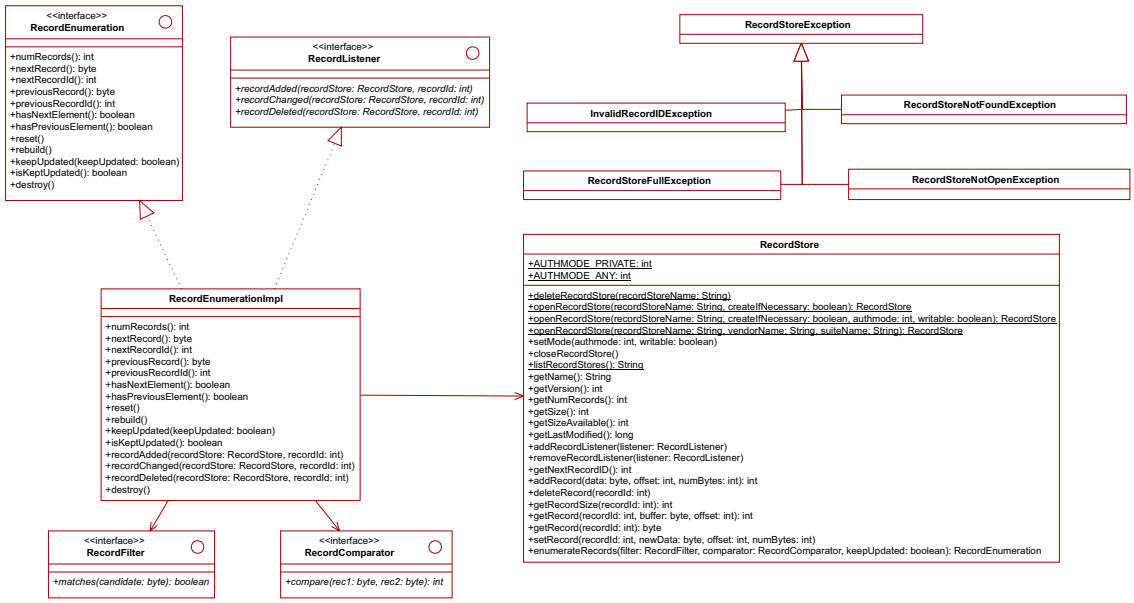


Figure 8.4: *Overview of javax.microedition.rms package*

8.2.2 The Stock.java Class

This class parses a string obtained from the Yahoo Quote Server or the record store into fields (such as, name of stock, or price). The string returned from the Yahoo Quote Server has the following format: NAME TIME PRICE CHANGE LOW HIGH OPEN PREV ”SUNW”, ”2:1PM - 79.75”, +3.6875, ”64.1875 - 129.3125”, 78, 76.0625 In this MIDlet, the fields retrieved are the name of the stock, the time, and the price.

```
%Listing 1: Stock.java
public class Stock {
  private static String name, time, price;
  // Given a quote from the server,
  // retrieve the name,
  //price, and date of the stock
  public static void parse(String data) {
    int index = data.indexOf('"');
    name = data.substring(++index,
        (index = data.indexOf('"', index)));
    index +=3;
    time = data.substring(index,
        (index = data.indexOf('-', index))-1);
```

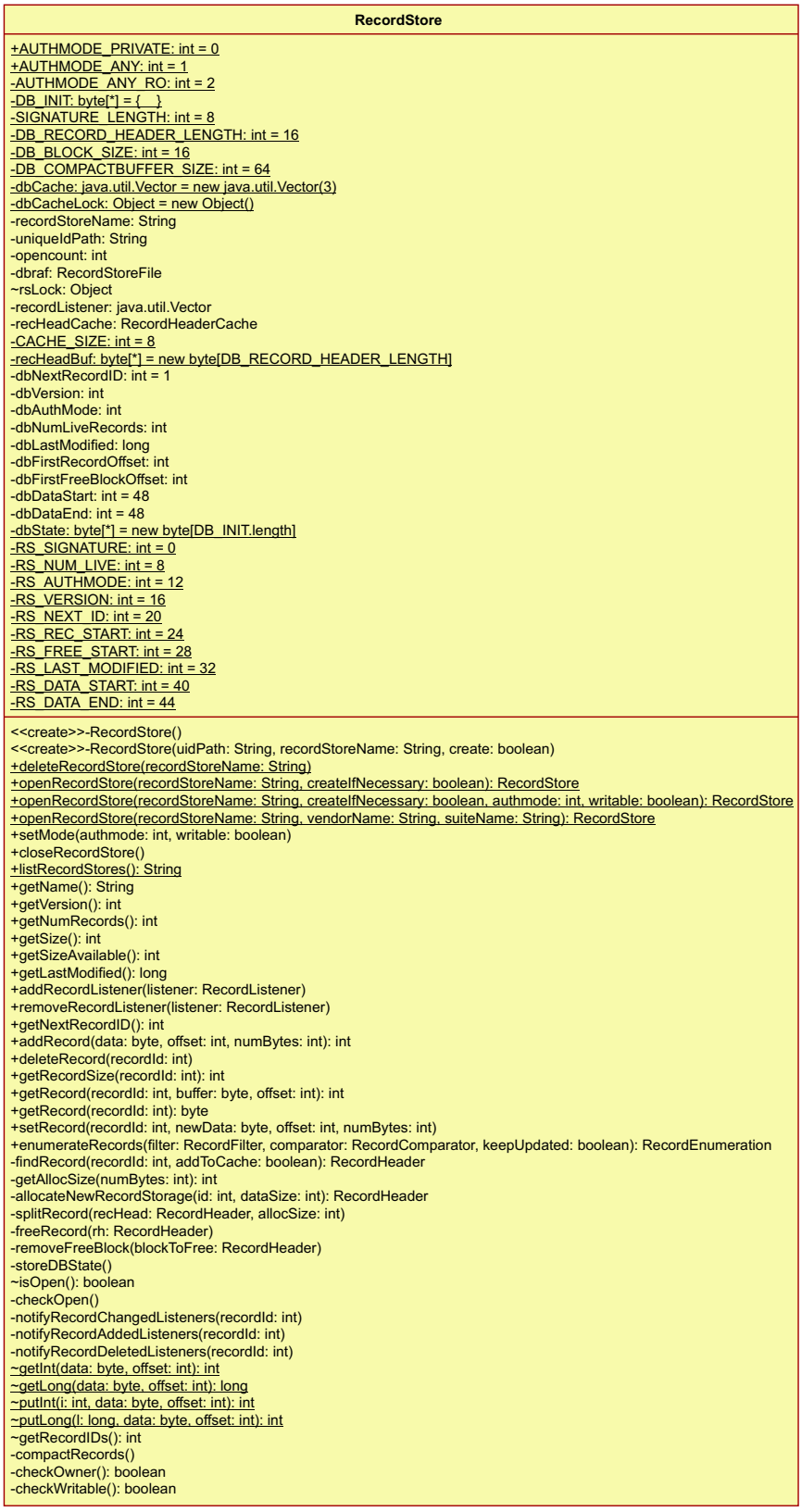


Figure 8.5: *Implementation of RecordStore*

```
    index +=5;
    price = data.substring(index,
        (index = data.indexOf('<', index)));
  }

  // Get the name of the stock from
  // the record store
  public static String
                getName(String record) {
    parse(record);
    return(name);
  }

  // Get the price of the stock from
  // the record store
  public static String
               getPrice(String record) {
    parse(record);
    return(price);
  }
}
```

8.2.3 The StockDB.java Class

This class provides methods that do the following:

- Opens a new record store.
- Adds a new record to the record store.
- Closes the record store.
- Enumerates through the records.

```
import javax.microedition.rms.*;
import java.util.Enumeration;
import java.util.Vector;
import java.io.*;

public class StockDB {
   RecordStore recordStore = null;
   public StockDB() {}

   // Open a record store with the given name
   public StockDB(String fileName) {
      try {
        recordStore =
              RecordStore.openRecordStore(
                          fileName, true);
      } catch(RecordStoreException rse) {
        rse.printStackTrace();
```

```
      }
   }

   // Close the record store
   public void close()
            throws RecordStoreNotOpenException,
                       RecordStoreException {
        if (recordStore.getNumRecords() == 0) {
            String fileName =
                           recordStore.getName();
            recordStore.closeRecordStore();
            recordStore.deleteRecordStore(
                                      fileName);
        } else {
            recordStore.closeRecordStore();
        }
    }

   // Add a new record (stock)
   // to the record store
   public synchronized void
             addNewStock(String record) {
        ByteArrayOutputStream baos = new
                ByteArrayOutputStream();
        DataOutputStream outputStream = new
                DataOutputStream(baos);
        try {
            outputStream.writeUTF(record);
        }
        catch (IOException ioe) {
            System.out.println(ioe);
            ioe.printStackTrace();
        }
        byte[] b = baos.toByteArray();
        try {
            recordStore.addRecord(b,
                              0, b.length);
        }
        catch (RecordStoreException rse) {
            System.out.println(rse);
            rse.printStackTrace();
        }
    }

    // Enumerate through the records.
    public synchronized
                 RecordEnumeration enumerate()
          throws RecordStoreNotOpenException {
       return recordStore.enumerateRecords(
                 null, null, false);
    }
}
```

8.2.4 The QuotesMIDlet.java Class

The QuotesMIDlet class is the actual MIDlet that does the following:

- Creates commands (List Stocks, Add New Stock, Back, Save, Exit).
- Handles command events.
- Connects to the YAHOO Quote Server and retrieves Quotes.
- Invokes methods from Stock and StockDB to parse quotes and add new stocks to the record store.

Listing 3: QuotesMIDlet.java

```
import javax.microedition.rms.*;
import javax.microedition.lcdui.*;
import javax.microedition.midlet.*;
import javax.microedition.io.*;
import java.io.*;
import java.util.Vector;

public class QuotesMIDlet
        extends MIDlet implements CommandListener {
    Display display = null;
    List menu = null; // main menu
    List choose = null;
    TextBox input = null;
    Ticker ticker =
                new Ticker("Database Application");
    String quoteServer =
        "http://quote.yahoo.com/d/quotes.csv?s=";
    String quoteFormat =
        "&f=slc1wop"; // The only quote format supported


    static final Command backCommand = new
                Command("Back", Command.BACK, 0);
    static final Command mainMenuCommand = new
                Command("Main", Command.SCREEN, 1);
    static final Command saveCommand = new
                Command("Save", Command.OK, 2);
    static final Command exitCommand = new
                Command("Exit", Command.STOP, 3);
    String currentMenu = null;

    // Stock data
    String name, date, price;

    // record store
    StockDB db = null;

    public QuotesMIDlet() { // constructor
    }
```

```
// start the MIDlet
public void startApp()
    throws MIDletStateChangeException {
  display = Display.getDisplay(this);
  // open a db stock file
  try {
    db = new StockDB("mystocks");
  } catch(Exception e) {}
  menu = new List("Stocks␣Database",
                  Choice.IMPLICIT);
  menu.append("List␣Stocks", null);
  menu.append("Add␣A␣New␣Stock", null);
  menu.addCommand(exitCommand);
  menu.setCommandListener(this);
  menu.setTicker(ticker);

  mainMenu();
}

public void pauseApp() {
  display = null;
  choose = null;
  menu = null;
  ticker = null;

  try {
    db.close();
    db = null;
  } catch(Exception e) {}
}

public void destroyApp(boolean
                  unconditional) {
  try {
    db.close();
  } catch(Exception e) {}
  notifyDestroyed();
}

void mainMenu() {
  display.setCurrent(menu);
  currentMenu = "Main";
}

// Construct a running ticker
// with stock names and prices
public String tickerString() {
   StringBuffer ticks = null;
   try {
      RecordEnumeration enum =
                    db.enumerate();
      ticks = new StringBuffer();
      while(enum.hasNextElement()) {
        String stock1 =
            new String(enum.nextRecord());
        ticks.append(Stock.getName(stock1));
        ticks.append("␣@␣");
```

```
          ticks.append(Stock.getPrice(stock1));
          ticks.append("␣␣␣␣");
        }
     } catch(Exception ex) {}
     return (ticks.toString());
  }

  // Add a new stock to the record store
  // by calling StockDB.addNewStock()
  public void addStock() {
    input = new TextBox(
              "Enter␣a␣Stock␣Name:", "", 5,
              TextField.ANY);
    input.setTicker(ticker);
    input.addCommand(saveCommand);
    input.addCommand(backCommand);
    input.setCommandListener(this);
    input.setString("");
    display.setCurrent(input);
    currentMenu = "Add";
  }

  // Connect to quote.yahoo.com and
  // retrieve the data for a given
  // stock symbol.
  public String getQuote(String input)
              throws IOException,
                    NumberFormatException {
    String url = quoteServer + input +
                               quoteFormat;
    StreamConnection c =
         (StreamConnection)Connector.open(
              url, Connector.READ_WRITE);
    InputStream is = c.openInputStream();
    StringBuffer sb = new StringBuffer();
    int ch;
    while((ch = is.read()) != -1) {
      sb.append((char)ch);
    }
    return(sb.toString());
  }

  // List the stocks in the record store
  public void listStocks() {
      choose = new List("Choose␣Stocks",
                     Choice.MULTIPLE);
      choose.setTicker(
           new Ticker(tickerString()));
      choose.addCommand(backCommand);
      choose.setCommandListener(this);
    try {
       RecordEnumeration re = db.enumerate();
       while(re.hasNextElement()) {
         String theStock =
                 new String(re.nextRecord());
         choose.append(Stock.getName(
                               theStock)+"␣@␣"
```

```
            + Stock.getPrice(theStock), null);
         }
      } catch(Exception ex) {}
      display.setCurrent(choose);
      currentMenu = "List";
   }

   // Handle command events
   public void commandAction(Command c,
                        Displayable d) {
      String label = c.getLabel();
      if (label.equals("Exit")) {
         destroyApp(true);
      } else if (label.equals("Save")) {
          if(currentMenu.equals("Add")) {
              // add it to database
            try {
              String userInput =
                           input.getString();
              String pr = getQuote(userInput);
              db.addNewStock(pr);
              ticker.setString(tickerString());
            } catch(IOException e) {
            } catch(NumberFormatException se) {
            }
            mainMenu();
          }
      } else if (label.equals("Back")) {
          if(currentMenu.equals("List")) {
            // go back to menu
            mainMenu();
          } else if(currentMenu.equals("Add")) {
            // go back to menu
            mainMenu();
          }
      } else {
         List down = (List)display.getCurrent();
         switch(down.getSelectedIndex()) {
           case 0: listStocks();break;
           case 1: addStock();break;
         }
      }
   }
}
```

In the next chapter, we relax many of our restrictions on BoBs and consider the BoB fragment as an artifact for reuse. We discuss the various BoB operations for application extension and contraction.

Architecture starts when you carefully put two bricks together. There it begins.

- Ludwig Mies van der Rohe

Chapter 9

Software Composition Using BoBs

In the previous chapters, we considered behavioral preserving transformations (split and merge) for BoBs. In this chapter, we discuss how BoBs are used as composable units for *reuse* and *extensibility*. We first build a formal model for BoBs and then explain basic *BoB operations*. BoBs can not only combine with other BoBs but also combine with other BoB fragments. This provides an effective and flexible mechanism for *incremental* software evolution.

9.1 Structure of BoB - revisited

BoB interface *exposes* the features of a BoB. A BoB is split on the basis of its *interface methods*. The BoB which is split is called *principal BoB*. Splits of a BoB are said to have a is_split_of relation to the *principal* BoB and are called fragments. It is depicted by the symbol $\succ$. The reverse relation is is_principal_of and is depicted by $\prec$.

For example: Consider the calculator example discussed in Chapter 5.

```
BoBClass Calculator {
    public int ADD (int x, int y) {return (x+y)};
    public int SUB (int x, int y) {return (x-y)};
    public int MUL(int x, int y) {return (x*y)};
    public int DIV (int x, int y) {return (x/y)};
}
```

If Calculator is split for each mathematical operation, we get four BoB splits, namely $\text{Calculator}_{\text{ADD}}$, $\text{Calculator}_{\text{SUB}}$, $\text{Calculator}_{\text{MUL}}$, and $\text{Calculator}_{\text{DIV}}$.

```
BoBClass Calculator_ADD{
    public int ADD (int x, int y) {return (x+y)};
}

BoBClass Calculator_SUB{
    public int SUB (int x, int y) {return (x-y)};
}
```

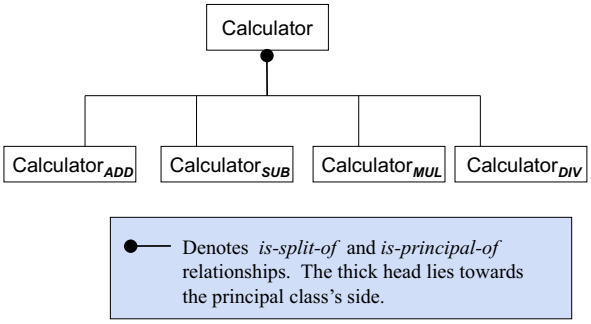


Figure 9.1: *Calculator BoB in extended UML notation*

```
BoBClass Calculator_MUL{
    public int MUL(int x, int y) {return (x*y)};
}

BoBClass Calculator_DIV {
    public int DIV (int x, int y) {return (x/y)};
}
```

Then, for the multiply-split we have:

$\mathsf{Calculator} \prec \mathsf{Calculator}_{\mathsf{MUL}}$

$\mathsf{Calculator}_{\mathsf{MUL}} \succ \mathsf{Calculator}.$

A principal BoB is acts a *sub-type* to any of its fragments. The interface methods of a BoB-fragment provide the same service, as they would have, had they been present inside the principal BoB.

We extend the UML notation to denote these new relationships between two classes. It is shown in the Figure 9.1.

9.1.1 Allowing inheritance

Since here the purpose here is reusability, and the programmer is aware of the classes he is creating at the time of code creation (extension is *explicit*, inheritance can be easily used and is a useful technique in the construction of BoBs. In fact, it provide a mechanism for building a structure of fragment refinements within a BoB.

9.1.2 Formal Model for BoB

We start with a basic model of BoB based program. Later we extend the definition of BoB class after the discussion of BoB compositions in the later sections. Consider a program P comprising of BoB classes $\mathsf{C_1}, \mathsf{C_2}, \ldots, \mathsf{C_n}$.

Figure below shows the basic construction of a BoB class. In its basic canonical form it is constructed from fields and methods or by extending it from another BoB.

Program		
P	::=	C_1, C_2 ... C_n
BoBClass		
C	::=	C { fd* con* md*} | C extends C
Field		
fd	::=	C f;
Constructor		
con	::=	C(e*){super(e*) | this(e*) ; s*}
Method		
md	::=	τ m (e*) {s*}
τ	::=	C |`void`
f	::=	field name
m	::=	method name
e	::=	argument expression
s	::=	statment

We now define some basic terminology to build our formal model for BoB compositions.

Let A be the BoB class.

$m \rightarrow$ set of interface `together` method specifications of A,

$n \rightarrow |m|$, cardinality of m or total number of interface `together` method specifications.

Definition 9.1.1 (Selection ρ) *It is a subset of the BoB interface method specification m and denotes a selection on the interface methods of BoB.*

$$\rho_x = selection_x(m \in A) \subset m,$$

$$x \subset \{y | y \rightarrow 1, n\}$$

where, x is a set having one to one correspondence with ρ and is used as short hand notation for selection. For example for $\rho = \{m_1, m_3, m_4\}$, we have $x = \{1, 3, 4\}$ and we denote this selection as: ρ_x or $\rho_{1,3,4}$

Also note that:

This signifies the process of splitting a BoB. See figure 9.2.

Definition 9.1.2 (Selection Set ζ) *It is the set specifying the configuration file for splitting. It is a set containing the selections on m such that all methods are included and no two selections intersect.*

$$\zeta = \{\rho_x \mid \forall x(\ \cup \rho_x = m,\ \wedge\ \cap \rho_x = \phi)\}$$

For example, For a BoB A having $m = \{m1, m2, m3, m4, m5, m6\}$, one such *selection set* is:

$$\zeta_A = \{\{m1, m2\}, \{m3, m5\}, m4, m6\}$$

or

$$\zeta = \{\rho_{1,2},\ \rho_{3,5},\ \rho_4,\ \rho_6\}$$

9.2 BoB Basic Operations

These are the extentions of the basic BoB operations *split* and merge disscussed in the previous chapters, except that we now consider the use of inheritance. Additonally, we provide an operation for *extracting* a fragment from a BoB.

9.2.1 Split $\otimes$

As discussed in the earlier chapters, it is the process of splitting a BoB into split-fragments. It results in a set having all the split-fragments obtained by using a selection set ζ.

$$A \xrightarrow[\zeta_A]{\otimes} \Big\{A_x\Big\}_{\forall\ \rho_x \in\ \zeta_A}$$

where, A_x is the split-fragment corresponding to the selection ρ_x.

For example, for a BoB A with interface methods, $m1, m2, m3, m4, m5, m6$, if we consider a selection set, $\zeta_A = \{\{m1, m2\}, \{m3, m5\}, m4, m6\}$, we get:

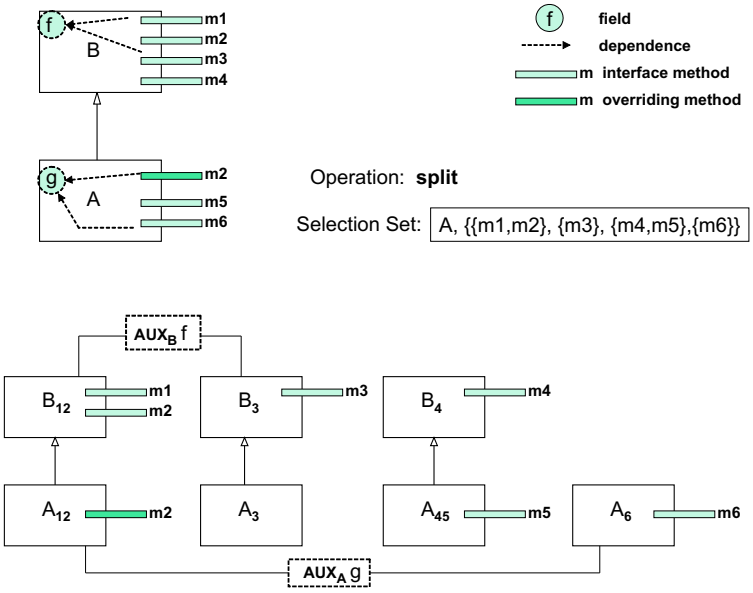


Figure 9.2: *Composition: Split - Operation*

$$A \xrightarrow{\otimes} \Big\{ A_{1,2} \ , \ A_{3,5} \ , \ A_4 \ , \ A_6 \Big\}$$

Please note that, each fragment is itself a BoB and can be further split if $|m| > 1$. A BoB with $|m|$ is called an *atomic* BoB and represents the smallest unit of composition in BoB compositions.

In the above example, $A_{1,2},\ and\ A_{3,5}$ can be further split:

$$A_{1,2} \xrightarrow{\otimes} \Big\{ A_1 \ , \ A_2 \Big\}$$

$$A_{3,5} \xrightarrow{\otimes} \Big\{ A_5 \ , \ A_5 \Big\}$$

The final split-set thus obtained:

$$\Big\{ A_1 \ , \ A_2 \ , A_3 \ , \ A_4 \ , \ A_5 \ , \ A_6 \Big\}$$

contains only atomic BoBs.

In Figure 9.2, we show the use of inheritance, and the corresponding splits for a selection set, $\zeta_A = \{\{m1, m2\}, \{m3\}, \{m4, m5\}, \{m6\}\}$. The shared auxiliary fields $(AUX_Ag, andAUX_Bf$ in this case), at each hierarchy level $(A, andB)$, are replicated in each BoB fragment that shares them.

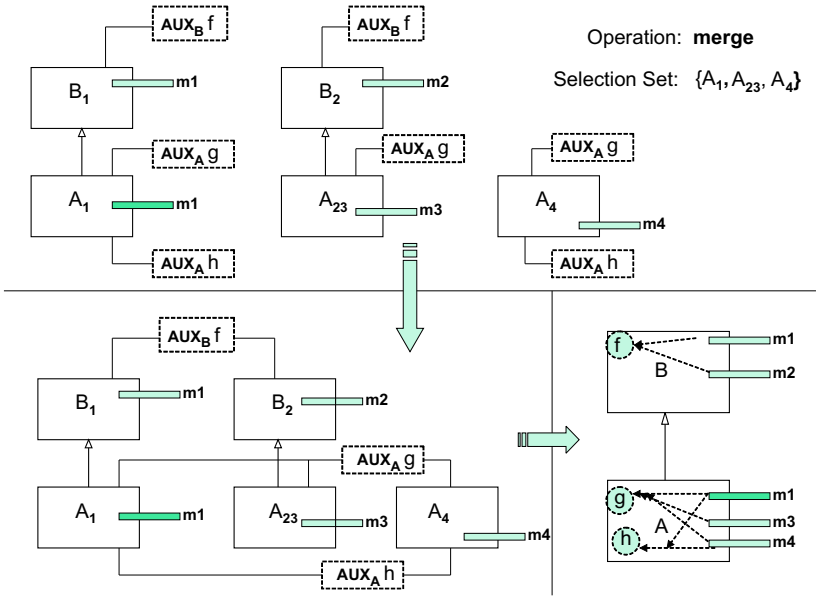


Figure 9.3: *Composition: Merge - Operation*

9.2.2 Merge ⋈

This signifies the process of integrating the BoBs from its splits. It is reverse of split. We specify the split-fragments that need to be merged and the result is an integrated BoB. We can either do a *partial merge* where only a subset of the split-fragments are integrated, or a *full merge* where the whole set of split-fragments from one particular split operation are integrated.

Full merge,

$$\left\{A_x\right\} \xrightarrow[\forall\ \rho_x \in\ \zeta_A]{\bowtie} A$$

Partial merge:

$$\left\{A_x\right\} \xrightarrow[forsome\ \rho_x \in\ \zeta_A]{\ltimes} A$$

Since, merging is reverse of splitting, following relation holds good:

$$A \quad \xrightarrow[\zeta_A]{\otimes} \left\{A_x\right\} \xrightarrow[\forall\ \rho_x \in\ \zeta_A]{\bowtie} A$$

Later we shall show that a *partial merge*, where only two fragments are involved, is equivalent to *addition* operation. Similarly a *partial split*, where only one fragment is output, is equivalent to *extract* operation.

Figure 9.3 shows an example of merging operation. Please note that corresponding BoB fragments at the same hiearchy level get merged together. The operation of merging follows the same rules as discussed in Chapter 7.

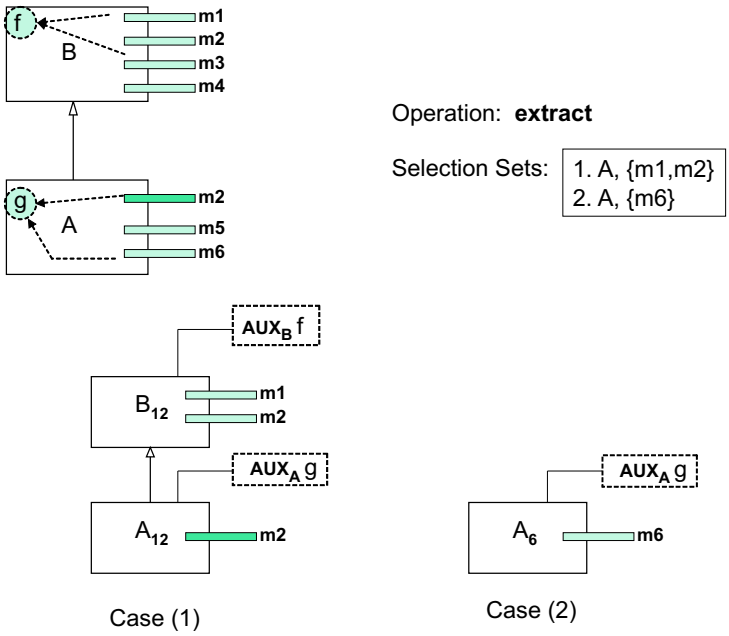


Figure 9.4: *Composition: Extract Operation*

9.2.3 Extract Fragment ξ

It is the process of extracting a fragment from a BoB by doing a selection m_x.

Step1: Obtain: A_x as described in previous sections.

Figure 9.4 illustrates this operation.

Other, BoB fragment compositions discussed in next section, do not allow inheritance in fragments. If an extract operation yields multiple levels of hiearchy, we collapse (or flatten) this hierarchy to obtain a BoB fragment. A BoB fragment, in that stricter sense is defined as:

Step 2: Collapse A_x

$$Frag_x A = collapse\{A_x\}$$

Summarizing:

$$\xi(C, \rho_x) \rightarrow Frag_x A$$

Note: A total of 2^n fragment combinations are possible, where $n = |m|$

9.2.4 Remove

It is the process of removing a ρ_x from the A and obtaining the remaining fragment. The process is complimentary to the extract opearation.

$$Remove(C, \rho_x) \rightarrow \xi(C, \rho_{\{m-x\}})$$

In the remaining sections, we describe the composition methods for extending BoBs.

9.3 BoB Fragment Compositions

These represent the mechanisms for extending (or contracting) a BoB by adding (or subtracting) another BoB or BoB fragment to it. These operations are of the form $C \circledcirc Frag$, where $Frag$ denotes a BoB fragment. A BoB fragment is equivalent to a BoB, but in the strict sense, as it is used in the operations here, it is a BoB whose inheritance hierarchy has been collapsed (or flattened). The symbol $\circledcirc$ denotes one of the composition operations described the following sections. A BoB class C forms the left hand side (LHS) of composition. The right-hand-side (RHS) is always a BoB fragment.

9.3.1 Structure Preserving Compositions

In these operations, the refinement structure is preserved. This is achieved by keeping the added refinement in the form of fragment hiearchy.

Fragment Add($C + F$)

Adds a new functionality to the BoB. The process is illustrated in Figure 9.5.

The semantics for addition are similar to the merge operations, except for some differences where an additional functionality is added.

1. For methods following rules are used:
   (a) If the method signatures are different, each method is included.
   (b) If method signatures are same:
      i. If one is `private` and another `public`, `public` access specifier is used. The general rule that is followed is that *access should not decrease* according to the following ordering `private < default < protected < public`.

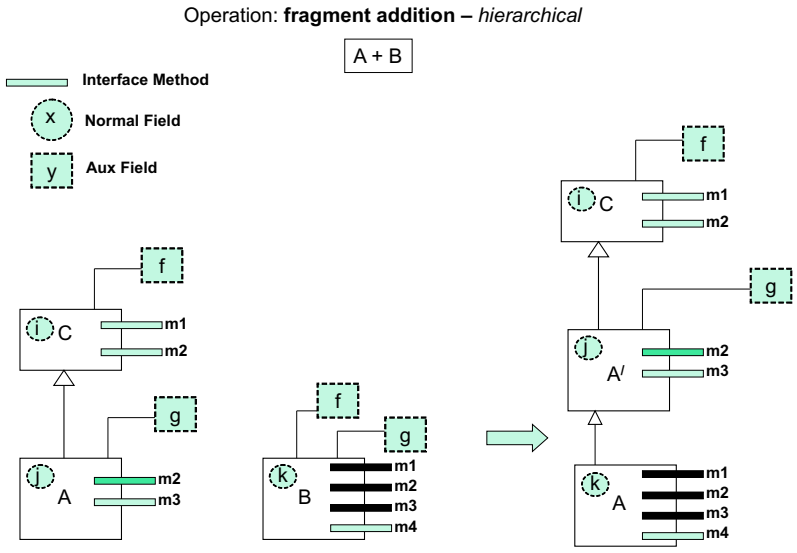


Figure 9.5: *Composition: Fragment - Addition - Hierarchical*

ii. If, the methods have different implementations, then methods of the added BoB-Fragment on the RHS override the methods on the LHS

2. For fields, following rules are used:

   (a) If the names are different, each field is included.

   (b) If two fields, for example, f1 on the LHS and f2 on the RHS, have same name, then the constraint is, Type (f1) = Type(f2), else the composition error is flagged.

3. The constructors are merged and for new additions their signatures are expanded[1].

4. The resultant BoB Class has the same name as the LHS Class unless explicit name is mentioned.

Figure 9.5 illustrates this operation. Please note that the added fragment adds one level of refinement to the hierarchy.

Fragment Subtract ($C - F$)

In this a fragment is subtracted from a BoB class. The *to be removed fragment* is specified. One of the requirement is that this fragment should find a match at

[1]For the sake of simply we consider only one fully expanded constructor for one fragment

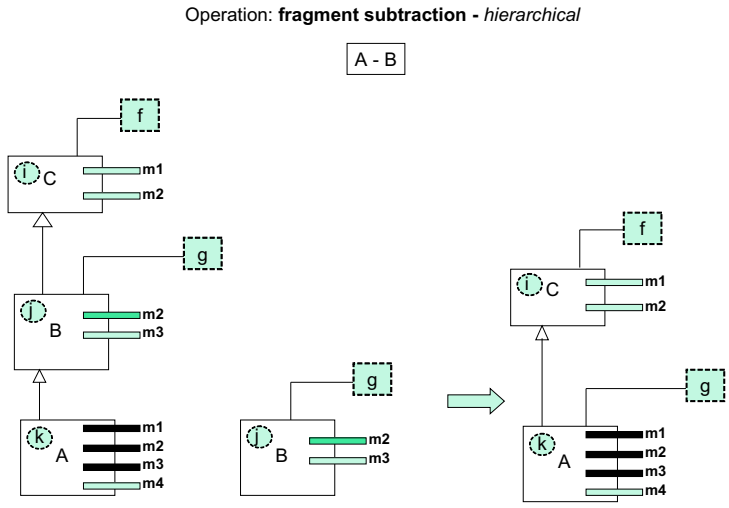


Figure 9.6: *Composition: Fragment Subtraction (Structure Preserving)*

some level of the hierarchy. If it gets matched in the fragment hierarchy of the given component, the corresponding fragment is removed. The hierarchy is readjusted, and the fields that are used by the fragment refinements in the regions lower to the removed fragment, are pushed down one level. Figure 9.6 illustrates this operation.

9.3.2 Fragment Replace ($\#(C, F_{old}, F_{new})$)

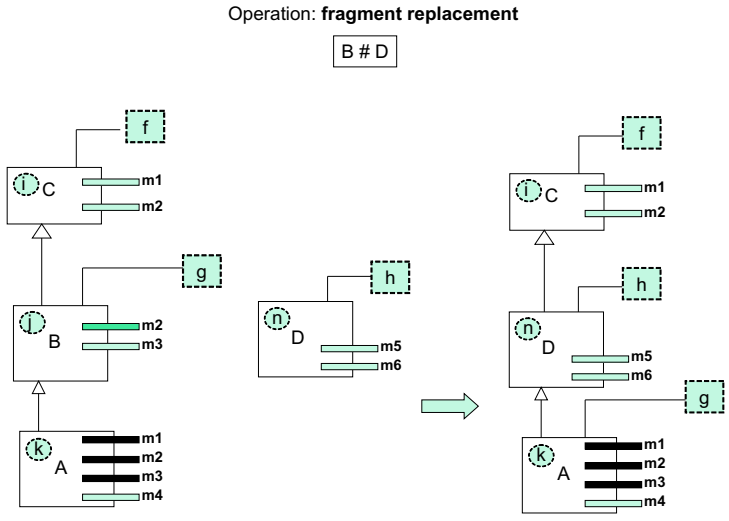


Figure 9.7: *Composition: Fragment Replacement*

The *to be removed fragment* is specified. Its match in the LHS BoB hierarchy is searched and subtracted. The new fragment is added *in place* to the removed

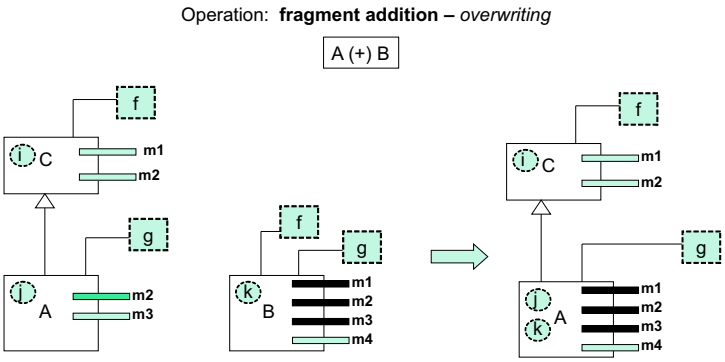


Figure 9.8: *Composition: Fragment Addition (Overwriting)*

fragment. Hence the process is two fold: (i) Fragment Subtraction , (ii) Fragment Addition (in-place). Figure 9.7 illustrates this operation.

9.3.3 Overwriting Compositions

In these compositions no new level of hierarchy is added or subtracted from a BoB.

Fragment Add ($C(+)F$)

The addition is similar to structure preserving addition except that the fragment addition in this case *overwrites* the methods in the LHS BoB class. In this form of addition, no new level of hierarchy is added. All the additions take place to the leaf node of the BoB class on LHS. Figure 9.8 illustrates this operation.

Fragment Subtract ($C(-)F$)

The *to be removed fragment* is specified. The signatures of the methods which are to be removed are derived from this fragment. The methods corresponding to this selection of methods are removed from the leaf of BoB class on LHS. Figure 9.9 illustrates this operation.

9.4 Extended Definition of BoB

We discussed the basic model for a BoB class in the Section 9.1.2. We now extend this model by incorporating the operations as discussed in the above sections.

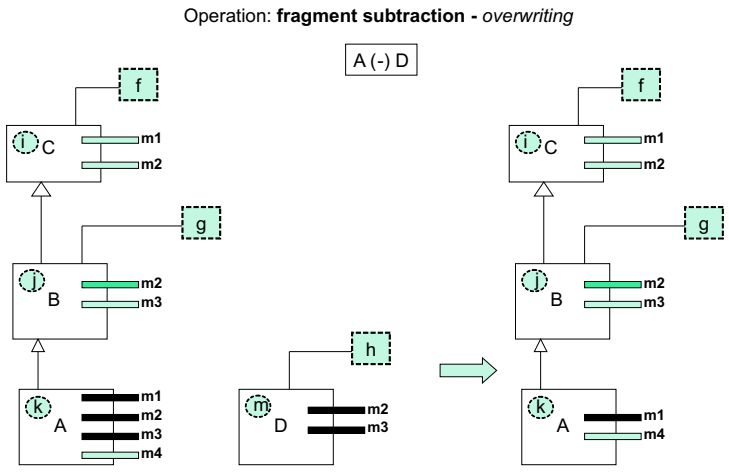


Figure 9.9: *Composition: Fragment Subtraction (Overwriting)*

The extended definition of a BoB is shown below:

C	::=	C extends C { fd* con* md*} | frag | ξ (C, ρ) | C $\circledcirc$ frag | #(C,frag, frag)
frag	::=	collapse(ξ (C, ρ))
$\circledcirc$	::=	+ | - | $\oplus$ | $\ominus$

In the next chapter we provide the case-study of a software product line component, where many of the above BoB compositions are illustrated.

With a painter or a sculptor, one cannot begin to alter his works, but an architect has to put up with anything, because he makes utility objects - the building is there to be used, and times change.

- Arne Jacobsen

Chapter 10

Case Studies-2

This chaper highlights the usage of BoB as a variability mechanism in software. We take the graph product line (GPL) example discussed in literature in [Bat03] and provide a BoB based design. Next we consider a large-scale distributed application, and explore where BoB based applications design can help. Finally, we discuss a small example of BoB based adaptation.

10.1 Graph Product Line

Graph Product Line (GPL) represents the family of graph applications. It is proposed as a standard problem for evaluating product line methodologies in [LH01] and is based on the some of the exensibility works in object-oriented technology, like [Hol93] and [Van96a]. The graph applications differ from each other on the basis of certain *features*. These features [LH01] are described briefly below:

- Directed, Undirected
- Weighted, Unweighted
- Search algorithm: breadth-first search (BFS) or depth-first search (DFS);
- Additional algorithms:
  - Vertex Numbering (Number): Assigns a unique number to each vertex as a result of a graph traversal.

- Connected Components (Connected): Computes the connected components of an undirected graph.
- Strongly Connected Components (StronglyConnected): Computes the strongly connected components of a directed graph, which are equivalence classes under the reachable-from relation. A vertex y is reachable form vertex x if there is a path from x to y.
- Cycle Checking (Cycle): Determines if there are cycles in a graph.
- Minimum Spanning Tree (MST Prim, MST Kruskal): Computes a Minimum Spanning Tree (MST), which contains all the vertices in the graph such that the sum of the weights of the edges in the tree is minimal.
- Single-Source Shortest Path (Shortest): Computes the shortest path from a source vertex to all other vertices.

In Figure 10.3, we show the dependencies between various features. Note that not all feature combinations are meaningful or even possible to co-exist.

10.2 BoB Based Design

Let us start with a graph which is minimal and has the following features: Undirected, Weighted and implements the MSTKruskal algorithm. Construction of such a BoB is shown below.

We have a client program which invokes a method `MSTKruskal()` which computes a minimum spanning tree for the read in graph and displays it.

```
public class Main {
    public static void main( String[] args ) {

       // Create graph object
        Graph g = new  Graph();

       // Read in graph values from the specified file
       //  and construct graph objects

        try {

            // for(i =0, i < num_vertices; i++) {g.addVertex(V[i])};
            // for(i =0, i < num_edges; i++)
            //   {g.addAnEdge(V[start_Vertices[i]], V[endVertices[i]], weights[i])}

            g.runReadInGraph( args[0] );
```

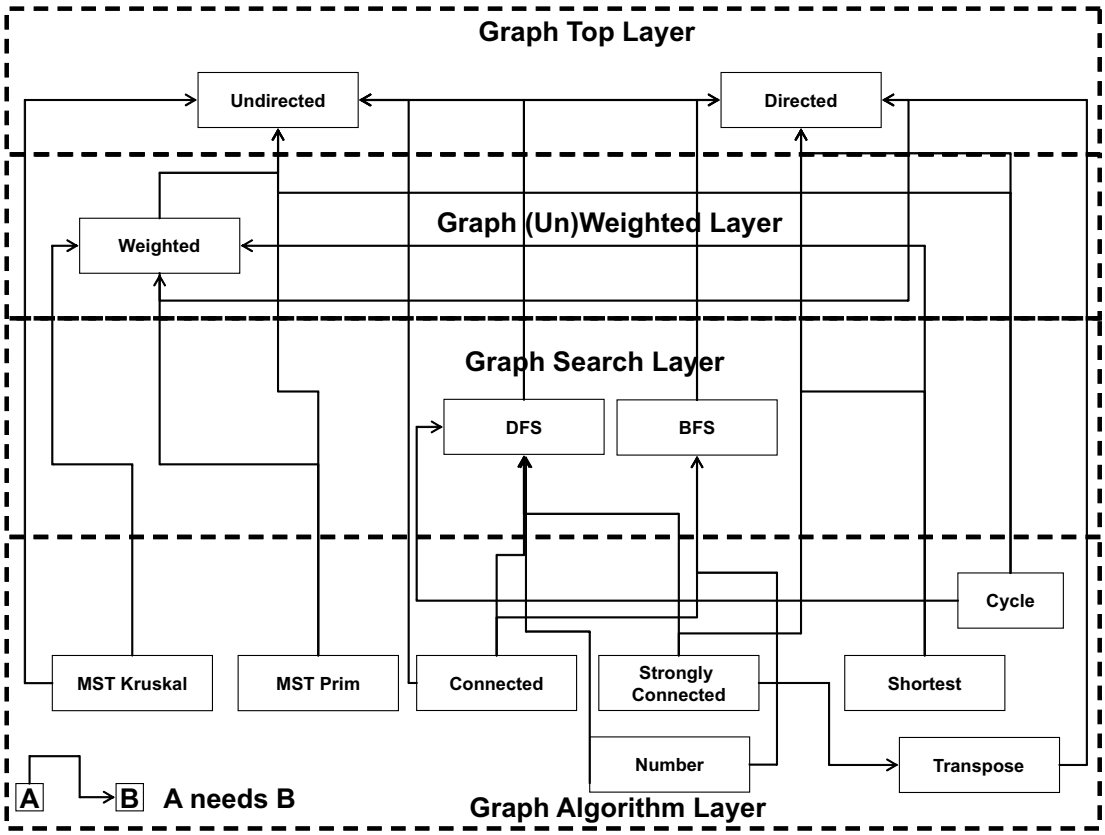


Figure 10.1: *GPL Layers*

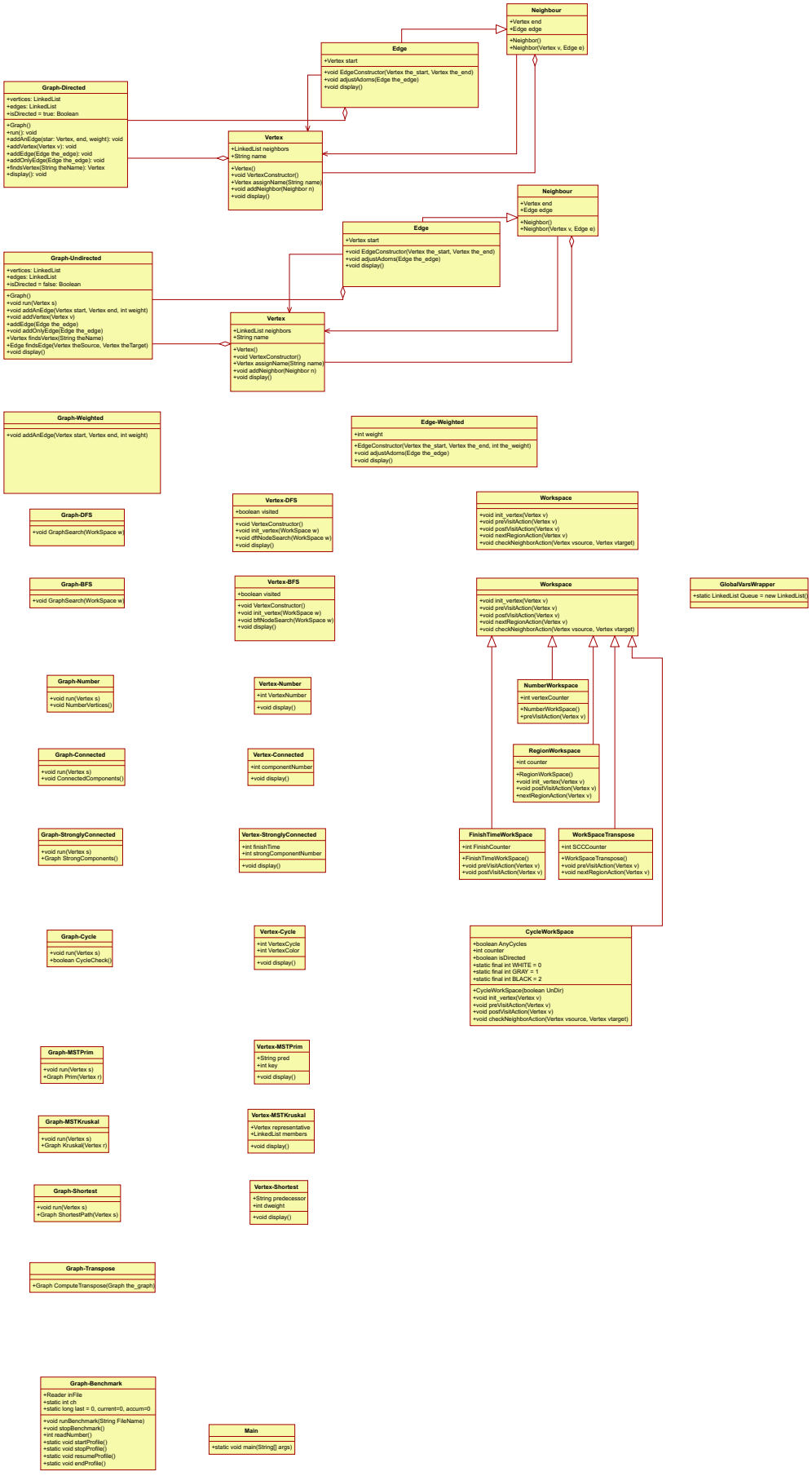


Figure 10.2: *GPL Classes*

```
        }
        catch( IOException e ) {}

       // Executes the selected features
        g.startProfile();

        g.run( g.findsVertex( args[1] ) );
        g.MSTKruskal();

        g.stopProfile();
        g.display();
        g.resumeProfile();

       // End profiling
        g.endProfile();

    } // End main

}
```

Below is the above BoBClass Graph code:

```
BoBClass Graph
{
   import java.lang.Integer;
   import java.util.LinkedList;
   import java.util.Collections;
   import java.util.Comparator;

   private LinkedList vertices;
   private LinkedList edges;

   F1 = public static final boolean isDirected = false;

   C1 = public Graph() {..}

   M1  = public void run( Vertex s ) {/* dummy */}
 //  public void addAnEdge( Vertex start,  Vertex end, int weight ) {..}
   M2  = public void addVertex( Vertex v ) {..}
   M3  = public void addEdge( Edge the_edge ) {..}
   M4  = public void addOnlyEdge( Edge the_edge ) {..}
   M5  = public Vertex findsVertex( String theName ) {..}
   M6  = public Edge findsEdge( Vertex theSource, Vertex theTarget ) {..}
   M7  = public void display() {..}

  //For weighted graph
   M8  = public void addAnEdge( Vertex start,  Vertex end, int weight ){..}

   M9  = public void run( Vertex s ){..}

   M10 = public Graph Kruskal( Vertex r ) {\ldots}

   M20 = public readInGraph(Vertex s){..}
}
```

Creating split objects from this BoB class, with following configuration:

```
Split 1 = F1, C1, M1-M7, M20
Split 2 = M8
Split 3 = M9, M10
```

The splits are shown below:

```
BoBClass Graph_Split_1_Undirected
{
   private LinkedList vertices;
   private LinkedList edges;

   F1 = public static final boolean isDirected = false;

   C1 = public GraphSplit_1() {..}

   M1  = public void run( Vertex s ) {/*Dummy*/}
 //  public void addAnEdge( Vertex start,  Vertex end, int weight ) {..}

{Use this to show that unnecessary methods are carried over}

   M2  = public void addVertex( Vertex v ) {..}
   M3  = public void addEdge( Edge the_edge ) {..}
   M4  = public void addOnlyEdge( Edge the_edge ) {..}
   M5  = public Vertex findsVertex( String theName ) {..}
   M6  = public Edge findsEdge( Vertex theSource, Vertex theTarget ) {..}
   M7  = public void display() {..}

   M20 = public readInGraph(Vertex s){..}
}


BoBClass Graph_Split_2_Weighted
{
   import java.lang.Integer;
   import java.util.LinkedList;
   import java.util.Collections;
   import java.util.Comparator;

   //* NOT required in this split */ private LinkedList vertices;
   private LinkedList edges;
   public -> Make Above public

   C1 = public GraphSplit_2( ) {..}

  //For weighted graph
   M8  = public void addAnEdge( Vertex start,  Vertex end, int weight ){..}

   M3 = private void addEdge( Edge the_edge){..}

}


BoBClass GraphSplit_3_MSTKruskal
{
   import java.lang.Integer;
```

```
   import java.util.LinkedList;
   import java.util.Collections;
   import java.util.Comparator;

   private LinkedList vertices;
   private LinkedList edges;

   C1 = public GraphSplit_3( ) {..}


   M9  = public void run( Vertex s ){..}

   M10 = public Graph Kruskal( Vertex r ) {\ldots}

}
```

The fully integrated BoB class for Graph is shown in the figure below:

```
BoBClass Graph
{

   private LinkedList vertices;
   private LinkedList edges;

   F1 = public static final boolean isDirected = false;

   C1 = public Graph() {..}

   M1  = public void run( Vertex s ) {/*Dummy*/}
 //  public void addAnEdge( Vertex start,  Vertex end, int weight ) {..}
   M2  = public void addVertex( Vertex v ) {..}
   M3  = public void addEdge( Edge the_edge ) {..}
   M4  = public void addOnlyEdge( Edge the_edge ) {..}
   M5  = public Vertex findsVertex( String theName ) {..}
   M6  = public Edge findsEdge( Vertex theSource, Vertex theTarget ) {..}
   M7  = public void display() {..}

  //For weighted graph
   M8  = public void addAnEdge( Vertex start,  Vertex end, int weight ){..}

   M9  =  public void DFSGraphSearch(Workspace w)
   M10 = public  void BFSGraphSerach(Workspace w)

   M9  = public void run( Vertex s ){..}

   M10 = public Graph Kruskal( Vertex r ) {..}
   M11 = public Graph Prim(Vertex r) {..}
   M12 = public Graph ComputeTranspose(Graph the_graph) {\ldots}
   M13 = public Graph ShortestPath(Vertex s) {..}
   M14 = public Graph StrongComponents(){..}

   M15 = public void NumberVertices() {..}

   M20 = public readInGraph(Vertex s){..}

}
```

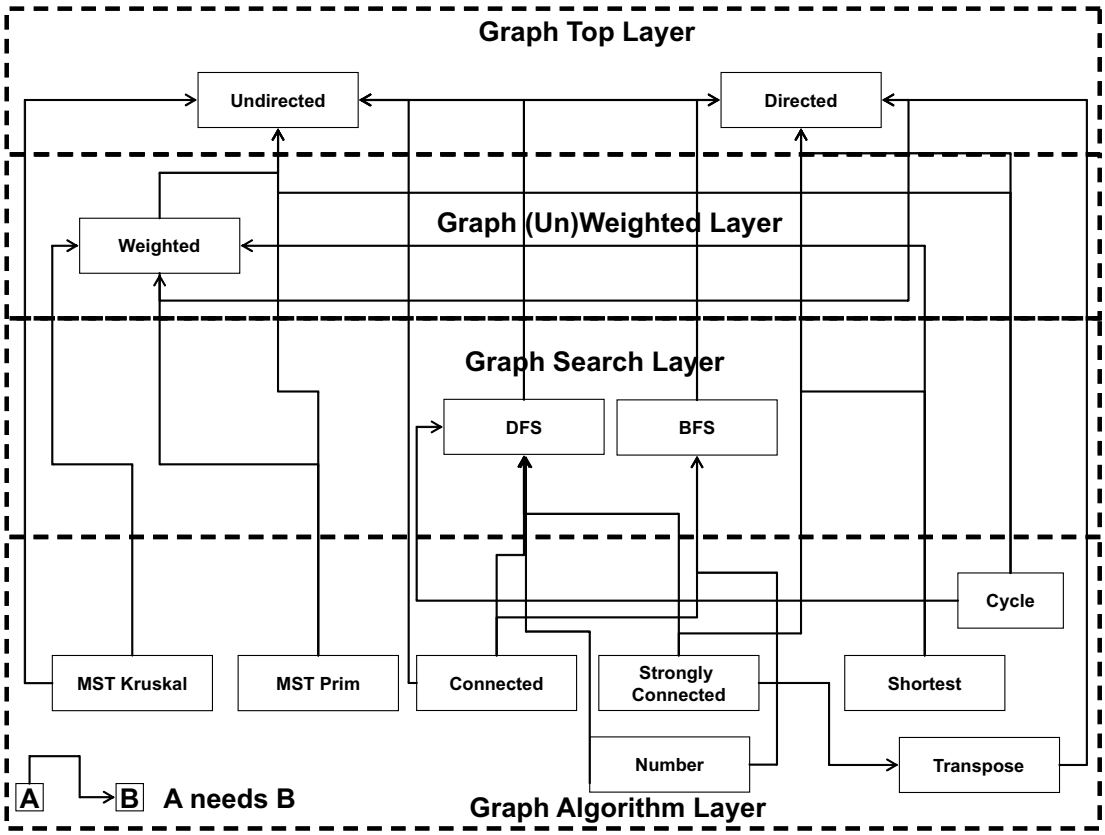


Figure 10.3: *GPL Layers*

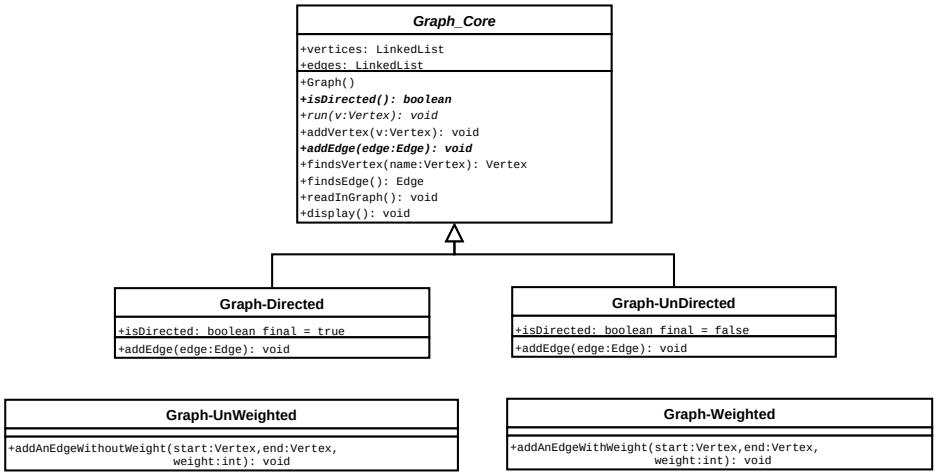


Figure 10.4: *Simple GPL Core Layers*

10.3 Programming using BoBs

10.3.1 Using scripts to create BoB classes

We shall now write a program in Java$_{BoB}$ scripting language that will construct two different graph objects and corresponding two different clients that use them, using options in the command line argument.

C:> BoBCompose −−script=graphcompose.bobs −−option=1

Listing 10.1: *Script graphcompose.bobs*

```
BoBScript (option) {

        switch (option) {
           case 1:
                 BoBClass Graph = Graph_Core + Graph_Undirected + Graph_Weighted;
                 break;
           case 2:
                 BoBClass Graph = Graph_Core + Graph_Directed + Graph_Unweighted;
                 break;
           default: console ("Wrong␣Option!"); break;
        }

        switch (option) {
           case 1:
                 BoBClass Client =  {
                                              public static void Main(args[]) {
                                                   Graph g = new Graph();
                                                   g.readInGraphValues(args[0]);
                                                   g.displayUnweighted( );
                                              }
                                         }
                 break;
           case 2:
```

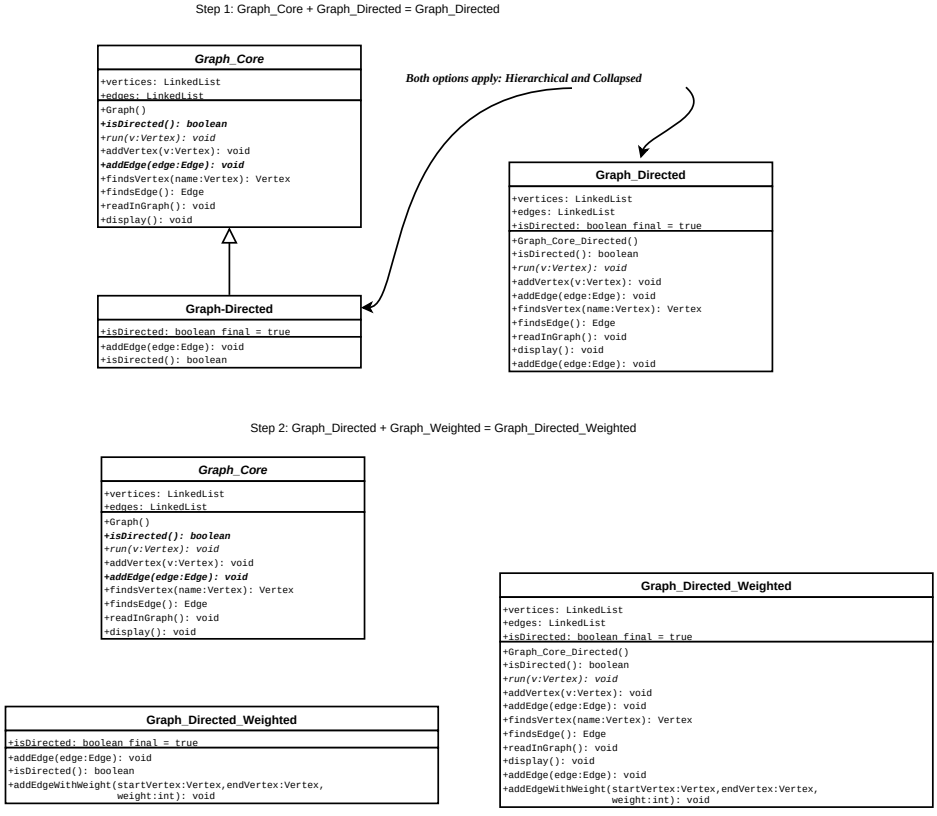


Figure 10.5: *Simple GPL BoB*

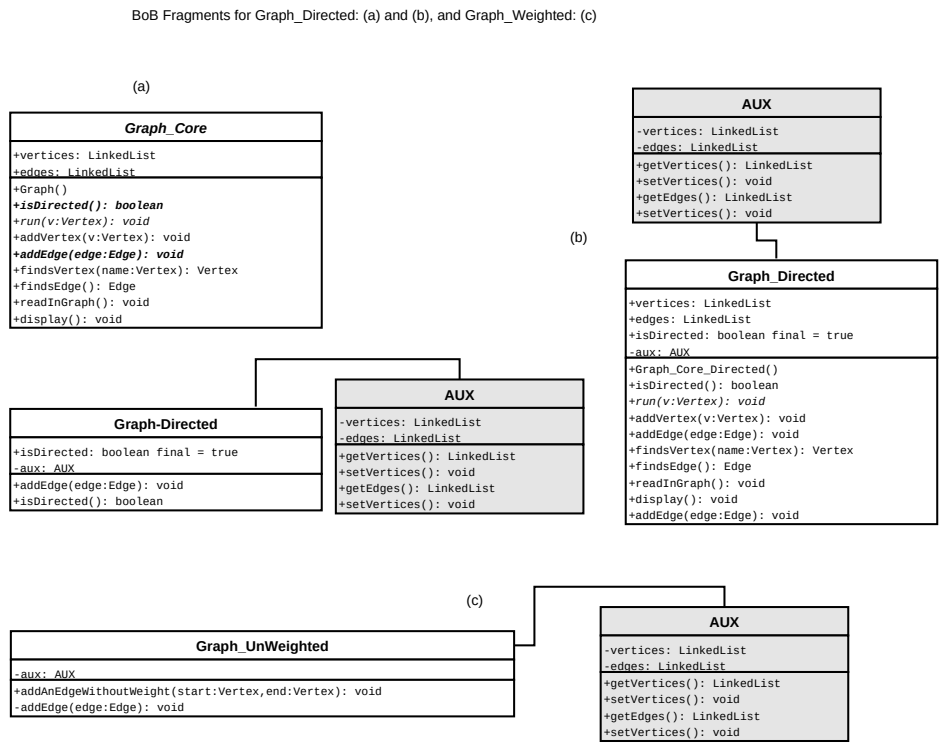


Figure 10.6: *Simple GPL BoB Fragments*

```
                BoBClass Client =  {
                                        public static void Main(args[]) {
                                             Graph g = new Graph();
                                             g.readInGraphValues(args[0]);
                                             g.displayWeighted( );
                                        }
                                   }
                break;
        }

           default: console(‘‘Wrong Option!’’); break;

        }
}
```

This script creates two BoB classes, `Graph.bob` and `Client.bob` with features as determined by the `option` parameter.

10.3.2 Using BoB Programming Language extenstions

The operations of BoB addition and BoB substraction have been included in an extension of programming language Java$_{BoB}$. A typical is use is shown in the example below.

Listing 10.2: *Source snippets from:* `GraphOperations.bob`

```
public BoBClass GraphOperations
{
   . . .

   public static void main (args[]) {

     BoBClass Graph_UW = Graph_Core + Graph_Undirected + Graph_Weighted;
     Graph_UW g_uw = new Graph_UW();
     g_uw.display();

     BoBClass Graph_DW = Graph_UW - Graph_Undirected + Graph_Directed;
     Graph_DW g_dw = new Graph_dw();
     g_dw.dispaly();

     BoBClass Graph_DuW = Graph_DW - Graph_Weighted + Graph_UnWeighted;
     Graph_DuW g_duw = new Graph_DuW();
     g_duw.display();

}
```

The pre-processor in invoked by the command: **C:> javabobc GraphOperations.bob**. It creates Java source code files: `Graph_UW.java`, `Graph_DW.java`, and `Graph_DuW.java` which are used in the `GraphOperations.bob`. It also produces a `GraphOperations.java` file as shown in the listing below.

Listing 10.3: *Source snippets from:* `GraphOperations.java`

```
public Class GraphOperations
{
   . . .

   public static void main (args[]) {

     Graph_UW g_uw = new Graph_UW();
     g_uw.display();

     Graph_DW g_dw = new Graph_dw();
     g_dw.dispaly();

     Graph_DuW g_duw = new Graph_DuW();
     g_duw.display();

}
```

10.4 Discussion: Object Morphing

This is the methodology that BoB based programming emphasises - start with an existing basic shape and obtain new morphed versions by doing simple addition and

subtractions (with the addition of some glue code in some instances).

Thus programming using BoBs makes it a logical and intuitive flow leading to a more *managed evolution* of software. programs.

10.5 Designing a Distance Evaluation Application

10.5.1 DE Application scenario

We consider an examination scenario where a large number of students are e-evaluated concurrently. A typical large-scale examination process involves the following stages: (i) preparation of question papers by gathering inputs from various paper-setters who may work at their respective remote locations, (ii) dispatch of question papers to the examination centers and distribution to the enrolled students, (iii) collection of answer papers and their dispatch to the evaluation center, (iv) evaluation of answer papers by the designated evaluators, and (v) compilation and publication of the results.

Our earlier approach: using Mobile Agents

Over the past few years, the Mobile Agent paradigm has emerged as a new mechanism for structuring distributed applications. It promises to alleviate many of the shortcoming of the client-server approach Mobile Agent (MA) is an autonomous piece of software that can migrate between the various nodes of the network and can perform computations on behalf of the user. Some of the benefits provided by MAs include reduction in network load, overcoming network latency and disconnected operations.For our application, Mobile Agents seemed particularly useful because they map directly to real life situations, are dynamic autonomous entities, and can work in both push and pull modes. We have designed and implemented MADE, a Mobile Agent based system for distance evaluation. We used the Voyager framework to implement our system.

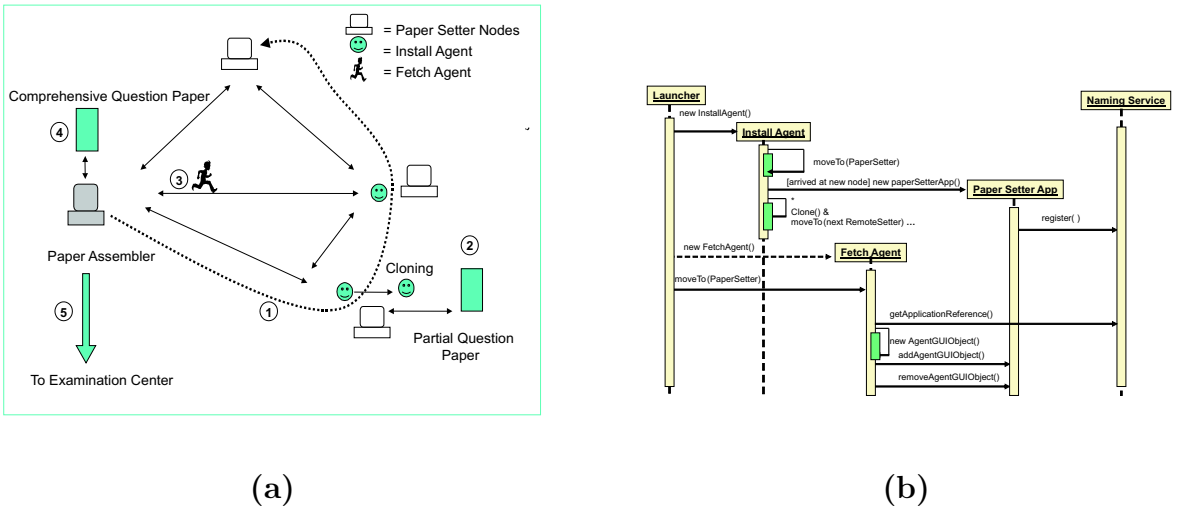
10.5.2 MADE Overview

MADE [Jam03] is a Mobile Agent based system for distance evaluation of students. It was designed with a view to map closely to the real world scenario. Other goals include automation and integration of the entire examination process, minimization of infrastructure requirements at different nodes, and ease of deployment

and maintenance. In MADE we divide the examination process into three stages: (i) paper-setting, (ii) distribution and testing and (iii) evaluation and result compilation.

Paper setting

As shown in Figure 10.7, the examination setting process takes place in a collaborative manner among the paper-setters who are at different remote locations. Install Agents are used to install the paper-setting application on each setter's machine (Step 1). Each setter prepares a partial question paper (Step 2). Fetch Agents are subsequently dispatched to collect these question papers (Step 3). The Paper Assembler node creates one/more comprehensive question paper from the partial papers (Step 4). One of this question papers is sent to the examination centers at the appropriate time (Step 5).



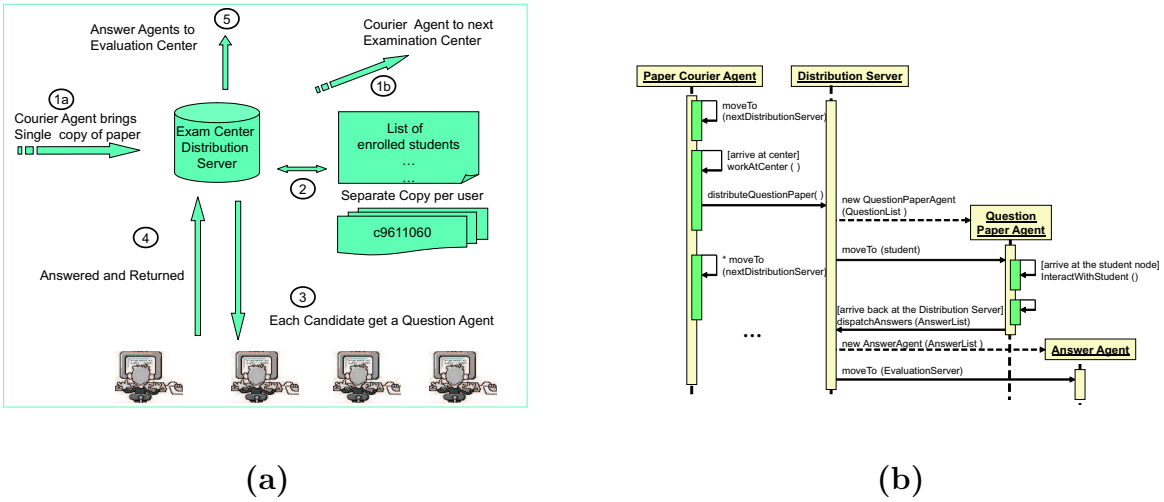
(a) **(b)**

Figure 10.7: *MADE - Paper Setting (a) Scenario, (b) Component Interactions*

Distribution and testing

As shown in Figure 10.8, this stage involves sending the question paper to different centers, distribution to students and the collection of answer papers. The question paper is dispatched to the different examination centers with the help of Courier Agents (Step 1a, 1b). The Distribution Server at each center has a list of candidates enrolled for that center. It creates Question Agents (one per student) and dispatches them to each student node in the center (Step 2, 3). After the designated examination duration or when the student finishes, each Question Agent returns to the Distribution

Server with the student's answers (Step 4). The Distribution Server now creates an Answer Agent for each answer-paper (Step 5), and sends it to the Evaluation Server.



(a) **(b)**

Figure 10.8: *MADE - Paper Testing (a) Scenario, (b) Component Interactions*

Evaluation and result compilation

As shown in Figure 10.9, this stage involves evaluation of the answer papers, compilation of the results and their publication. When an Answer Agent reaches the Evaluation Server, it is supplied with an itinerary of evaluators (Step 1). The Answer Agent visits various evaluators, until all the answers are evaluated (Step 2). Finally the Answer Agent moves to the Publishing Server where it supplies its results (Step 3). The comprehensive results are then compiled and published (Step 4).

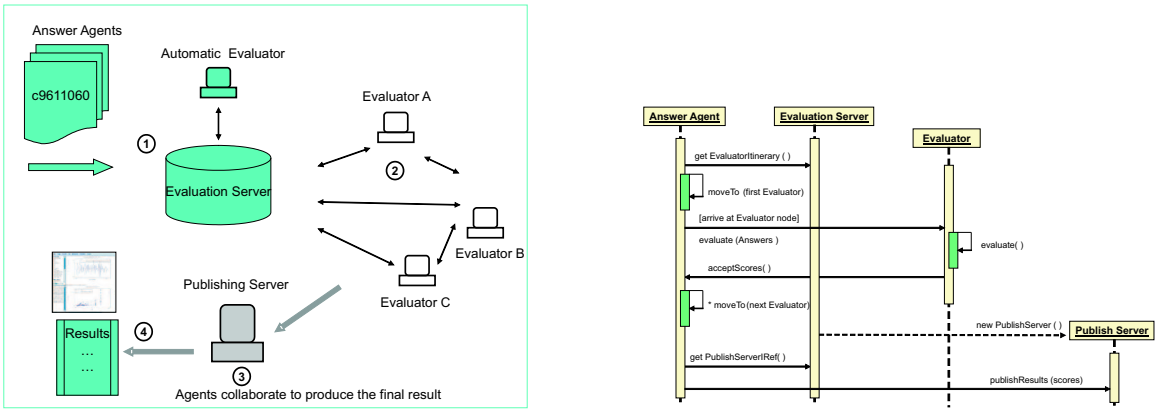
The detailed design and implementation aspects of MADE are available in [Jam03].

10.6 Design using Breakable Objects

Places where BoB are employed in DE application:

1. Question Paper Adaptation - Different versions of question paper
2. Question Paper GUI adaptation
3. Dynamically building and decomposing BoBs.

We show how GUI is to be built for partitioning.



(a) (b)

Figure 10.9: *MADE - Paper Evaluation (a) Scenario, (b) Component Interactions*

```
/*
 * TabbedPaneDemo.java is a 1.4 example that requires one additional file:
 *   images/middle.gif.
 */

import javax.swing.JTabbedPane;
import javax.swing.ImageIcon;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JFrame;
import javax.swing.JComponent;
import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.GridLayout;
import java.awt.event.KeyEvent;

public class BoBTabbedPaneDemo extends JPanel {
    private static  JTabbedPane tabbedPane; // = new JTabbedPane();
     private   ImageIcon icon = createImageIcon("images/middle.gif");

    public BoBTabbedPaneDemo() {
        super(new GridLayout(1, 1));

        tabbedPane = new JTabbedPane();
        //        add(tabbedPane);
    }

    public void addFirstTab() {

        JComponent panel1 = makeTextPanel("Panel \#1");
        tabbedPane.addTab("Tab 1", icon, panel1,
                          "Does nothing");
        //        tabbedPane.setMnemonicAt(0, KeyEvent.VK_1);
```

```
    //    add(tabbedPane);

    //Uncomment the following line to use scrolling tabs.
    //tabbedPane.setTabLayoutPolicy(JTabbedPane.SCROLL_TAB_LAYOUT);
}

 public void addRestofTabs() {

    JComponent panel2 = makeTextPanel("Panel #2");
    tabbedPane.addTab("Tab 2", icon, panel2,
                      "Does twice as much nothing");
    // tabbedPane.setMnemonicAt(1, KeyEvent.VK_2);

    JComponent panel3 = makeTextPanel("Panel #3");
    tabbedPane.addTab("Tab 3", icon, panel3,
                      "Still does nothing");
    //tabbedPane.setMnemonicAt(2, KeyEvent.VK_3);

    JComponent panel4 = makeTextPanel(
            "Panel #4 (has a preferred size of 410 x 50).");
    panel4.setPreferredSize(new Dimension(410, 50));
    tabbedPane.addTab("Tab 4", icon, panel4,
                      "Does nothing at all");
    //tabbedPane.setMnemonicAt(3, KeyEvent.VK_4);

}

protected JComponent makeTextPanel(String text) {
    JPanel panel = new JPanel(false);
    JLabel filler = new JLabel(text);
    filler.setHorizontalAlignment(JLabel.CENTER);
    panel.setLayout(new GridLayout(1, 1));
    panel.add(filler);
    return panel;
}

/** Returns an ImageIcon, or null if the path was invalid. */
protected static ImageIcon createImageIcon(String path) {
    java.net.URL imgURL = TabbedPaneDemo.class.getResource(path);
    if (imgURL != null) {
        return new ImageIcon(imgURL);
    } else {
        System.err.println("Couldn't find file: " + path);
        return null;
    }
}

/**
 * Create the GUI and show it.  For thread safety,
 * this method should be invoked from the
 * event-dispatching thread.
 */
private static void createAndShowGUI() {
    //Make sure we have nice window decorations.
    JFrame.setDefaultLookAndFeelDecorated(true);
```

```
        //Create and set up the window.
        JFrame frame = new JFrame("TabbedPaneDemo");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        //Create and set up the content pane.
        JComponent newContentPane = new BoBTabbedPaneDemo();


         newContentPane.setOpaque(true); //content panes must be opaque


       //New methods added

         BoBTabbedPaneDemo newTP = new BoBTabbedPaneDemo();

         newTP.addFirstTab();
         //newTP.addRestofTabs();
         newTP.add(tabbedPane);

           frame.getContentPane().add(newTP, //new TabbedPaneDemo(),
                                BorderLayout.CENTER);

        //Display the window.
        frame.pack();
        frame.setVisible(true);
    }

    public static void main(String[] args) {
        //Schedule a job for the event-dispatching thread:
        //creating and showing this application's GUI.
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                createAndShowGUI();
            }
        });
    }
```

When each thing is unique in itself, there can be no comparison made.... There is only this strange recognition of present otherness.

D.H. Lawrence

Chapter 11

Related Work Comparison

We discuss here the concepts and works which are related to our work. We also indicate the manner in which these efforts relate to our work.

11.1 Objects

As already mentioned in this thesis, BoB is similar to an object, except that we view it as a *breakable* object. In BoB we can designate some methods as inseparable by the keyword `together`. So an object can be considered as one extreme case of a BoB where *all* its interface methods are together. Another extreme is when *none* of the public methods are `together`. This gives a user the maximum flexibility to configure the fragments.

Some of the features of objects like multi-threading, and polymorphism require a more disciplined used in the context of BoBs. Similarly, some mechanisms like reflection pose special difficulties and are not presently supported by BoBs.

11.1.1 Fine Grained Objects/Components

One can design an application in a manner that it is made up of components of extremely fine granularity. Though this approach will give flexibility in functionality composition, we believe building applications this way creates very cumbersome and unintuitive designs. Our approach gives the choice to create fine grained components on demand.

11.1.2 Fragmented Objects (FOs)

BoBs are FOs [Mak94] are two distinct concepts serving different purposes. While a FO is a distributed shared object by definition, BoBs are *objects that can be readily split*. The fragments may exist in a single address space or multiple address space. BoBs do not carry any notion of distribution per say. FOs are monolithic in nature. BoB splits can be dynamically specified.

11.2 Application Partitioning

Application partitioning, has been active area of research in the last decade. For example, J-orchestra [Til02], Pangaea [Spi99] , Addistant [Tat01] try to automate application partitioning of arbitrary Java programs, Coign [Hun98] 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 start from the first principles and 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 *declarative* way of application partitioning. Additionally the granularity level in these systems is fixed by the objects or components of the programming language/system they use, while in our case, granularity level of objects is dynamically decided and is related to the methods of a BoB.

11.3 Multidimensional Separation of Concerns (MDSOC)

A concern is a piece of interest or focus in a program. Separation of concern is a fundamental software engineering principle which states that a given problem involves different kinds of concerns, which should be identified and separated to cope with complexity [Aks01]. This is done to achieve the required quality factors such as robustness, adaptability and reusability. The concept of multi-dimensional separation of concerns(MDSOC) was proposed in [Tar99] [Oss01b]. The key idea is to *simultaneously* support or partition various concerns in a software system along multiple dimensions of composition and decomposition. This includes overlapping and interacting concerns. Most of the software formalisms allow separation of concerns, but there is a tendency towards a single, main dimension. This is also referred to as

tyranny of dominant decomposition. MDSOC tries to ameliorate some of these limitations. Once such approach is *hyperspaces*. In it, based on a concern a set of units can be selected to from modules called *hyperslices*. Thus a hyperslice encapsulates concerns. Relationships between hyperslices can be specified, and they can be used to control flexible composition of hyperslices into *hypermodules*. Sets of hyperslices thus represent different decompositions of the software and by composing them in a desired manner, components and systems can be built.

Hyper/J [Oss01a] supports MDSOC for Java using hyperspaces approach. It allows: (i) specification of Java files to consider, (ii) specification of concern mappings, what are the concerns and what classes are members affect them, (iii) specification of concerns to be encapsulated in a hyperslice. (iv) specification of composition mechanisms for hyperslice compositions into hypermodules, (v) generating java classes for hypermodules.

BoBs provide a concrete formulation of many of the abstractions that MDSOC uses. For example, the units can be BoB classes or BoB Fragments. This implies that units of finer granularity can not be created. BoB's concept of shared (AUX) variables between two partitions provides a realization and implementation of abstract declarations. Further more, BoB composition rules define the dimensions of unit compositions and decompositions in a flexible, yet concrete and systematic manner.

11.4 Fragmentation in Object-Oriented Database Systems

The problem of distributed database design comprises first, the fragmentation of database entities, and second the allocation of these entities to distributed sites. In the relational database environment, the entity of distribution is a *relation*, while in a distributed object based database system, entity of fragmentation and distribution can be a *class*. Classes can be divided into fragments which can be later distributed. A class is a ordered relation $\mathcal{C} = (\mathbf{K}, \mathcal{A}, \mathcal{M}, \mathcal{I})$ where K is the class identifier, A the set of attributes, $\mathcal{M}$ the set of methods and $\mathcal{I}$ is the set of objects using $\mathcal{A}$ and $\mathcal{M}$. A fragment is a subset of the whole class extension.

In *horizontal fragmentation* [Eze95] case class instances are distributed across fragments. Each horizontal fragment (C_h) of a class contains all attributes and methods of a class but only some instance objects ($I^{'} \subseteq I$) of the class. Thus $C_h = (\mathbf{K}, A, M, I^{'})$. In BoB splitting does not occur along these lines.

In *Vertical Fragmentation* [Eze98] the attributes and methods are distributed across

fragments. The main motivation is to reduce irrelevant data accessed by applications. It is similar to the way we approach splits of a BoB class. However, in vertical fragmentation of classes (in the work that has appeared so far), fragmentation is done only for unshared fields of the class cases. We find this to be a limiting feature when we consider the classes and objects as a basic entities for programming.

Each vertical fragment C^v of a class contains its class identifier, and all of its instance objects for only some of its attributes ($A^{'} \subseteq A$), and some of its methods ($M^{'} \subseteq M$). Thus $C_v = (\mathbf{K}, A^{'}, M^{'}, I)$

Hybrid Fragmentation: Each hybrid fragment (C_h^v) of a class contains its class identifier, some of its instance objects ($I^{'} \subseteq I$) for only some of its attributes ($A^{'} \subseteq A$) and some of its methods ($M^{'} \subseteq M$). Thus $(C_h^v) = (\mathbf{K}, A^{'}, M^{'}, I^{'})$

BoBs look at fragmentation from the programming perspective, whereas fragmentation in Object Databases concentrates on fragmentation of data, as stored in the form of objects. In Horizontal fragmentation of database objects, fragmentation occurs by separating out a particular set of instances of a class from the others instance object of that class. The focus is on how to choose the lines of split for optimal queries, whereas we abstract this out in the form of a split-configuration file and as separate from the basic BoB concept.

Summarizing, we can say that BoBs can be easily used to provide vertical fragmentation in distributed object oriented databases. Horizontal fragmentation, on the other hand, is an orthogonal concept to the mechanisms of splitting as specified in BoBs and does not have any bearing on the mechanisms employer in BoB per say.

11.5 Class Refactorings

Different methods of refactorings have been proposed in literature [Opd92] [Bec97] [Fow99]. [Men04] does a comprehensive survey and elaborates these refactoring techniques. Class refactoring methods like *extract class*, *extract interface* etc. [Fow03] provide a means to refactor classes for improving designs of the existing code. However no comprehensive techniques exist to provide refactorings for redeployment as discussed in this thesis.

11.6 Class Extensibility Mechanisms

Various mechanisms have been proposed for extending the classes in object-oriented programs. The usage of inheritance and aggregation (containment, composition) are the most popular usage. Inheritance can be *single* or *multiple*. Single inheritance is

the mechanism by which one class (called the derived class) acquires the properties (data and operations) of another class (called the base class). In multiple inheritance, one class acquires the properties of two or more base classes. We use single inheritance in BoB as more as a mechanism for preserving the refinement changes, rather than for providing polymorphic support. Polymorphic support can be provided with *interface* inheritance, by declaring that interface as atomic by using `together` specification. Multiple inheritance is not allowed. Containment is a useful mechanism for creating BoBs, but a decision has to be made, if the containments are, in turn, are breakable or atomic. Other mechanisms for class extensibility have been proposed e.g. *mixins* [Fla98] [Anc03] , which act as abstract subclasses and *traits* [Sch03], which provides mechanisms for behavioral compositions. The latter also provides mechanism for addition and removal of traits from a class. Mixins pose same challenges for us a single inheritance. The traits, though providing a more flexible mechanism, do not allow us to directly apply their composition mechanisms because the states are exclusive to traits. Once flattened to a single class, however, the manner of these compositions becomes orthogonal to subsequent decomposition, and we can treat the resultant classes as normal classes. Another mechanism for collective extensions of collaborating class called *mixin layers* is proposed in [Sma02]. BoB composition mechanisms, implicitly use a similar concept when we use the most specialized field instances during a fragment's addition to a BoB. In this thesis, we have provided BoB composition rules and mechanisms. We also provide rationale and correctness of their use. However, a rigorous type implication study of these techniques, is an area of future work.

11.7 Feature Oriented Programming (FOP)

Feature Oriented Programming (FOP) considers applications as provisions for a set of features. Feature-oriented programming is particularly useful in the applications where a large variety of similar objects is needed. There are few related approaches to this form of programming. One of the prominent one is *Algebraic Hierarchical Equations for Application Design (AHEAD)* [Bat04]. It is an algebraic method for program synthesis based on step-wise refinement. Here a feature refinement (say, f) is a function that takes program (say, P) as an input, and the produces an output that is a program with included feature, in this case $f(P)$. A multi feature application is an equation, e.g. $g(h(f(P))$ for features f, g, and$h$. One of the intended goals is that suits of product families be progressively built by using this approach [Liu04]. An alternative approach for stepwise refinement method for feature introduction is discussed in [Bac02]. In [Pre97], a model of feature compositions is proposed, where

a formulation called *feature* is used for feature-compositions in a software. A feature entity is similar to mixin (or abstract class). The main difference from abstract subclasses is that, method overriding is considered in a more general sense and not tied to - a child class overriding the methods of a parent class - semantics. Only two features are combined at a time. Feature interactions are resolved by *lifting* functions (method-overwriting) of one feature to the context of another. The fact that multiple features can be incorporated by considering two features at a time, greatly simplifies the model.

The BoB model of composition directly supports feature combinations by all these different methodologies. In BoB composition, fragment additions and subtractions can take place in any arbitrary order. The composition order of the arguments determines both the order of method overriding/overwriting and the naming resolution of the resultant class. Since a BoB fragment can be conveniently separated out, it is easy to take a *hyperslice* across multiple BoB classes. This is particularly useful, when a feature needs to form a separate concern and is mapped across multiple classes.

11.8 Variability in Software Product Lines

Product lines help to develop a variety of products as a variation from reusable core assets [Cle01]. Variability is the ability of a system to support such variations. Various architecture variability mechanisms exist in practice[Sva02], e.g., (i) component replacement, omission, replication (ii) parameterization (including macros, templates), (iii) compile-time selection of different implementations (e.g., #ifdef), (iv) OO extensibility techniques: inheritance, specialization, (v) configuration and module interconnection languages, (vi) generation and generators, (vii) aspect-oriented programming (viii) application frameworks etc. Some of these are directly supported by the base language, while other require additional meta-level support. E.g.aspect-oriented programming [Kic97], which an approach for modularizing system properties that cross-cut different modules, would require an additional aspect compiler. Similarly a generative programming and configuration approaches require external generator and configurator respectively.

BoBs, by virtue of their design, are meant to be *variable* components. They support variability in a flexible, but controlled manner by addition, removal and replacement of the subset functionalities. They can easily work in conjunction with other-variability mechanisms like compile time selectors, generators, aspects, configurators to produce powerful techniques and mechanisms for introducing variability in a planned and flexible manner in the software product lines. This thesis indicates how variability is

achieved at an intra-component level using BoBs. The full exploration of language and meta-level support techniques, particularly their combinations with other mechanisms, is an area of future research.

11.9 BODA as Distributed Design Paradigm

In this section we discuss the existing distributed design paradigms and then discuss BODA with respect to these paradigms.

For distributed applications that exploit (static/dynamic) reconfiguration of software components, the concepts of location, distribution of components among locations, and migration of components to different locations need to be taken explicitly into account during the design stage [Fug98]. Some of the basic architectural concepts that are an abstraction of the entities that constitute such a software system, are:

.

Components Components are the constituents of a software architecture. They can be further divided into:

- **Code components:**,encapsulate the *know-how* to perform a particular computation
- **Resource components:** represent *data or devices* used during the computation
- **Computational components:** active executors capable to carry out a computation as specified by a corresponding know-how.

Interactions Interactions are events that involve two or more components, e.g., a message exchanged among two computational components.

Sites A site represents the intuitive notion of location. Sites host components and support the execution of computational components.

A computation can be actually carried out only when the know-how describing the computation, the resources used during the computation, and the computational component responsible for execution are located at the same site.

11.9.1 Distributed Design Paradigms

Design paradigms are described in terms of interaction patterns that define the relocation of and coordination among the components needed to perform a service.

As described in[Fug98] , we consider a scenario where a computational component A, located at site SA needs the results of a service. Site SB, provides this service.

Client-Server (CS)

In this paradigm, a computational component B (the server) offering a set of services is placed at site SB. Resources and know-how needed for service execution are hosted by site SB as well. The client component A, located at SA, requests the execution of a service with an interaction with the server component B. As a response, B performs the requested service by executing the corresponding know-how and accessing the involved resources Co-located with B.

Remote Evaluation (REV)

In the REV paradigm, a component A has the know-how necessary to perform the service but it lacks the resources required, which happen to be located at a remote site SB. Consequently, A sends the service know-how to a computational component B located at the remote site. B, in turn, executes the code using the resources available there. An additional interaction delivers the results back to A.

Code on Demand (COD)

In the COD paradigm, component A is already able to access the resources it needs, which are co-located with it at SA. However, no information about how to manipulate such resources is available at SA. Thus, A interacts with a component B at SB by requesting the service know-how, which is located at SB as well. A second interaction takes place when B delivers the know-how to A, that can subsequently execute it.

Mobile Agent (MA)

In the MA paradigm, the service know-how is owned by A, which is initially hosted by SA, but some of the required resources are located on SB. Hence, A migrates to SB carrying the know-how and possibly some intermediate results. After it has moved to SB, A completes the service using the resources available there. The mobile agent paradigm is different from other mobile code paradigms since the associated interactions involve the mobility of an existing computational component.

Table 11.1: *Distributed Computation Paradigms*

Paradigm	SA-*initial*	SB-*initial*	SA-*later*	SB-*later*
Client-Server	A	know-how resource B	A	know-how resource **B**
Remote Evaluation	know-how A	resource B	A	*know-how* resource **B**
Code on Demand	resource A	know-how B	resource *know-how* **A**	B
Mobile Agent	know-how A	resource	-	know-how resource **A**
Breakable Object	know-how A resource	resource	(**A**)-part-1, (knowhow)-part-1 resource	(**A**)**-part-1**, (*know-how*)-part-2 resource

11.9.2 Breakable Object (BoB)

It combines the power of all the paradigms and leaves open the choice of how a computation is distributed inside an object. In the BoB paradigm too, the service know-how is owned by A, which is initially hosted by SA, but some of the required resources are located on SB. But in BoB case, only the requisite part of A that needs to interact locally with the resources at SB is moved (or deployed in case of static distribution) to SB.

11.9.3 Discussion and Comparison

Table 11.9.3 shows the location of the components before and after the service execution. For each paradigm, the computational component in bold face is the one that executes the code. Components in italics are those that have been moved.

Client-Server paradigms are static with respect to code and location. Mobile code paradigms overcome these limits by providing component mobility. By changing their location, components may change dynamically the quality of interaction, reducing interaction costs. To this end, the REV and MA paradigms allow the execution of code on a remote site, enabling local interactions with components located there and COD paradigm enables computational components to retrieve code from other remote components, providing a flexible way to extend dynamically their behavior and the types of interaction they support [Fug98].

BoBs, per say, do not introduce any notion of code-mobility but provide easy mechanisms and lines along with which the component can be broken and distributed across the network nodes. The static or dynamic behavior of the various BoB splits, will depend upon the paradigm that is needed and also being supported by the distributed platform.

In the next chapter we conclude the discussions of this thesis and also provide pointers to new and future directions of work that emerge as result of the notion of Breakable Objects.

My interest in the future is because I am going to spend the rest of my life there.

-Charles F. Kettering

Chapter 12

Conclusions and Future Work

In this thesis we have motivated the need for structuring programs in such a way that they can be easily refactored for deployment in various scenarios. Toward this end, we have developed the notion of Breakable Object (BoB) as an entity in a programming language and also provided a methodology for BoB based application architecture (BODA). We use BODA in two contexts - application paritioning and application evolutions.

More concretely, we have defined a programming model for BoBs in Java, which requires the introduction of one new construct `together` in the language/preprocessor. Additionally, in this thesis we developed BoB *fragments* as means to language level compositions in the context of *object-oriented* applications. BoB compositions raise fragments to the level of first-class entities and provide a flexible fine-grained mechanism for extensibility and variability-control in an application. We also validated these BoB-based programming and restructuring techniques through some real-word application case-studies.

Future Directions of Work: Some of the future directions of our work are:

- *Optimal BoB partitioning and redeployment:* One would like to define the mechanisms for *automatic* deployment of BoBs in various distributed scenarios. This would involve finding optimal distribution settings for a BoB component and its different fragment configurations. Presently the splitting and deployment setups are separate processes. Techniques which allow for splitting and the deployment codes to be inter-weaved are desirable. This would allow optimal implementations of components for redeployable applications. Additionally, mechanisms for object-level partitioning instead of BoB class-level partitionings can be explored to gain a more fine grained control over the application partitioning. Also, a

utility evaluation and feasibility study of incorporating additional programming language features, like threads etc., is one important direction of future research.

- *Refactoring or re-architecting existing applications to BoB driven architectures:* The process is two fold: (i) Identifying and converting existing objects in an application to BoBs, and (ii) Restructuring of rest of the application. Some of the the existing objects can be, with minimal changes, made suitable as BoBs. Others might require non-trivial refactoring or redesign.

  If this involves redistribution of application functionalities in terms of newer interface methods implementations or newer components, the rest of the prgram might also might require non-trivial restructuring. A full investigation of this problem, has thus, naturally become an an area of future interest BODA related work.

- *Integration with feature-oriented and automatic programming:* Another area for further considerations is the integration of BoB compositional mechanisms with feature-oriented and automatic program generation techniques. This can be done both at architectural and implementation levels. This thesis provided the initial directions in this area, but a full integration of techniques is are area of future research.

- *Meta-layer for large-scale program restructuring:* This thesis has provided mechanisms by which application structuring can be separated as concerns of two different layers - a physical layer where the actual application is implemented, and (ii) a meta-layer where program refactoring and program compositions are achieved. Devising a generic methodology for such application developments is, hence, another area of future research.

Summarizing, we can say that instead of visualizing an object as a monolithic entity, it is advantageous to consider it as *breakable* in terms of functionality and yielding split-objects which have factored functionality. The notion of Breakable Objects (BoBs) greatly facilitates *flexible application architecturing*. That is, these applications can now provide functionality partitioning, feature selection and extensibility to a more optimal level of granularity. We believe, that this has direct relevance in ubiquitous computing systems, software product lines and role and feature oriented application developments. It can also help to produce leaner implementations of softwares.

Although, in this thesis we concern ourselves with *class-based* object-oriented programming language models only, the definition of BoB is generic and is applicable to an

object in object based systems, a component in component driven systems, or a service in service oriented systems. Lastly, an important aspect of future work is providing methodologies and processes for BoB-oriented analysis, design and implementation.

Appendix A

Java$_{BoB}$

A.1 Class Declaration

Class Declaration:
ClassModifiers$_{opt}$ *class Identifier ClassBody*

ClassModifiers

ClassModifiers: public

ClassBody and Member Declarations

ClassBody:
ClassBodyDeclarations$_{opt}$
ClassBodyDeclarations:
ClassBodyDeclaration
ClassBodyDeclarations ClassBodyDeclaration
ClassBodyDeclaration:
ClassMemberDeclaration
StaticInitializer
ConstructorDeclaration
TogetherMethodsDeclaration
ClassMemberDeclaration:
FieldDeclaration
MethodDeclaration

A.2 Field Declarations

FieldDeclaration:
FieldModifiers$_{opt}$ Type VariableDeclarators ;
VariableDeclarators:
VariableDeclarator
VariableDeclarators , VariableDeclarator
VariableDeclarator:
VariableDeclaratorId
VariableDeclaratorId = VariableInitializer
VariableDeclaratorId:
Identifier
VariableDeclaratorId [ ]
VariableInitializer:
Expression
ArrayInitializer

Filed Modifiers

FieldModifiers:
FieldModifier
FieldModifiers FieldModifier
FieldModifier:one of
private
final static

A.3 Method Declarations

MethodDeclaration:
MethodHeader MethodBody
MethodHeader:
MethodModifiers$_{opt}$ ResultType
MethodDeclarator Throws$_{opt}$
ResultType:
Type
void
MethodDeclarator:
Identifer (FormalParameterList$_{opt}$)

Formal Parameters

FormalParameterList:
FormalParameter
FormalParameterList , FormalParameter
FormalParameter:
Type VariableDeclaratorId

VariableDeclaratorId:
Identifier
VariableDeclaratorId [ ]

Method Modifiers

MethodModifiers:
MethodModifier
MethodModifiers MethodModifier
MethodModifier:one of
public private
static final synchronized volatile

Method Throws

Throws:
throws ClassTypeList
ClassTypeList:
ClassType
ClassTypeList , ClassType

Method Body

MethodBody:
Block
;

A.4 Constructor Declarations

ConstructorDeclaration:

ConstructorModifiers$_{opt}$ *ConstructorDeclarator*
Throws$_{opt}$ *ConstructorBody*
ConstructorDeclarator:
SimpleTypeName (FormalParameterList$_{opt}$ *)*

Constructor Modifiers

ConstructorModifiers:
ConstructorModifier
ConstructorModifiers ConstructorModifier
ConstructorModifier:one of
public private

Constructor Body

ConstructorBody:
ExplicitConstructorInvocation$_{opt}$ *BlockStatements*$_{opt}$
ExplicitConstructorInvocation:
this (ArgumentList$_{opt}$ *) ;*

A.5 Together Declarations

TogetherDeclarations:
TogetherDeclaration
TogetherDeclarations, TogetherDeclaration
TogetherMethodDeclaration:
MethodSignatureList
MethodSingatureList:
MethodSignature
MethodSignatureList MethodSignature
MethodSignature:
MethodModifiers$_{opt}$
MethodDeclarator Throws$_{opt}$
MethodDeclarator:
Identifer (FormalParameterList$_{opt}$ *)*

Appendix B

ASM Model for Java$_{BoB}$

ASM models for Java$_{BoB}$ described here are derived from those of Java [Wal99][Bor98] [Stä01]. We divide Java$_{BoB}$ into three execution models: namely $Java_I$, $Java_C$ and $Java_O$ denoting the imperative, class-based and object-based execution modules respectively.

B.1 Dynamic state

Dynamic state is given by the following functions:

- $pos : Pos$
- $restbody : Phrase$
- $locals : MAP(Loc, Val)$
- $meth : Class/MSig$
- $frames : (Meth, Restbody, Pos, Locals)*$
- $classState : Class\ toLinked, Initialized, Unusable$
- $globals : Class/Field\ toVal$
- $heap : Ref\ toObject(Class, MAP(Class/Field, Val))$

- where: *pos* gives current position of program execution, *restbody* assigns to each position a phrase, *restbody/pos* denotes currently to be executed sub-term of *restbody* at *pos*, *locals* gives the current value of local variables, *meth* denotes the currently executing method, $frames$ gives the sequence of still to be executed frames on the stack, *classState* records the current initialization status of a class, *globals* yields the value stored under a field specification in a class, and *heap* records the class object together with the field values of an object.

B.2 Syntax of Language Modules

Syntax of $Java_I$

$$
\begin{array}{ll}
Exp & := Lit \mid Loc \mid Uop \quad Exp \mid Exp \quad Bop \quad Exp \\
 & \quad \mid Exp \ ? \ Exp : Exp \mid Asgn \\
Asgn & := Loc = Exp \\
Stm & := ; \mid Asgn; \mid Lab \ : \ Stm \mid \texttt{break}\ Lab; \\
 & \quad \mid \texttt{continue}\ Lab; \mid \texttt{if}\ (Exp)\ Stm\ \texttt{else}\ Stm \\
 & \quad \mid \texttt{while}\ (Exp)\ Stm \mid Block \\
Block & := \{Bstm_1 \ldots Bstm_n\} \\
Bstm & := Type\ Loc; \mid Stm \\
Phrase & := Exp \mid Bstm \mid Val \mid Abr \mid Norm
\end{array}
$$

B.2.1 Syntax of $Java_C$

$$
\begin{array}{ll}
Exp & := \ldots \mid Invk \\
Exps & := \ Exp_1, \ldots, Exp_n \\
Invk & := Meth(Exps) \mid Class \cdot Meth(Exps) \\
Stm & := \ldots \mid Invk; \mid \texttt{return}\ Exp; \mid \texttt{return}; \\
Phrase & := \ldots \mid \texttt{static}\ Block
\end{array}
$$

B.2.2 Syntax of $Java_O$

$$
\begin{array}{ll}
Exp & := \ldots \mid \texttt{null} \mid \texttt{this} \\
 & \quad \mid Exp\ \texttt{instanceof}\ Class \mid (Class)\ Exp \\
Invk & := \ldots \mid \texttt{new}\ Class(Exps) \mid Exp \cdot Meth(Exps)
\end{array}
$$

B.3 Operational Semantics Model

B.3.1 Execution Machine

This is described below:

$$
\begin{aligned}
execJava =& execJava_I \\
& execJava_C \\
& execJava_O
\end{aligned}
$$

$$
\begin{aligned}
execJava_I =& \\
& execJavaExp_I \\
& execJavaStm_I
\end{aligned}
$$

$$
\begin{aligned}
execJava_C =& \\
& execJavaExp_C \\
& execJavaStm_C
\end{aligned}
$$

$$
\begin{aligned}
execJava_O =& \\
& execJavaExp_O
\end{aligned}
$$

$$
\begin{aligned}
context(pos) =& \textbf{if } (pos = firstPos \vee \\
& restbody/pos \in Bstm \cup Exp) \quad \textbf{then} \\
& restbody/pos \\
& \textbf{else} \\
& restbody/up(pos)
\end{aligned}
$$

$$
\begin{aligned}
yieldUp(result) =& \\
& restbody := restbody[result/up(pos)] \\
& pos := up(pos)
\end{aligned}
$$

$$
\begin{aligned}
yield(result) =& \\
& restbody := restbody[result/pos]
\end{aligned}
$$

B.3.2 Execution of Java$_I$ expressions

$$
\begin{array}{l}
execJavaExp_I = \textbf{case} \quad context(pos) \quad \textbf{of} \\
\qquad lit \rightarrow yield(JLS(lit)) \\
\\
\qquad loc \rightarrow yield(locals(loc)) \\
\\
\qquad uop^{\alpha} exp \rightarrow pos := \alpha \\
\qquad uop^{\blacktriangleright} val \rightarrow yieldUp(JLS(uop, val)) \\
\\
\qquad {}^{\alpha}exp_1 bop^{\beta} exp_2 \rightarrow pos := \alpha \\
\qquad {}^{\blacktriangleright}val_1 bop^{\beta} exp \rightarrow pos := \beta \\
\qquad {}^{\alpha}val_1 bop^{\blacktriangleright} val2 \rightarrow \\
\qquad\quad \textbf{if} \quad (bop \in divMod \wedge isZero(val_2)) \quad \textbf{then} \\
\qquad\qquad yieldUp(JLS(bop, val1, val2)) \\
\\
\qquad loc =^{\alpha} exp \rightarrow pos := \alpha \\
\qquad loc =^{\blacktriangleright} val \rightarrow locals := locals \oplus (loc, val) \\
\qquad yieldUp(val) \\
\\
\qquad {}^{\alpha}exp_0 ?^{\beta} exp_1 :^{\gamma} exp_2 \rightarrow pos := \alpha \\
\qquad {}^{\blacktriangleright}val ?^{\beta} exp_1 :^{\gamma} exp_2 \rightarrow \\
\qquad\quad \textbf{if} \quad val \quad \textbf{then} \quad pos := \beta \quad \textbf{else} \quad pos := \gamma \\
\qquad {}^{\alpha}True ?^{\blacktriangleright} val :^{\gamma} exp \rightarrow yieldUp(val) \\
\qquad {}^{\alpha}False ?^{\beta} exp :^{\blacktriangleright} val \rightarrow yieldUp(val)
\end{array}
$$

B.3.3 Execution of Java$_I$ statements

$$
\begin{aligned}
execJavaStm_I = \mathbf{c}ase \quad & context(pos) \quad \mathbf{of} \\
& ;\rightarrow yield(Norm) \\
& {}^{\alpha}exp;\ topos := \alpha \\
& {}^{\blacktriangleright}val;\ to\ yieldUp(Norm)
\end{aligned}
$$

$$
\begin{array}{ll}
\mathtt{b}break \quad lab; & \rightarrow yield(Break(lab)) \\
\mathtt{c}ontinue \quad lab; c & \rightarrow yield(Continue(lab)) \\
lab :^{\alpha}\ stm & \rightarrow pos := \alpha \\
lab :^{\blacktriangleright}\ Norm & \rightarrow yieldUp(Norm) \\
lab :^{\blacktriangleright}\ Break(lab_b) & \rightarrow \mathbf{i}f \quad lab = lab_b \\
 & \quad \mathbf{t}hen \quad yieldUp(Norm) \\
 & \quad \mathbf{e}lse \quad yieldUp(Break(lab_b)) \\
lab :^{\blacktriangleright}\ Continue(lab_c) & \rightarrow \mathbf{i}f \quad lab = lab_c \\
 & \quad \mathbf{t}hen \quad yield(body/pos) \\
 & \quad \mathbf{e}lse \quad yieldUp(Continue(lab_c))
\end{array}
$$

$$
\begin{aligned}
phrase({}^{\blacktriangleright}abr) \rightarrow \quad & \mathbf{if} \quad pos \neq firstPos \wedge \\
& propagatesAbr(restbody/up(pos)) \quad \mathbf{then} \\
& \quad yieldUp(abr)
\end{aligned}
$$

$$
\begin{array}{ll}
\{\} & \rightarrow yield(Norm) \\
\{{}^{\alpha_1}stm_1 \ldots {}^{\alpha_n}\ stm_n\} & \rightarrow pos := \alpha_1 \\
\{{}^{\alpha_1}Norm \ldots {}^{\blacktriangleright}\ Norm\} & \rightarrow yieldUp(Norm) \\
\{{}^{\alpha_1}Norm \ldots {}^{\blacktriangleright}\ Norm^{\alpha_{i+1}} stm_{i+1} \ldots {}^{\alpha_n}\ stm_n\} & \rightarrow pos := \alpha_{i+1}
\end{array}
$$

$$
\begin{array}{ll}
\texttt{if}\ \ ({}^{\alpha}exp)^{\beta}stm1\ \ \texttt{else}\ \ {}^{\gamma}stm2 & \rightarrow pos := \alpha \\
\texttt{if}\ \ ({}^{\blacktriangleright}val)^{\beta}stm1\ \ \texttt{else}\ \ {}^{\gamma}stm2 & \rightarrow \textbf{if}\ \ val \\
 & \quad \textbf{then}\ \ pos := \beta \\
 & \quad \textbf{else}\ \ pos := \gamma \\
\texttt{if}\ \ ({}^{\alpha}True)^{\blacktriangleright}Norm\ \ \texttt{else}\ \ {}^{\gamma}stm & \rightarrow yieldUp(Norm) \\
\texttt{if}\ \ ({}^{\alpha}False)^{\beta}stm\ \ \texttt{else}\ \ {}^{\blacktriangleright}Norm & \rightarrow yieldUp(Norm)
\end{array}
$$

$$
\begin{array}{ll}
\texttt{while}\ \ ({}^{\alpha}exp)^{\beta}stm & \rightarrow pos := \alpha \\
\texttt{while}\ \ ({}^{\blacktriangleright}val)^{\beta}stm & \rightarrow \ \ \textbf{if}\ \ val\ \ \textbf{then}\ \ pos := \beta \\
 & \quad \textbf{else}\ \ yieldUp(Norm) \\
\texttt{while}\ \ ({}^{\alpha}True)^{\blacktriangleright}Norm & \rightarrow yieldUp(body/up(pos))
\end{array}
$$

$$Typex; \rightarrow yield(Norm)$$

B.3.4 Execution of Java$_C$ expressions

$$
\begin{array}{ll}
execJavaExp_C & = \mathbf{c}ase\ \ context(pos)\ \ \mathbf{o}f \\
c \cdot m^{\alpha}(exps) & \rightarrow pos := \alpha \\
c \cdot m^{\blacktriangleright}(vals) & \rightarrow \mathbf{i}f\ \ initialized(c) \\
 & \quad \mathbf{t}hen \\
 & \quad invokeMethod(up(pos), c = m, vals) \\
 & \quad \mathbf{e}lse\ \ initialize(c)
\end{array}
$$

$$
\begin{array}{ll}
() & \rightarrow yield([]) \\
({}^{\alpha_1}exp_1, \ldots, {}^{\alpha_n}\, exp_n) & \rightarrow pos := \alpha_1 \\
({}^{\alpha_1}val_1, \ldots, {}^{\blacktriangleright}\, val_n) & \rightarrow yieldUp \\
 & ([val_1, \ldots, val_n]) \\
({}^{\alpha_1}val_1, \ldots, {}^{\blacktriangleright}\, val_i, {}^{\alpha_{i+1}}\, exp_{i+1}, \ldots {}^{\alpha_n}\, exp_n) & \rightarrow pos := \alpha_{i+1}
\end{array}
$$

$$
\begin{array}{l}
initialize(c) = \\
\textbf{if} \quad classState(c) = Linked \quad \textbf{then} \\
classState(c) := InProgress \\
\quad \textbf{forall} \quad f \in staticFields(c) \\
\qquad globals(f) := defaultVal(type(f)) \\
\qquad invokeMethod(pos; c = <\texttt{clinit}>; []) \\
\textbf{if} \quad classState(c) = Unusable \quad \textbf{then} \\
\quad fail(\texttt{NoClassDefFoundErr})
\end{array}
$$

B.3.5 Execution of Java$_C$ statements

$$
\begin{array}{l}
execJavaStm_C = \textbf{case} \quad context(pos) \textbf{of} \\
\quad \texttt{static} \quad {}^{\alpha}stm \rightarrow \textbf{let} \quad c = classNm(meth) \\
\qquad \textbf{if} \quad c = \texttt{Object} \vee initialized(super(c)) \\
\qquad \textbf{then} \quad pos := \alpha \\
\qquad \textbf{else} \quad initialize(super(c)) \\
\quad \texttt{static} \quad {}^{\alpha}Return \rightarrow yieldUp(Return)
\end{array}
$$

`return` $^{\alpha}exp$;	$\rightarrow pos := \alpha$
`return` $^{\blacktriangleright}val$;	$\rightarrow yieldUp(Return(val))$
`return`;	$\rightarrow yield(Return)$
$lab :^{\blacktriangleright} Return$	$\rightarrow yieldUp(Return)$
$lab :^{\blacktriangleright} Return(val)$	$\rightarrow yieldUp(Return(val))$
$Return$	$\rightarrow$ **if** $pos = firstPos \wedge$ $notnull(frames)$ **then** $exitMethod(Norm)$
$Return(val)$	$\rightarrow$ **if** $pos = firstPos \wedge$ $notnull(frames)$ **then** $exitMethod(val)$

$^{\blacktriangleright}Norm; \rightarrow yieldUp(Norm)$

B.3.6 Execution of Java$_C$ methods

$invokeMethod(nextPos, c/m, values)$

$frames$	$:= push(frames,$
$(meth, restbody, nextPos, locals))$	
$meth$	$:= c/m$
$restbody$	$:= body(c/m)$
pos	$:= firstPos$
$locals$	$:= zip(argNames$ $(c/m), values)$

$exitMethod(result) =$

$$\begin{array}{ll}
\textbf{let} \quad (oldMeth; oldPgm & ; oldPos; oldLocals) \\
 & \quad = top(frames) \\
meth & := oldMeth \\
pos & := oldPos \\
locals & := oldLocals \\
frames & := pop(frames)
\end{array}$$

if $\quad methNm(meth) = \text{''} < \texttt{clinit} > \text{''} \wedge result = Norm \quad$ **then**

$\quad restbody := oldPgm$

$\quad classState(classNm(meth)) := Initialized$

elseif $\quad methNm(meth) = \text{''} < \texttt{init} > \text{''} \wedge result = Norm \quad$ **then**

$\quad restbody := oldPgm[locals(\text{''}\texttt{this}\text{''}) = oldPos]$

else

$\quad restbody := oldPgm[result = oldPos]$

B.3.7 Execution of Java$_O$ expressions

$$
\begin{array}{ll}
execJavaExp_O & = \textbf{case} \quad context(pos) \quad \textbf{of} \\
\quad \texttt{this} & \rightarrow yield(locals(''\texttt{this}'')) \\
\quad \texttt{new} \quad c & \rightarrow \textbf{if} \quad initialized(c) \quad \textbf{then} \\
 & \quad create \quad ref \\
 & \quad\quad heap(ref) := Object(c, \\
 & \quad\quad\quad \{(f, defaultVal(type(f))) \\
 & \quad\quad\quad | f \in instanceFields(c)\}) \\
 & \quad\quad yield(ref) \\
 & \quad \textbf{else} \quad initialize(c) \\
 & \\
{}^{\alpha}exp \ \texttt{instanceof} \quad c & \rightarrow pos := \alpha \\
{}^{\blacktriangleright}exp \ \texttt{instanceof} \quad c & \rightarrow yieldUp(ref \neq null \\
 & \quad \wedge classOf(ref) \preceq c) \\
 & \\
(c)^{\alpha}exp & \rightarrow pos := \alpha \\
(c)^{\blacktriangleright}ref & \rightarrow \textbf{if} \quad ref = null \\
 & \quad \vee classOf(ref) \preceq c \\
 & \quad \textbf{then} \quad yieldUp(ref) \\
 & \\
{}^{\alpha}exp \cdot c/m^{\beta}(exps) & \rightarrow pos := \alpha \\
{}^{\blacktriangleright}ref \cdot c/m^{\beta}(exps) & \rightarrow pos := \beta \\
{}^{\alpha}ref \cdot c/m^{\blacktriangleright}(vals) & \rightarrow \textbf{if} \quad ref \neq null \quad \textbf{then} \\
 & invokeMethod(up(pos), \\
 & c^{'}/m, [ref] \cdot vals)
\end{array}
$$

Appendix C

Publications

Following are the publications that are related to this work.

1. Automated Refactoring of Objects for Application Partitioning. *12th Asia-Pacific Software Engineering Conference (APSEC),Taipei , Taiwan*, December 15-17, 2005. **Authors**: Vikram Jamwal and Sridhar Iyer.

2. Breakable Objects: Building Blocks for Flexible Application Architectures. *5th Working IEEE/IFIP Conference on Software Architecture(WICSA)*, November 6 - 10, 2005, Pittsburgh, Pennsylvania, USA. **Authors**: Vikram Jamwal and Sridhar Iyer.

3. BoBs: Breakable Objects (Poster Paper) *20th Object-Oriented Programming, Systems, Languages And Applications (OOPSLA)*, October 16- 20, 2005, San Diego, California, USA. **Authors**: Vikram Jamwal and Sridhar Iyer.

4. Mobile Agent based Realization of a Distance Evaluation System. *2003 International Symposium on Application and the Internet (SAINT 2003)*, Orlando, Florida, USA, Jan 27-31, 2003. **Authors**: Vikram Jamwal and Sridhar Iyer.

5. Mobile Agents for effective structuring of large-scale distributed applications. *Workshop on Software Engineering and Mobility, ICSE 2001* at Toronto, Canada. **Authors**: Vikram Jamwal and Sridhar Iyer.

Publications currently under submission:

- Fine Grained Application Partitioning Using Breakable Objects. *IEEE Transactions on Software Engineering.* **Authors**: Vikram Jamwal and Sridhar Iyer.

- BODA: Breakable Object Driven Application Architecture. *29th International Conference on Software Engineering, 2007.* **Authors**: Vikram Jamwal and Sridhar Iyer.

References

[Agh90] G. Agha. *ACTORS : A model of Concurrent computations in Distributed Systems*. The MIT Press, Cambridge, Mass., 1990.

[Aks01] M. Aksit, B. Tekinerdogan, and L. Bergmans. The six concerns for separation of concerns. In L. Bergmans, M. Glandrup, J. Brichau, and S. Clarke, editors, *Workshop on Advanced Separation of Concerns (ECOOP 2001)*. Jun 2001.

[Anc03] D. Ancona, G. Lagorio, and E. Zucca. Jam - designing a java extension with mixins. *ACM Trans Program Lang Syst*, vol. 25(5):pp. 641–712, 2003.

[Bac02] R.-J. Back. Software construction by stepwise feature introduction. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB 2002: Formal Specification and Development in Z and B, proceedings of 2nd International Conference of B and Z Users*, no. 2272 in Lecture Notes in Computer Science, pp. 162–183. Springer-Verlag, Grenoble, France, Jan 2002.

[Bac05] F. Bachmann and P. C. Clements. Variability in software product lines. Tech. Rep. CMU/SEI-2005-TR-012, Software Engineering Institute, Carnegie Mellon University, Dec 2005.

[Bat03] D. S. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. In *ICSE*, pp. 187–197. 2003.

[Bat04] D. S. Batory. Feature-oriented programming and the AHEAD tool suite. In *ICSE*, pp. 702–703. IEEE Computer Society, 2004.

[Bec97] K. Beck. Make it run, make it right: Design through refactoring. *The Smalltalk Report*, vol. 6(4):pp. 19–24, Jan 1997.

[Bor98] E. Borger and W. Schulte. Programmer Friendly Modular Definition of the Semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, no. 1523 in LNCS. Springer, 1998.

[Bos99] J. Bosch. Superimposition: A component adaptation technique. *Information and Software Technology*, vol. 41(5):pp. 257–273, Mar 1999.

[Che05] D. Chernuchin, O. Lazar, and G. Dittrich. Comparison of object-oriented approaches for roles in programming languages. In *2005 AAAI Fall Symposium: Roles, an interdisciplinary perspective*. 2005.

[Cle01] P. Clements and L. M. Northrop. *Software Product Lines : Practices and Patterns*. Professional. Addison-Wesley, 2001.

[Cza00] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Boston, 2000.

[Dah00] M. Dahm. Doorastha - A step towards distribution transparency. In *Net.ObjectDays*. Oct 9–10 2000.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[Dor94] D. Dori and E. Tatcher. Selective multiple inheritance. *IEEE Software*, vol. 11(3):pp. 77–85, May 1994.

[Eze95] C. I. Ezeife and K. Barker. A comprehensive approach to horizontal class fragmentation in a distributed object based system. *Distributed and Parallel Databases*, vol. 3(3):pp. 247–272, 1995.

[Eze98] C. Ezeife and K. Barker. Distributed object based design: Vertical fragmentation of classes. *Distrib Parallel Databases*, vol. 6(4):pp. 317–350, 1998.

[Fla98] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *POPL*, pp. 171–183. 1998.

[Fow99] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: improving the design of existing code*. Object Technology Series. Addison-Wesley, 1999.

[Fow03] M. Fowler. Refactorings in alphabetical order, 2003. Available at `http://www.refactoring.com/catalog/index.html`.

[Fug98] A. Fuggetta, G. Picco, and G. Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, vol. 24(5):pp. 342–361, 1998.

[Gac01] C. Gacek and M. Anastasopoules. Implementing product line variabilities. In *Proceedings of 2001 Symposium on Software Reusability : Putting Software Reuse in Context*, pp. 109–117. ACM Press, 2001.

[Gam00] Gamma, Helm, Johnson, and Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 2000.

[Gen03] T. Genssler and V. Kuttruff. Source-to-source transformation in the large. In L. Böszörményi and P. Schojer, editors, *JMLC*, vol. 2789 of *Lecture Notes in Computer Science*, pp. 254–265. Springer, 2003.

[Gos96] J. Gosling, B. Joy, and G. Steele. *The Java^{TM} Language Specification*. Sun Microsystems, 1.0 edn., Aug 1996. Appeared also as book with same title in Addison- Wesleys 'The Java Series'.

[Gos00] J. Gosling, B. Joy, G. L. Steele, and B. Bracha. *The Java language specification*. Java series. Addison- Wesley, Reading, MA, USA, second edn., 2000.

[Gur01] J. van Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In *WICSA*, pp. 45–54. IEEE Computer Society, 2001.

[Hol93] I. M. Holland. *The Design and Representation of Object-Oriented Components*. Ph.D. thesis, Northeastern University, 1993.

[Hun98] G. C. Hunt. *Automatic distributed partitioning of component-based applications*. Ph.D. thesis, University of Rochester. Dept. of Computer Science, 1998.

[Jac97] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse. Architecture, Process and Organization for Business Success*. Addison-Wesley, 1997.

[Jam03] V. Jamwal and S. Iyer. Mobile agent based realization of a distance evaluation system. In *SAINT*, pp. 362–369. IEEE Computer Society, 2003.

[Kan90] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Nov 1990.

[Kic97] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *ECOOP '97—Object-Oriented Programming 11th European Conference, Jyväskylä, Finland, Proceedings*, vol. 1241 of *Lecture Notes in Computer Science*, pp. 220–242. Springer-Verlag, New York, NY, Jun 1997.

[Koj06] S. Kojarski and D. H. Lorenz. Comparing white-box, black-box, and glass-box composition of aspect mechanisms. In *Proceedings of the 9th International*

Conference on Software Reuse, pp. x–x. ICSR9 2006, IEEE Computer Society, Torino, Italy, Jun 11-15 2006.

[Kru92] C. W. Krueger. Software reuse. *ACM Comput Surv*, vol. 24(2):pp. 131–183, 1992.

[Lav95] R. G. Lavender and D. C. Schmidt. Active object: an object behavioral pattern for concurrent programming. *ProcPattern Languages of Programs,*, Sep 1995.

[Leh98] M. M. Lehman. Software's future: Managing evolution. *IEEE Software*, vol. 15(1):pp. 40–44, Jan / Feb 1998.

[Leh01] M. M. Lehman and J. F. Ramil. Evolution in software and related areas. In *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*, pp. 1–16. ACM Press, New York, NY, USA, 2001.

[LH01] R. E. Lopez-Herrejon and D. Batory. A standard problem for evaluating product-line methodologies. In J. Bosch, editor, *Proceedings GCSE '2001*, vol. 2186 of *LNCS*. Springer-Verlag, 2001.

[Lio04] N. Liogkas, B. MacIntyre, E. D. Mynatt, Y. Smaragdakis, E. Tilevich, and S. Voida. Automatic partitioning: Prototyping ubiquitous-computing applications. *IEEE Pervasive Computing*, vol. 03(3):pp. 40–47, 2004.

[Liu04] J. Liu and D. S. Batory. Automatic remodularization and optimized synthesis of product-families. In G. Karsai and E. Visser, editors, *GPCE*, vol. 3286 of *Lecture Notes in Computer Science*, pp. 379–395. Springer, 2004.

[Loh03] D. Lohmann and J. Ebert. A generalization of the hyperspace approach using meta-models. Fachberichte Informatik 4–2003, Universität Koblenz-Landau, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 2003.

[Lud00] A. Ludwig and D. Heuzeroth. Metaprogramming in the large. In G. Butler and S. Jarzabek, editors, *GCSE*, vol. 2177 of *Lecture Notes in Computer Science*, pp. 178–187. Springer, 2000.

[Mak94] M. Makpangou, Y. Gourhant, J.-P. Le Narzul, and M. Shapiro. Fragmented objects for distributed abstractions. In T. L. Casavant and M. Singhal, editors, *Readings in Distributed Computing Systems*, pp. 170–186. IEEE Computer Society Press, Jul 1994.

[Med00] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans Softw Eng*, vol. 26(1):pp. 70–93, 2000.

[Men04] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, vol. 30(2):pp. 126–139, Feb 2004.

[Mez97] M. Mezini. *Variation-Oriented Programming Beyond Classes and inheritance*. Ph.D. thesis, University of Siegen, 1997.

[Mez02] M. Mezini. Towards Variational Object-Oriented Programming: The RONDO Model. Tech. Rep. TUD-ST-2002-02, Software Technology Group, Darmstadt University of Technology, Alexanderstrasse 10, 64289 Darmstadt, Germany, 2002.

[MR02] M. Mikic-Rakic and N. Medvidovic. Architecture-level support for software component deployment in resource constrained environments. In *Component Deployment*, pp. 31–50. 2002.

[MR04] M. Mikic-Rakic. *Software Architectural Support for Disconnected Operation in Distributed Environments*. Ph.D. thesis, University of Southern California, 2004.

[Omm02] R. van Ommering. The Koala component model. In I. Crnkovic and M. Larsson, editors, *Building Reliable Component-Based Software Systems*. Artech House Publishers, Jul 2002.

[Opd92] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, University of Illinois, Urbana-Champaign , IL , USA, 1992.

[Oss01a] H. Ossher and P. Tarr. Hyper/J: Multi-dimensional separation of concerns for Java. In *Proc. 23rd Int'l Conf. on Software Engineering*, pp. 729–730. IEEE Computer Society, 2001.

[Oss01b] H. Ossher and P. Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Communications of the ACM*, vol. 44(10):pp. 43–50, Oct 2001.

[Ous98] J. K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, vol. 31(3):pp. 23–30, Mar 1998.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the Association of Computing Machinery*, vol. 15(12):pp. 1053–1058, Dec 1972.

[Par78] D. L. Parnas. Designing software for ease of extension and contraction. In *Proceedings of the Third International Conference on Software Engineering*, pp. 264–277. IEEE, 1978.

[Par85] D. Parnas, B. Clements, and Weiss. The modular structure of complex systems. *IEEE Trans on Software Engineering*, 1985.

[Phi97] M. Philippsen and M. Zenger. Javaparty - transparent remote objects in java. *Concurrency - Practice and Experience*, vol. 9(11):pp. 1225–1242, 1997.

[Pit00] A. M. Pitts. Operational semantics and program equivalence. In *APPSEM*, pp. 378–412. 2000.

[Pre97] C. Prehofer. Feature-oriented programming: A fresh look at objects. In M. Aksit and S. Matsuoka, editors, *Proceedings ECOOP '97*, vol. 1241 of *LNCS*, pp. 419–443. Springer-Verlag, Jyväskylä, Jun 1997.

[Rie03] M. Riebisch, D. Streitferdt, and I. Pashov. Modeling variability for object-oriented product lines. In F. Buschmann, A. P. Buchmann, and M. Cilia, editors, *ECOOP Workshops*, vol. 3013 of *Lecture Notes in Computer Science*, pp. 165–178. Springer, 2003.

[Sch03] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In *Proceedings ECOOP 2003 (European Conference on Object-Oriented Programming)*, vol. 2743 of *LNCS*, pp. 248–274. Springer Verlag, Jul 2003.

[Sma02] Y. Smaragdakis and D. Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM TOSEM*, vol. 11(2):pp. 215–255, Apr 2002.

[Spi99] A. Spiegel. Pangaea: An automatic distribution front-end for java. In J. D. P. R. et. al., editor, *IPPS/SPDP Workshops*, pp. 93–99. 1999.

[Spi02] A. Spiegel. *Automatic Distribution of Object-Oriented Programs*. Ph.D. thesis, FU Berlin, FB Mathematik und Informatik, December 2002.

[Stä01] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001.

[Ste00] F. Steimann. On the representation of roles in object-oriented and conceptual modelling. *Data Knowl Eng*, vol. 35(1):pp. 83–106, 2000.

[Sva02] M. Svahnberg, J. van Gurp, and J. Bosch. A taxonomy of variability realization techniques. Tech. rep., Blekinge Institute of Technology, Sweden, 2002.

[Sva05] M. Svahnberg, J. van Gurp, and J. Bosch. A taxonomy of variability realization techniques. *Software Practice and Experience*, vol. 35(8):pp. 705–754, Jul 2005.

[Szy02] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, second edition edn., 2002.

[Tar99] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of ICSE '99*, pp. 107–119. Los Angeles CA, USA, 1999.

[Tat01] 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*, vol. 2072 of *Lecture Notes in Computer Science*, pp. 236–255. Springer, 2001.

[Til02] E. Tilevich and Y. Smaragdakis. J-orchestra: Automatic java application partitioning. In *ECOOP*, pp. 178–204. 2002.

[Van96a] M. VanHilst and D. Notkin. Using c++ templates to implement role-based designs. In *ISOTAS '96: Proceedings of the Second JSSST International Symposium on Object Technologies for Advanced Software*, pp. 22–37. Springer-Verlag, London, UK, 1996.

[Van96b] M. VanHilst and D. Notkin. Using role components to implement collaboration-based designs. In *OOPSLA '96 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications*, pp. 359–369. ACM Press, 1996.

[Ved06] A. Veda. *Application Partitioning - A Dynamic, Runtime, Object-level Approach*. Master's thesis, KR School of IT, IIT Bombay, 2006.

[Wal99] C. Wallace. The semantics of the java programming language: Preliminary version, Dec 30 1999.

[Zha04] H. Zhang and S. Jarzabek. XVCL: a mechanism for handling variants in software product lines. *Science of Computer Programming*, vol. 53(3):pp. 381–407, 2004.

O Give yee thanks unto the Lord,
Because that good is hee:
Because his loving kindenes lasts
To perpetuitee.

Bible: Hebrew Psalm CVII

Acknowledgement

This perhaps is the most difficult of all the thesis chapters to write, but it has another quality too. In a thesis devoted to technical matters, this one allows one to write at a meta-level.

My first thanks to my guide Prof. Sridhar Iyer. There is a long list of *"this wouldn't have been possible... had it not been for him to be my guide"*. To list few: my desire to do a Ph.D. at KR School of IT, IIT Bombay, my choosing BoBs as the Ph.D. topic, his allowing me to spend endless hours thinking in abstract, pushing me to the other extreme of having to go to the minutest of details and putting all of them formally, allowing me to have 'dram breaks' in between Ph.D. pursuits, letting me *indulge in my topic*... and much more. To sum up his guidance, "There has never been a time when I have gone to him with my problems, technical or otherwise, and not come out a happier person". His ability to open a problem knot amazes me. Thanks Sridhar for all your guidance and support.

My next thanks is to my parents and my sister. They remain my biggest sources of strength and inspiration. They have direct influence on this Ph.D. thesis too. The roller-coaster ride that a Ph.D. is, the journey is greatly soothened only by a constant loving support. But there are other things that contribute.To my father I always look up to for sound engineering and design fundamentals - I have seen him excel at these during all stages of my life. I can have infinite discussions with him on the topic and his insights help to shape my own views on the issues. To my mother I always look up to for 'separation of wheat from chaff -the separation of essential from non-essential', the strength of service, hard work, successful management and execution. My sister will always be my role model in almost all spheres of life. The standards that she sets not only enkindle my spirits , but also leave me in constant awe of her.

My first introduction to the field of object-oriented computing has been through lectures of T. M. Vijayraman at NCST way back in 1996-97. The topic caught my fancy then and there and charm carries over. An interesting event that happened during the same time was the design-pattern discussion group formed by Vinod Kumar

at NCST. A *pattern a week* from the GOF book, opened the world of application structuring. To them I owe most as for the laying the foundations of a fascinating subject.

This Ph.D. has directly benefited from critical comments and evaluations by my Ph.D. programme committee comprising of Prof. S.Ramesh and Prof. Rushikesh Joshi. Discussions and questions raised by various peer researches at OOPSLA and WICSA 2005 opened up many interesting perspectives. Prof. Umesh Bellur's initial support to the idea of BoBs provided the initial confidence to take the plunge. The feedbacks and review comments of various papers by the anonymous referees were perhaps the most valuable part of the thesis shaping process. When Simon Peyton says *"Every review is a gold dust... Be(truly) grateful for criticism as well as praise"*, I cannot but agree more.

The one year sabbatical of Paul Clements of SEI CMU, at IIT Bombay was fortuitously placed event for this thesis. His suggestion to consider BoBs as more general mechanism for product line architectures instead of restricting them to distributed setups, opened up very interesting venues both for me and the BoBs.

It would be missing an entire world if I fail to mention the friends who have been of great help throughout - both in terms of their loving support and the academic discussions. I have subjected Srinath Perur most to my Ph.D. woes, including the proof-reading of various papers. Divya Sitaraman, who has nothing to with computer science and everything to do with biological science, surprisingly proved to be a very interesting buyer of BoB-concepts. She would not only patiently listen but also come up with the most throught provoking questions. To this date she remains the most ardent supporter of BoBs :-). Other colleagues and friends... Chikki, Neeraja, SP, street-play group, BONDA, Raghu, Shantanu (who was my office-mate throughout my stay at IITB), Rahul Jha, the entire research scholar group, the very warm and helpful KReSIT admin-staff, Diwakar Shukla, my hostel-wing-mates, Neeraj, Satya and Deepa, Radhika, Gopa Di, Reema, Preetam, Pallavi, Dhawal, and the entire *dhinchak* group... have provided the much needed other perspectives in life.

I would also like to thank Infosys, as much of my financial worries were taken care by the generous Infosys fellowship grants. Last but not least. My obeisances to Him! ..., and, as I often quote:

Thank God, there is God!

⊞