Breakable Objects: Building Blocks for Flexible Application Architectures

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

Abstract

This paper proposes the concept of Breakable Objects (BoBs) as the building blocks for flexible application architectures. We claim that BoB Driven Architecture (BODA) greatly facilitates automated refactoring of an application for various deployment scenarios.

1. Introduction

Distributed systems are becoming increasingly varied in terms of computing and communication capabilities. The same application may often need to be deployed in diverse scenarios. One would like to design/write an application for one scenario in a deployment independent manner and then simply refactor [2] the application for use in another scenario. For this to happen, one requirement is that application functionality should cleanly separate into deployment-specific component subsets. This is hard to achieve in practice as *some functionality may span across multiple components, or a single component may include parts of multiple functionality*.

To enable automatic refactoring, we need the basic components in a form that can be easily partitioned into sub-entities. Traditional objects are not always suitable for such partitioning and we need some modifications to make them amenable to refactoring. We propose the use of *Breakable Objects - BoBs* for constructing such applications.

2. BoB

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

2.1. Features of a BoB

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

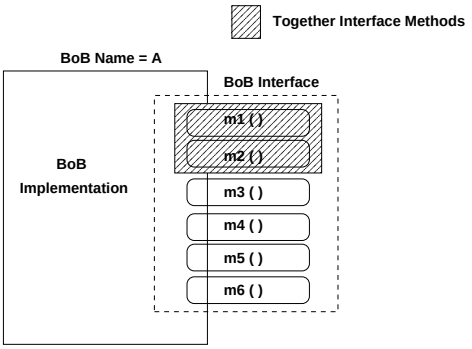

Figure 1. BoB

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

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

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

BoBs are split on the basis of interface methods. External *split-specifications* determine the interface methods which belong to each split.

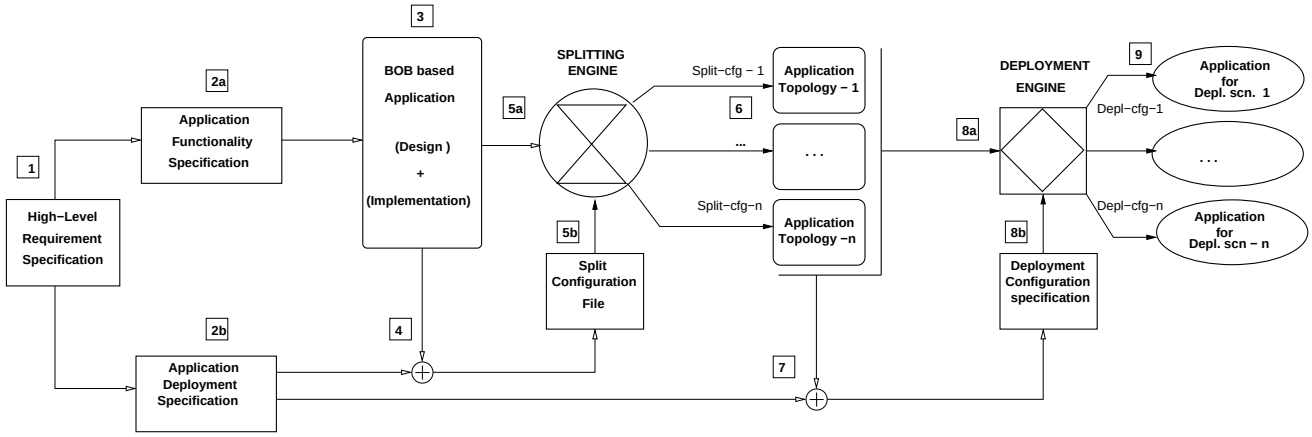

Figure 2. BODA process stages

3. BoB Driven Architecture (BODA)

Stage 1: Design and implementation (steps 1-3, Figure 2) In this stage a *deployment independent* version of the application implementation is prepared.

- The program designer divides the application functionality into a set of - Objects and BoBs. The choice as to which class should be designated breakable is application specific and is based on designer's 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.

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

Stage 2: Splitting and reorganization (steps 4-6 , Figure 2) In this stage application functionality is partitioned into *node-specific subsets of objects*.

- 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.

Stage 3: Distribution and deployment (steps 7-9, Figure 2) In this stage *deployable distributed components* for the new scenario are generated, distributed and deployed.

- Each application object is mapped to a node of the application deployment setup (*deployment-configuration* generation).
- Components are prepared for new distributed environments through source/binary level transformations.
- The resulting application components are distributed and deployed across these nodes.

4. Conclusions and Future Work

The main advantage that BoB Driven Architecture (BODA) offers is the separation of *partitioning and deployment* concerns from the application *design and implementation* concerns. It achieves this through: (i) use of BoBs, (ii) external split and deployment specs, and (iii) automated refactoring of BoB based program. We have automated stage 2. Stage 3 is an ongoing work; it builds on the existing application partitioning systems like J-orchestra [4], and Pangaea [3], which try to automate partitioning of arbitrary Java programs, and Coign [1], which does partitioning of COM based applications.

References

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

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

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

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