Classboxes: Supporting Unanticipated Variation Points in the Source Code

Alexandre Bergel
Hasso-Plattner Institut, University of Potsdam, Germany
Alexandre.Bergel@hpi.uni-potsdam.de

Claus Lewerentz
Software and Systems Engineering Research Group, Technical University of Cottbus, Germany
cl@tu-cottbus.de

Liam O'Brien
National ICT Australia, Canberra, Australia &
LERO, University of Limerick, Ireland
Liam.OBrien@nicta.com.au

Abstract

Software product lines refer to engineering techniques for creating a portfolio of similar software systems from a shared set of software assets in a controlled way. Managing variability is the key issue of software product line practice. Modelling variation points is largely addressed by a selection of linguistic constructs and modelling techniques (*e.g.,* design pattern, macro, configuration files). New constraints and industrial requirements often result in the emergence of new variation points. The success of the evolution of a product line depends on its capability to absorb unanticipated variation points.

This paper presents the *classboxes* programming construct to support unanticipated variation point in the software source code. Classboxes offer a visibility mechanism that controls the scope of an evolution step and limits it only to the part of a program that needs to be affected by this evolution. Benefits of classboxes are illustrated on an arcade game maker product line.

1. Introduction

Product line engineering is a widely used approach for efficient development of whole portfolios of software products [20]. This approach defines a *core asset base* as a collection of artefacts that have been designed specifically for use across the portfolio [4]. Although it is desirable to have a core asset generic enough that it does not require adaptation across products, in many cases adaptations are required. Variation mechanism used in core assets help to control the required adaptations [3].

Variation points are explicitly defined in the core asset to enable variations. For instance, class inheritance and class templates are frequently used to model variations [4]. In this case, variation points are therefore explicit by means of abstract classes and method (in case of inheritance) and template definitions.

Explicitly designating location in a program as variation points in the core asset raises a serious issue regarding the inevitable evolution of a product. All the variation points have to be known and anticipated while the core asset is being defined [5]. As a result, addition of new variation points after the completion of the core asset may come at high cost. Associating variation points to a set of static locations goes against the fact that software evolves in an unpredictable way [8, 14].

Classboxes [7] is a mechanism that supports evolution of product lines by dissociating the core asset with variation point declarations. It is realised as an extension of the Java programming language[1]. A classbox is a modular unit that contains class and method definitions. A classbox may refine another classbox.

In this paper we present how classboxes may be applied to implement product lines and model evolution. The idea is that each classbox represents a software product version. Classboxes advocate the use of a *visibility mechanism* to limit the propagation of changes to a scope, which delimits parts of a software intended to benefit from those changes.

The paper is organised as follow. Section 2 illustrates a situation where creating an unanticipated variant comes at a high cost. This section briefly introduces the Arcade Game Maker example. Section 3 describes the classboxes mechanism, gives its properties and shows how the problems solves the extension problem. Section 4 presents an application of classboxes in a product line. Section 5 points to some related work, and Section 6 concludes the paper.

[1] java.sun.com

2. Unanticipated Software Evolution

2.1 The Arcade Game Maker

The Arcade Game Maker [9] is a product line that allows for creating simple arcade games. It encompasses three games[2]. Each game is a one-player game in which the player controls, to some degree, the moving objects. The objective is to score points by hitting stationary obstacles. The games range from low obstacle count to high and will be available on a variety of platforms.

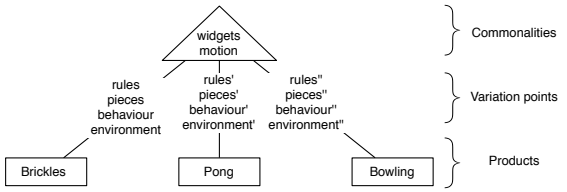

Figure 1. The Arcade Game Maker product line.

As illustrated in Figure 1, the commonalities in this product line are: (i) every game has a set of sprites, (ii) every game has a set of rules, (iii) all the games involve motion. The variations are: (i) rules of the game, (ii) types and numbers of pieces involved, (iii) behaviour of those pieces, and (iv) physical environment in which the game operates.

Brickles, Pong and Bowling are the three products issued from the Arcade Game Maker product line.

2.2 Unanticipated Variation Point

Powerful graphic processors are becoming ubiquitous. For instance, most new cellphones include an advanced graphics display and provide enough processing power to enable three dimensional presentation of games. Business requirements change unexpectedly after initial development and it emerges that the Arcade Game Maker should benefit from the lastest innovation in graphics hardware. One important requirement is that old games must still be obtainable from future versions of the product line.

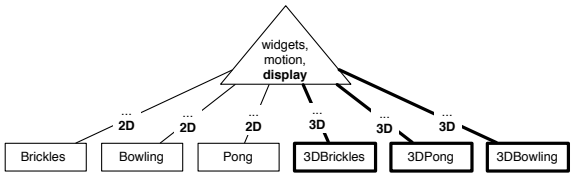

Figure 2. Ideal evolution of the Arcade Game Maker product line (bold strokes represent addition).

The ideal solution is to have a variation point related to the display, a kind of switch that allows for two or three dimensional rendering. Figure 2 illustrates this situation. However, incorporating this kind of evolution may come at high cost:

- Core asset's commonalities have to be modified, which may impact all the products in an unpredictable way.
- Forcing the existing products (*i.e.,* Brickles, Bowling, Pong) to use the new display variation point with traditional programming technologies may lead to some severe refactorings that are difficult to realise.

In the following sections we will present classboxes and show how they provide an approach to deal with evolution in product lines.

3. Classboxes in a Nutshell

A *classbox* is a modular construct supporting software refinements to be defined in a non-invasive way. As a running example, we will use a contrived but compact and representative case study.

Modular unit. A classbox is a kind of module that contains class definitions and class refinements. It defines a namespace of classes. The following example defines a set of classes contained in a classbox Commonalities:

```
classbox Commonalities;
public abstract class Component {
  abstract void draw();
}

public class Point extends Component {
  int x;
  int y;
  public void draw () {
    /* Some code */
  }
}

public class Rectangle extends Component {
  int x1, y1;
  int x2, y2;
  public void draw () {
    /* Some code */
  }
}
```

From a syntactic point of view, the difference between Java and the language used in this example is the introduction of the classbox keyword. The code above defines a classbox named Commonalities that contains three classes, Component, Point, and Rectangle. These classes are meant to represent two dimensional graphical widgets.

Refinements. Contrary to Java packages, definitions contained in a classbox are not fixed, they may be refined into some new versions. Classes defined in a classbox may be imported into another classbox. Moreover, they may be refined in the importing classbox. Imports are specified by the import keyword and refinements by the refine keyword. From the point of view of the importing classboxes, there is no dis-

[2] www.sei.cmu.edu/productlines/ppl/

tinction between classes that are imported and those that are locally defined. The following classbox refines classes defined in Commonalities by adding variables and redefining the draw() method.

```
classbox 3DCommonalities;
import Commonalities.*;

refine Component {
  Texture texture;
}

refine Point {
    /* Class members (like variables x, y) contained
    in Commonalities.Point are accessible */
  int z;
  public void draw () {
    /* New code that use the variables x, y, z and
     texture*/
  }
}

refine Rectangle {
    /* Class members (like variables x1, y1, x2, y2) contained
    in Commonalities.Point are accessible */
  int z1, z2;
  public void draw () {
    /* Code that use the variables x1,y1, z1,
      x2,y2,z2 and texture */
  }
}
```

The class Component is refined with a new variable, texture of type Texture. Definition of the Texture class is not showed in order to keep the example concise. The class Point is refined with a z variable and a draw() method. In a similar fashion, Rectangle is refined with two additional variables (z1 and z2) and a redefined method draw().

Multiple class versions. A classbox defines a particular version of a group of classes. In the example above, two versions of Component, Point, and Rectangle coexist: Commonalities offers a set of two dimensional graphical widgets, and the newly defined 3DCommonalities a set of textured three dimensional widgets.

The class definitions offered by 3DCommonalities are obtained from a concatenation and replacement of class members definitions from Commonalities with class refinements. The definition of 3DCommonalities below is rigourously equivalent to the one presented below:

```
classbox 3DCommonalities;
public abstract class Component {
  Texture texture;
  abstract void draw();
}

public class Point extends Graphics {
  int x;
  int y;
  int z;
  public void draw () {
    /* New code that use the variables x, y, z and
     texture*/
  }
}

public class Rectangle extends Graphics {
  int x1, y1, z1;
  int x2, y2, z2;
  public void draw () {
    /* Code that use the variables x1,y1 z1,
      x2, y2, z2 and texture */
  }
}
```

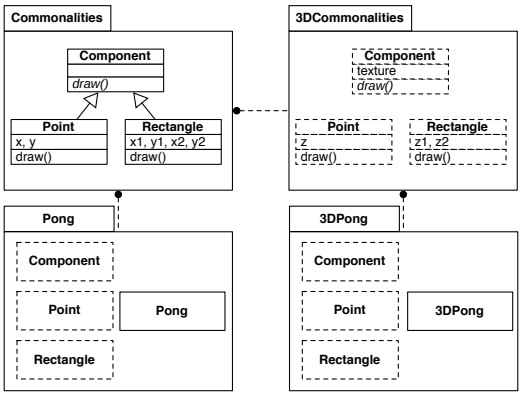

Figure 3. Several clients may rely on different versions of the commonalities.

Since several versions of the same group of classes coexist in the same system, clients may rely on different versions of a common library. Figure 3 illustrates this situation. The classbox Pong imports the original definition of the widgets, while 3DPong imports the refined widgets from 3DCommonalities.

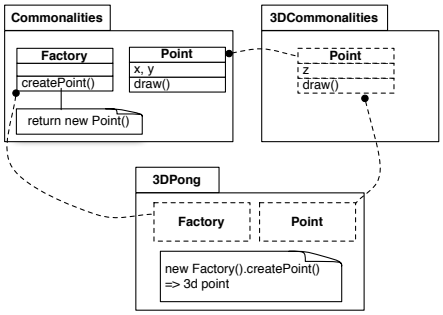

Figure 4. The class Factory is locally rebound in the 3DPong classbox.

Local rebinding. Local class definitions have precedence over the imported definitions. Figure 4 gives an example of a local rebinding: the class Factory is imported in 3DCommonalities without being refined. New points, however, obtained from a factory in this classbox will have the refinements defined in 3DCommonalities.

The local rebinding is a mechanism that puts the "most recent version" of a class in use. By most recent version we mean the class definition that is visible in the current classbox.

4. Classboxes in the Arcade Game Maker Product Line

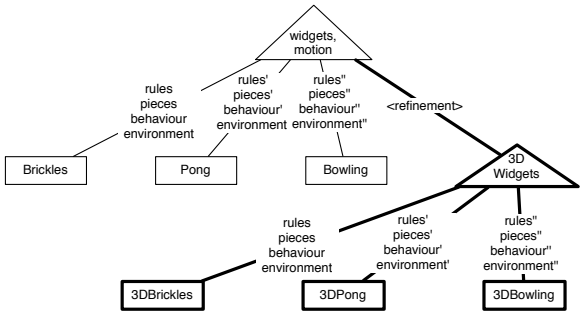

Figure 5. Preferred evolution of the Arcade Game Maker product line (bold strokes represent addition).

A classbox-based approach to incorporate the new variation point is to obtain a second set of commonalities which offers the 3D functionality. Figure 5 represents the new version of the product line. The import between classes is represented in the model with a <refinement> link. It denotes a new version of the commonalities.

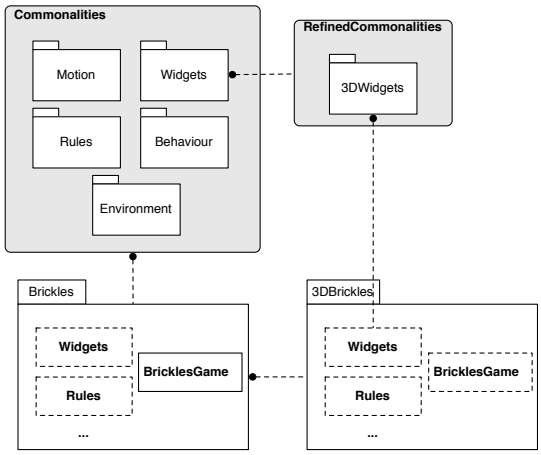

Figure 6. Excerpt of the new classbox-based architecture of the product line.

Figure 6 shows an excerpt of the new version of the Arcade Game Maker product line using a classbox-based architecture. The commonality is defined by a set of five classboxes Motion, Widgets, Rules Behaviour and environment (in the Commonalities box). The classbox 3DCommonalities refines the widgets as illustrated in Figure 3. The 3DBrickes classbox imports the definitions from the 3DBrickes classbox and imports the 3DWidgets from RefinedCommonalities. Within the 3DBrickes classbox the class BricklesGames (imported from Brickles) is locally rebound with definitions visible in 3DBrickles.

Core asset's commonalities have been refined with some 3D widgets, however those refinements are not invasive. The original set of variations are left untouched by the 3D refinements: they are kept separate from the original commonalities.

5. Related Work

This section covers works related to Classboxes in a context of software product line.

Design Pattern. A *Design Pattern* is a general repeatable solution to a commonly occurring problem in software design. It is a description or template for how to solve a problem that can be used in many different situations [11,12]. By means of hooks, most design patterns enable addition and replacement of behaviour without any modification of the base code. For example, after the edification of an abstract syntax tree (AST), any object may traverse a reified program as soon as the protocol defined in the Visitor Pattern is properly implemented. And thus, without any further requirement on the AST.

Although the capability of handling unanticipated evolution of design patterns is widely recognised, the way how such evolution may occur has to be *foreseen* and *explicitly planned*. Whereas a new AST traversal strategy may be easily implemented, the addition of a new AST node results in an adaptation of existing visitors. Such an adaptation may turn out to come at a high cost if the invariant of the original version has to be preserved.

Aspect-oriented programming. Several applications of AOP techniques to product lines have been studied [1]. For example, Liu *et al.* [16] present the benefit of AOP to handle crosscutting variability. They identified desired characteristics of aspect-oriented modelling techniques for product lines.

Concern Hierarchies [17] facilitate the design of aspect-oriented software product lines by supporting derivation of class hierarchies, aspects, and their dependency relations.

Middleware specialisation is also a research topic for which AOP brings significant contributions [13]. Although AOP is primarily used for separation of concerns, it may be used to select the specific set of features needed by a product line. Aspect weaving is subsequently used to specialise the middleware.

One major advantage of classboxes over traditional AOP languages, is the non-invasiveness obtained from the scoping mechanism.

AspectJ[3] proposes inter-type as a mechanism to define class members in a compilation unit different than the one that define the class. For example, methods part of a class C may be defined in an aspect, a location different than where C is defined. This is similar to Classboxes, however, AspectJ does not handle conflict and method redefinition. Thanks to the scoping mechanism of Classboxes, class refinements lives in different scope, which results in no-conflict.

Feature-oriented programming. FOP studies the modularity of *features* in program families [6]. The idea is to build software by composing features. Features are basic blocks and first-class entities in design and implementation. Features may refine other features in an incremental way [2].

Mixin layers [21], AHEAD [6] and FeatureC++ [4] are probably the most representative FOP implementation. Mixin Layers express layered design in terms of *collaborations* and *roles*. A collaboration is typically modelled with a class and a role as an inner class. An inner mixin-class may incrementally refine another role located in another collaboration. The Algebraic Hierarchical Equations for Application Design (AHEAD) model generalises equational specifications to multiple programs and multiple representations. It expresses the code representation of an individual program as an equation. FeatureC++ is a proprietary C++ language extension based on mixin layers.

Feature-oriented programming involve sophisticated techniques to manipulate program representations (typically source code) as a set of artefacts. Different versions of the source code represent multiple variations of a program. However, FOP allows one version of a program to be present when running. Prior the compilation phase, a flattening-like algorithm is applied to reduce the variations of source code into one complete system. On the contrary, Classboxes preserve information related to different version, which enables multiple versions of a group of classes to be present at *execution time*.

Virtual classes. Virtual classes were originally developed for the language BETA [15], primarily as a mechanism for generic programming [18] rather than for expression program variation. Keris [22, 23], Caesar [19], and gbeta [10] offer virtual classes, where method and class lookup are unified under a common lookup algorithm. In a similar way that a method is looked up according to an instance, a class is looked up according to an instance (i.e., an encapsulating class). This implies that the outer class to be instantiated. On the contrary, a Classbox cannot be instantiated. Classboxes does not introduce a class lookup, but enhanced the method lookup algorithm.

The extended version of Java proposed by Classboxes is fully compatible with previous version of Java. As a result, any existing Java program will be accepted by our compiler. This is an important difference with virtual classes since virtual classes usually necessitates the base program to be annotated with classes that are virtual. For example, CaesarJ requires a class to be defined with the cclass to be virtual. On the contrary, Classboxes may operate on any class without any prior preparation.

[3] eclipse.org/aspectj

[4] wwwiti.cs.uni-magdeburg.de/iti_db/fcc

6. Conclusion

Programming languages traditionally used to construct software do not provide means to model evolution and variation in a satisfactory way. Although constructs such as module, package, inheritance, and macro increase reuse and maintenance in software product lines, unanticipated software variation points is not properly addressed.

The work presented in this paper advocates the use of a visibility mechanism to deal with unanticipated variation points. Classboxes enable incorporation of new variation points in a product line without disrupting existing productions.

As future works, we plan to conduct larger case studies and assess more accurately benefits from using a visibility mechanism for software product lines.

Acknowledgments. We would like to thank Matthias Uflacker for his review of an earlier draft of this paper.

We gratefully acknowledge the financial support of Science Foundation Ireland (under grant no. 03/CE2/I303_1).

References

[1] M. Anastasopoulos and D. Muthig. An evaluation of aspect-oriented programming as a product line implementation technology. In *In Proceedings of the 8th International Conference on Software Reuse*, volume 3107 of *LNCS*, pages 141–156. Springer-Verlag, 2004.

[2] S. Apel, T. Leich, and G. Saake. Aspectual mixin layers: aspects and features in concert. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 122–131, New York, NY, USA, 2006. ACM Press.

[3] F. Bachmann and L. Bass. Managing variability in software architectures. In *ACM SIGSOFT Symposium on Software Reusability*, pages 126–132, 2001.

[4] F. Bachmann and P. C. Clements. Variability in software product lines. CMU/SEI-2005-TR-012, Carnegie Mellon University, Software Engineering Institute, 2005.

[5] F. Bachmann, M. Goedicke, J. Leite, R. Nord, K. Pohl, B. Ramesh, and A. Vilbig. A meta-model for representing variability in product family development. In *Proceedings of Europäischen Workshop zur Produktfamilien-Entwicklung (PFE'03)*, volume 3014 of *Lecture Notes in Computer Science*, pages 66–80. Springer-Verlag, 2004.

[6] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling stepwise refinement. In *Proceedings of the 25th international*

conference on Software engineering, pages 187–197. IEEE Computer Society, 2003.

[7] A. Bergel, S. Ducasse, and O. Nierstrasz. Classbox/J: Controlling the scope of change in Java. In *Proceedings of 20th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 177–189, New York, NY, USA, 2005. ACM Press.

[8] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel. Towards a taxonomy of software change. *Journal on Software Maintenance and Evolution: Research and Practice*, pages 309–332, 2005.

[9] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, MA, 2002.

[10] E. Ernst. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Århus, Denmark, 1999.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Mass., 1995.

[12] E. Gamma, R. Helm, J. Vlissides, and R. E. Johnson. Design patterns: Abstraction and reuse of object-oriented design. In O. Nierstrasz, editor, *Proceedings ECOOP '93*, volume 707 of *LNCS*, pages 406–431, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[13] D. Kaul and A. Gokhale. Middleware specialization using aspect oriented programming. In *ACM-SE 44: Proceedings of the 44th annual Southeast regional conference*, pages 319–324, New York, NY, USA, 2006. ACM Press.

[14] G. Kniesel, J. Noppen, T. Mens, and J. Buckley. The first workshop on unanticipated software evolution (use 2002). In *Proceedings of ECOOP 2002 Workshop Reader*, volume 2548 of *LNCS*. Springer Verlag, 2002.

[15] B. B. Kristensen, O. L. Madsen, B. Moller-Pedersen, and K. Nygaard. The BETA programming language. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 7–48. MIT Press, Cambridge, Mass., 1987.

[16] J. Liu, R. Lutz, and H. Rajan. The role of aspects in modeling product line variabilities. In *Proceedings of the 1st Workshop on Aspect-oriented Product Line Engineering (AOPLE' 06)*, Portland, OR, USA, 2006. In conjunction with GPCE'06.

[17] O. S. D. Lohmann and W. Schröder-Preikschat. Concern hierarchies. In *Proceedings of the 1st Workshop on Aspect-oriented Product Line Engineering (AOPLE' 06)*, Portland, OR, USA, 2006. In conjunction with GPCE'06.

[18] O. L. Madsen and B. Moller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 397–406, Oct. 1989.

[19] M. Mezini and K. Ostermann. Conquering aspects with Caesar. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 90–99. ACM Press, 2003.

[20] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, Berlin Heidelberg New York, 2005.

[21] Y. Smaragdakis and D. Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM TOSEM*, 11(2):215–255, Apr. 2002.

[22] M. Zenger. Evolving software with extensible modules. In *International Workshop on Unanticipated Software Evolution*, Malaga, Spain, June 2002.

[23] M. Zenger. Type-safe prototype-based component evolution. In *Proceedings ECOOP 2002*, volume 2374 of *LNCS*, pages 470–497, Malaga, Spain, June 2002. Springer Verlag.