Using AspectJ to Implement Product-Lines: A Case Study

Roberto E. Lopez-Herrejon and Don Batory
Department of Computer Sciences
The University of Texas
Austin, Texas 78712
{rlopez,batory}@cs.utexas.edu

Abstract. *Aspect-Oriented Programming (AOP)* is an emerging technology whose goal is to modularize concerns that cross-cut multiple classes. The purpose of this report is to describe how one of the main representatives of AOP, namely AspectJ, was used to implement a simple yet illustrative product-line of graph algorithms so that we can focus on the implementation details. We expect that studies like this can shed light on the applicability of AOP beyond the traditional examples of logging and debugging.

1 Introduction

A *product-line* is a family of related software applications. A *product-line architecture* is a design for a product-line that identifies the underlying building blocks or *components* of family members, and enables the synthesis of any particular member by composing these components. Different family members (product-line applications) are represented by different combination of components. The motivation for product-line architectures is one of economics and practicality: it is too expensive to build all possible family members; it is much cheaper to build components and to assemble desired family members from them.

Many methodologies have been invented to create product-line architectures and several technologies have been used to implement them (e.g., [2, 4, 10, 12]). In this paper, we explore how AOP techniques could fit in the product-line context. To do that we selected AspectJ [1, 7, 8] and applied it to the problem domain of graph algorithms [9], which is relies on common knowledge yet it is complex enough to highlight the key points of a product-line design.

2 The Graph Product Line (GPL)

The *Graph Product-Line (GPL)* is a family of classical graph applications that was inspired by work on software extensibility [6, 11]. GPL is typical of product-lines in that applications are distinguished by the set of features that they implement, where no two applications have the same set.[1] Also typical is that applications are modeled as sentences of a grammar. Figure 1a[2] shows this grammar, where tokens are names of

1. A *feature* is a functionality or implementation characteristic that is important to clients [5].
2. For simplicity, the grammar does not preclude the repetition of algorithms, whereas the GUI does.

features. Figure 1b shows a GUI that implements this grammar and allows GPL products to be specified declaratively as a series of radio-button and check-box selections.

(a)

```
GPL := Gtp Wgt Src Alg+;

Gtp := Directed | Undirected;

Wgt := Weighted | Unweighted;

Src := DFS | BFS | None;

Alg := Number | Connected | StronglyConnected
     | Cycle | MST Prim | MST Kruskal | Shortest;
```

(b)

Figure 1. GPL Grammar and Specification GUI

The semantics of GPL features, and the domain itself, are straightforward. A graph is either `Directed` or `Undirected`. Edges can be `Weighted` with non-negative numbers or `Unweighted`. Every graph application requires at most one search algorithm: breadth-first search (`BFS`) or depth-first search (`DFS`); and one or more of the following algorithms [3]:

- **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, which are equivalence classes under the reachable-from relation. For every pair of vertices x and y in an equivalence class, there is a path from x to y.
- **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. A cycle in directed graphs must have at least 2 edges, while in undirected graphs it must have at least 3 edges.

- **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.

A fundamental characteristic of product-lines is that not all features are compatible. That is, the selection of one feature may disable (or enable) the selection of others. GPL is no exception. The set of constraints that govern GPL features are summarized in Table 1.

Algorithm	**Required Graph Type**	**Required Weight**	**Required Search**
Vertex Numbering	Directed, Undirected	Weighted, Unweighted	BFS, DFS
Connected Components	Undirected	Weighted, Unweighted	BFS, DFS
Strongly Connected Components	Directed	Weighted, Unweighted	DFS
Cycle Checking	Directed, Undirected	Weighted, Unweighted	DFS
Minimum Spanning Tree	Undirected	Weighted	None
Single-Source Shortest Path	Directed	Weighted	None

Table 1. Feature Constraints

A GPL application implements a valid combination of features. As examples, one GPL application implements vertex numbering and connected components using depth-first search on an undirected graph. Another implements minimum spanning trees on weighted, undirected graphs. Thus, from a client's viewpoint, to specify a particular graph application with the desired set of features is straightforward. And so too is the implementation of the GUI (Figure 1b) and constraints of Table 1.

3 Graph Representation

While deciding how to represent our graphs, we recognized that there are a standard set of "conceptual" objects that are referenced by all graph algorithms: Graphs, Vertices, Edges, and Neighbors (i.e., adjacencies). Algorithms in graph textbooks define fundamental extensions of graphs, and these extensions modify Graph objects, Vertex objects, Edge objects, and Neighbor objects. Thus, the simplest way to express such

extensions is to reify all of these “conceptual” objects as physical objects and give them their own distinct classes.

Therefore we represent a graph with these four classes:

- `Graph`: contains a list of `Vertex` objects, and a list of `Edge` objects.
- `Vertex`: contains a list of `Neighbor` objects.
- `Neighbor`: contains a reference to a neighbor `Vertex` object (the vertex in the other end of the edge), and a reference to the corresponding `Edge` object.
- `Edge`: extends the `Neighbor` class and contains the start `Vertex` of an `Edge`.

Edge annotations are performed by adding extra fields to the `Edge` class. This representation is illustrated in Figure 2. For example, Edge E1 connects vertex V1 to V2 with weight of 7.

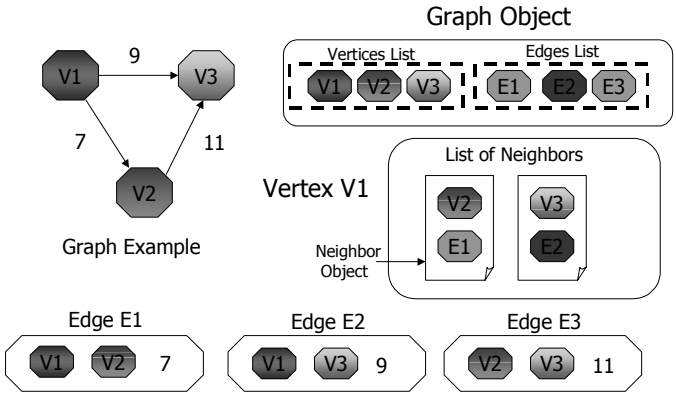

Figure 2. Edge and Neighbor List Representation Example

4 AspectJ Implementation

We implemented GPL with the purpose of exploring how AspectJ can be used to implement product-lines and to compare and contrast it with other methodologies.

Creating an application in AspectJ entails the implementation of two things:

- Base code: consists of the classes and interfaces of a Java program.
- Aspect code: consists of the aspect files that add crosscutting implementation to the base code.

The Base code of GPL consists of the 4 classes that are used to represent a graph: *Graph*, *Vertex*, *Neighbor*, and *Edge*. These classes have empty bodies to which aspects add new fields, methods, and constructors.

Each feature from Figure 1a was implemented by means of an aspect.[3] Recall from Table 1 that some algorithms require a search method. These search methods are called BFS and DFS, and work on WorkSpace objects. Each algorithm that uses any of these search methods declares one class[4] that extends WorkSpace to customize the search for its particular needs. These classes are shown in *italics* in Table 2. For example, the vertex numbering algorithm Number, customizes WorkSpace via the NumberWorkSpace class to implement its functionality.

Directed	directed graph	**Cycle** *CycleWorkSpace*	cycle checking
Undirected	undirected graph	**MSTPrim**	MST Prim algorithm
Weighted	weighted graph	**MSTKruskal**	MST Kruskal algorithm
DFS *WorkSpace*	depth-first search	**Shortest**	single source shortest path
BFS *WorkSpace*	breadth-first search	**Transpose**	graph transposition
Number *NumberWorkSpace*	vertex numbering	**Benchmark**	benchmark program
CC *RegionWorkSpace*	connected components	**Prog**	main program
StronglyCC *FinishTimeWorkSpace* *WorkSpaceTranspose*	strongly connected components		

Table 2. AspectJ Aspects and Classes of GPL

There are three aspects that do not appear in Figure 1a: `Transpose`, `Benchmark`, and `Prog`. `Transpose` performs graph transposition and is used (only) by the `StronglyConnected` algorithm. It made sense to separate the `StronglyConnected` algorithm from `Transpose`, as they dealt with separate concerns. (This means that an implementation constraint in using the `StronglyConnected` aspect is that the `Transpose` aspect must also be included, and vice versa). `Benchmark` contains functions to read a graph from a file and elementary timing functions for profiling. `Prog` contains the main method. It creates the objects required to represent a graph whose elements are read from a file, and starts the execution of the algorithms.

3. Note that the production GPL is not a feature rather, it is the application definition. Also note that Unweighted and None are identity features, that is, they do not add anything to the base code and therefore they do not appear on Table 2.
4. Note that StronglyCC declares two classes since it calls DFS twice with different purpose.

In particular the *run* method of the Graph Class is important. Every algorithmic[5] aspect such as Number, defines a pointcut to add advice to this method, whose empty body is introduced to the Graph class in the Directed or Undirected aspects[6]. In the advice to the run method the aspects execute the algorithm they implement; effectively producing a daisy-chain like effect. Figure 3 illustrates this for the Number aspect.[7]

```
// A point cut to calls to Graph run
pointcut graph_run(Graph g, Vertex v): target(g) && args(v) &&
       call(void Graph.run(Vertex));

// An after advice to run Vertex Numbering
after(Graph g, Vertex v): graph_run(g,v) {
       System.out.println("Running Vertex Numbering ");
       g.NumberVertices();
}

// Effectively runs the vertex numbering algorithm
public void Graph.NumberVertices() {
       NumberWorkSpace nw = new NumberWorkSpace();
       GraphSearch(nw);
}
```

Figure 3. Pointcut for run method in Number aspect

An algorithmic aspect like Number requires that its auxiliary class be declared to extend the WorkSpace class as mentioned before. There are two equivalent ways to do that:

- Use standard Java class extension, that is, declare in the NumberWorkSpace Java file that it *extends* WorkSpace.
- Use the *declare parents* introduction capability as shown in Figure 4.

The customization of WorkSpace is done as follows: the Number aspect introduces to NumberWorkSpace a vertex counter that is going to keep track of the vertex number, then the constructor that initializes it, and finally it defines the behavior of the method preVisitAction which simply increments this counter. Something along these lines is done for the other algorithmic aspects that use search methods. See Figure 4.

Another interesting issue is how weights are handled. The Prog aspect reads a graph from a file. For each edge read, it calls method addAnEdge of class Graph to add it to the class' Edges list. If there are no weights in the graph, the method addAnEdge in Undirected or Directed aspects takes care of the creation of the new weightless edges.

---

5. Algorithmic aspects are: Number, CC, StronglyCC, Cycle, MSTPrim, MSTKruskal, Shortest.
6. Directed and Undirected aspects contain the basic functionality of a GPL application.
7. Here we used an after advice, but we could have used before or around advice as well.

```
// **** NumberWorkspace class extends the WorkSpace class
declare parents: NumberWorkSpace extends WorkSpace;

int NumberWorkSpace.vertexCounter;

public NumberWorkSpace.new() {
       vertexCounter = 0;
   }

public void NumberWorkSpace.preVisitAction(Vertex v) {
       // This assigns the values on the way in
       if (v.visited!=true) v.VertexNumber = vertexCounter++;
   }
```

Figure 4. Extending WorkSpace in Number aspect

However, if there are weights involved, the Weighted aspect must intercept those calls, that is, the addAnEdge default method in either Undirected or Directed aspects must not be called, instead Edge objects with weights must be created. The way to do that is illustrated in Figure 5. There, a pointcut is defined to capture all the calls to the method addAnEdge. The around advice is key because it allows us to effectively override the addAnEdge method defined in Undirected and Directed aspects.

```
// Adds an edge with weights
// Gets the jps of the targets that call addAnEdge in Graph
pointcut graph_addAnEdge(Graph g, Vertex start, Vertex end,
       int weigth):
       target(g) && args(start,end,weigth) &&
       call(void Graph.addAnEdge(Vertex, Vertex, int));

// An around advice to add the weigth of the edge
void around(Graph g, Vertex start, Vertex end, int weigth):
       graph_addAnEdge(g, start, end, weigth) {
       Edge e = new Edge(start, end, weigth);
       g.addEdge(e);
}
```

Figure 5. Adding weighted edges in Weighted aspect

5 Findings

In product-line designs it is the case that not all syntactically valid composition of features are semantically valid. The legal compositions of features in Table 1 are defined by simple constraints called *design rules* [2]. In AspectJ there is no support for design rules, that is, the programmer has to manually select all the files necessary to create a new member, this activity is complex and error prone even for small product-lines.

Aspects can only introduce classes that are private to the aspects and that can only be used in the advices defined in them. This causes that any new classes required by an aspect have to be added manually to the list of files that have to be passed to the weaver which again is error prone and tedious.

6 Appendix

An application that works on a weighted, directed graph that implements vertex numbering, cycle checking, strongly connected components, and shortest path algorithm requires the following files to be passed to the AspectJ compiler:

```
Vertex, Graph, Edge, Neighbor, Directed, Weighted, DFS,
WorkSpace, NumberWorkSpace, Number, CycleWorkSpace, Cycle,
Transpose, FinishTimeWorkSpace, WorkSpaceTranspose,
StronglyCC, Shortest, Benchmark, Prog
```

As can be seen from this example, even for the very simple product lines, the number of files to be considered is large.

6.1 Running examples

The zip file associated with this report contains all the source code of GPL, and a couple of files *Example1.lst* and *Example2.lst* that contain valid examples of GPL applications. To compile them, follow the standard procedure:

```
ajc -argfile Example1.lst
```

This for example creates the application described in the previous section. The generated class files are put in the GPL directory, the package all aspects and classes belong to. The zip file also contains some benchmark files that can be used to run the GPL applications. Prog receives two arguments: the benchmark file, and the starting vertex. For example:

```
java GPL.Prog ./BENCH/MSTExample.bench v0
```

This runs the family member using MSTExample benchmark file, and starts the execution of the algorithms that require a starting point from vertex v0. As result of the execution, the final values of the different fields that the algorithms used are displayed along the execution time that takes to run the application.

7 References

[1] AspectJ. Programming Guide. http://aspectj.org/doc/proguide

[2] D. Batory and B. Geraci. Composition Validation and Subjectivity in GenVoca Generators. *IEEE Transactions on Software Engineering*, February 1997.

[3] T.H. Cormen, C.E. Leiserson, and R.L.Rivest. *Introduction to Algorithms*, MIT Press, 1990.

[4] K. Czarnecki and U.W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.

[5] M. Griss, "Implementing Product-Line Features by Composing Component Aspects", *First International Software Product-Line Conference*, Denver, Colorado., August 2000.

[6] I. Holland. "Specifying Reusable Components Using Contracts", *ECOOP 1992*.

[7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, "Aspect-Oriented Programming", ECOOP 97, 220-242.

[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kirsten, J. Palm, W.G. Griswold. "An overview of AspectJ". In Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP). Springer, 2001.

[9] R. E. Lopez-Herrejon, D. Batory. "A Standard Problem for Evaluating Product-Line Methodologies". Third International Conference on Generative and Component-Based Software Engineering (GCSE), September 2001, Erfurt, Germany.

[10] D.L. Parnas, "On the Design and Development of Program Families", *IEEE Transactions on Software Engineering*, March 1976.

[11] M. VanHilst and D. Notkin, "Using C++ Templates to Implement Role-Based Designs", *JSSST International Symposium on Object Technologies for Advanced Software*, Springer-Verlag, 1996, 22-37.

[12] D.M. Weiss and C.T.R. Lai, *Software Product-Line Engineering*, Addison-Wesley, 1999.