Appendix D

Architecture of the Software Prototype

We have constructed a prototype software tool, DPT (Design Pattern Tool), that implements seven of the design pattern transformations that have been discussed in this thesis. In section D.1 we describe the architecture of this prototype, while in section D.2 an example of the application of the prototype to a Java program is presented.

D.1 Tool Architecture

DPT has a 4-tier architecture (see figure D.1) that matches the layers defined in the structure of the behaviour preservation arguments:

1. Design Pattern transformations.
2. Minitransformations.
3. Analysis functions, helper functions and primitive refactorings.
4. AST operations.

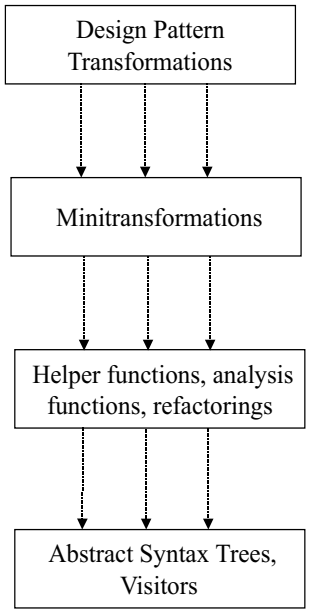

Figure D.1: Architecture of the Design Pattern Tool

The top layer implements the design pattern transformations we have discussed[1]. The next layer comprises the implementations of the six mini-transformations that emerged during the development of the design pattern transformations. The third layer is the implementation of the supporting analysis functions, helper functions and primitive refactorings described in appendix B.

The bottom layer implements the actual changes to the program code by performing surgery directly on the parse trees generated from the Java source files. Visitors [41] are frequently used at this level to perform operations that involve an entire parse tree. The parsing of the source files and the construction of the parse trees were implemented using the parser generator JavaCC [65].

[1]Seven design pattern transformations have been prototyped, namely, Abstract Factory, Factory Method, Singleton, Builder, Prototype, Bridge and Strategy.

DPT does not extract an abstract model from the Java source code. This would have made the high-level transformations such as addClass much easier, but would have made the subsequent code regeneration much more difficult. The program being transformed is stored internally as a set of parse trees, and it is the operations provided in the top three layers of the architecture that provide an abstract view of this program. A programmer building a design pattern transformation need only be concerned with the minitransformation layer, and some refactorings and helper functions, in order to complete their task.

D.2 Sample Operation of DPT

We provide an example of the application of the Factory Method transformation to a generic program:

```
class Creator {
        public void doIt() {
                Product p = new Product("some text");
                Product q = new Product(1234);
                p.foo();
                q.foo();
        }
}
class Product {
        public Product(int x){...}
        public Product(String s){...}
        public void foo() {...}
}
```

The Factory Method transformation (section 4.4) is now applied to the above program as follows:

```
applyFactoryMethod("Creator", "Product", "absProduct",
                   "absCreator", "createProduct")
```

DPT applies the transformation and outputs the following code:

```
abstract class absCreator {
    public void doIt (){
        absProduct p = createProduct("some text");
        absProduct q = createProduct(1234);
        p.foo();
        q.foo();
    }
    public abstract absProduct createProduct (int x);
    public abstract absProduct createProduct (String s);
}
class Creator extends absCreator {
    public absProduct createProduct (int x) {
        return new Product(x);
    }
    public absProduct createProduct (String s) {
        return new Product(s);
    }
}
interface absProduct {
    public void foo ();
}
```

```
class Product implements absProduct {
        public absProduct (int x) {...}
        public absProduct (String s) {...}
        public void foo() {...}
}
```

Note how in the absCreator and Creator classes, all references to Product have been changed to absProduct and instantiations of the Product class only occur via invocations of the new construction methods, createProduct. The only change to the Product class is that it now implements the new interface absProduct, which describes the complete interface to the Product class. The significance of these changes is that it is now easy to build a Creator class that works with a new type of Product. This can be achieved in two steps as follows:

1. Add an implements link from the new Product class that is being added to the absProduct interface;

2. Subclass the absCreator class, overriding the createProduct methods to instantiate the new type of Product class.

The new subclass of absCreator created in the second step will provide the required functionality. No further changes are necessary.