Tool support for implementation of object-oriented class relationships and patterns

Paterson, James H.; Haddow, John

Published in:
Italics

Publication date:
2007

Document Version
Early version, also known as pre-print

Link to publication in ResearchOnline

Citation for published version (Harvard):
Paterson, JH & Haddow, J 2007, 'Tool support for implementation of object-oriented class relationships and patterns', *Italics*, vol. 6, no. 4, pp. 108-124. <http://www.tandfonline.com/doi/full/10.11120/ital.2007.06040108>

Tool Support for Implementation of Object-Oriented Class Relationships and Patterns

James H. Paterson, Glasgow Caledonian University
John Haddow, University of Paisley

Address for correspondence:
Dr J. Paterson
School of Engineering and Computing
Glasgow Caledonian University
Cowcaddens Road
Glasgow G4 0BA
James.Paterson@gcal.ac.uk

Abstract: This paper reports the use of the PatternCoder tool in the teaching of object-oriented design and programming. This tool has been developed by the authors as an extension to the BlueJ Java Integrated Development Environment (IDE).

PatternCoder encapsulates knowledge of design patterns and basic class relationships, and of the techniques required for their implementation in Java. It guides students through a step-by-step process: select an appropriate pattern or relationship; give the classes which participate in the pattern names relevant to the current problem domain; and generate code for minimal Java class definitions which can then be explored and extended.
The tool was initially developed with a view to teaching advanced design patterns, but we have explored its use within introductory classes, viewing binary class relationships as simple design patterns. Initial experience with the tool within an introductory Java module has been positive, with students actively choosing to use the tool and feeling that the teaching approach based on its use was beneficial to their understanding of class relationships.

Keywords: Design; Object-orientation; Code-generation; Java; Patterns

1. Introduction

Many textbooks recommend an objects-first approach to teaching introductory programming using an object-oriented language such as Java (e.g. Barnes & Kölling, 2006). This approach introduces the concepts of classes, objects and object interactions from the beginning of the course. Objects-first often involves integration of programming and object-oriented design, either through explicitly teaching UML modelling (Alphonce & Ventura, 2002) or through a simplified representation of class diagrams such as that provided within the BlueJ integrated development environment (IDE) (Kölling *et al.,* 2003).

In the course of teaching object-oriented software development within a three-year undergraduate pass-degree programme we have moved towards an objects-first approach. Design and programming are taught within distinct modules, with an integrated assessment. We have observed that the process of moving from a UML model to a working implementation in Java presents a major obstacle to many students. This paper describes the general approaches we have taken to supporting the transition from design to implementation, and describes a software tool which we have developed for the purpose of assisting students specifically with the detailed implementation of relationships defined within a UML class diagram.

1.1 Common design faults

The difficulties which students experience with this transition appear to occur for a number of reasons. Where the design has also been produced by the student, then limitations or faults in the design can often make the implementation difficult. The issues that we have observed reflect many of those identified by other authors (Thomasson *et al.* (2006),Sanders & Thomas (2007)). For example, students may not identify an appropriate set of classes based on the requirements of the problem scenario. Thomasson *et al.* reported a wide variation in their study in the number of classes identified in response to a written scenario.

Where appropriate classes are identified, it is common for the relationships within the class diagram to be either not identified or inadequately specified. Thomasson *et al.* reported that up to 97% of designs in their evaluation exhibited what they call ‘non-referenced class faults’. Where classes are referenced, we have observed a student will often simply draw an association line between two classes without considering the multiplicity and directionality of the association; both of which significantly influence the way the relationship will be implemented. There are a number of other misconceptions which can give rise to design faults, such as references to what Thomasson *et al.* call ‘non-existent classes’ and ‘attribute misrepresentations’. Such faults do not necessarily prevent the design from being implemented successfully, but the result is ‘not object-oriented’. A particular example which we see frequently may arise from previous exposure to databases and the relational model. For example, given an aggregation relationship between classes representing

Order and *OrderLine* objects, the student would include in the implementation of *OrderLine* a string or numerical field to contain a value matching the value of an *id* field in *Order*, so that relationships are defined by values rather than references.

We have focused attention in our teaching on the specific issue of the implementation of relationships between classes as defined by object references in instance variables and method and constructor parameters. By placing emphasis on the way in which relationships can be expressed within a class diagram and at the same time on how these relationships are in turn implemented in code, we hypothesise the following benefits:

- Students will be better able to determine what relationship is likely to be appropriate in a given situation;
- They will then be able to implement the relationships confidently in code.

This focus does of course address only one of the many areas of difficulty encountered in learning to design and program object-oriented systems. Sanders & Thomas (2007) summarise a range of problems identified by a number of authors, and in fact relationship, or 'linkage' problems were not the most significant observed in the particular assignments they report. However, from our experience we believe that a strong understanding of object relationships is a fundamental requirement for successful object-oriented design and programming.

1.2 Introducing design patterns

We have encouraged students at a more advanced level to explore design patterns and apply them in their projects. Design patterns are recurring solutions to common problems, and describe best practices, good designs, and capture experience in such a way that it is possible for others to reuse this experience. The seminal work by Gamma *et al.* (1995) described a wide range of patterns: these are often referred to as the Gang of Four, or GoF, patterns. More recent pattern catalogues are also available with examples which are specific to Java, for example Grand (2002). A design pattern defines a set of classes which work together through specific relationships and collaborations. In fact, we see the fundamental class relationships essentially as simple patterns: the 'common problems' are simply the problems of modelling the ways in which two types of real-world objects can be related. These relationships then form the building blocks of the more complex patterns such as the GoF patterns. The student needs to have a strong understanding of the class relationships and to be able to implement these correctly – if the static relationships are not correct then the dynamic collaborations will not work. Thus the same process of moving from design to implementation is a key obstacle also at this level.

1.3 Supporting implementation of class relationships

The approaches which we have taken to supporting the transition from design to implementation are broadly similar at both introductory and advanced levels, relying on the content of the teaching and on the support of a software tool developed specifically for this purpose.

Specific teaching input is given on the types of relationship which can be represented in a UML class diagram with detailed examples of the way these should be implemented in Java. For example, it is emphasised that if the multiplicity at one end of an association is 1 then the class at the other end needs a field to contain a reference to the associated object, while if the multiplicity is >1, then the referring field needs to be a collection type. The importance of directionality is emphasised: we show that a bidirectional association requires referring fields in both classes, and ask students to think about the circumstances in which such a relationship is required.

This knowledge of relationships and their implementation is encapsulated within the tool, PatternCoder, which we have developed to support teaching of design patterns and binary class relationships. An initial version of the tool and its application to design patterns has been described previously by the authors (Paterson & Haddow, 2006). PatternCoder provides a wizard-driven interface which guides the student through the steps involved in implementing a set of related classes, and generates basic code for classes which will compile ‘out-of-the-box’ and which can then be explored and modified to meet detailed requirements. It is customisable, allowing instructors to tailor the patterns offered to the student to suit the content of their course.

The following section of this paper describes PatternCoder and its use in detail, while the final section discusses the experience of using the approaches described here with students.

2. The PatternCoder tool

PatternCoder is implemented as an extension to the popular BlueJ IDE (Kölling *et al.,* 2003), taking advantage of the extension API that allows third parties to develop additional capabilities which integrate with the BlueJ environment. We have used BlueJ for several years within our introductory object-oriented programming course. BlueJ provides a development environment in which it is simple to directly instantiate objects which appear within an area of the interface called the Object Bench, and to directly examine objects in the Object Bench. It is also possible to explore the dynamic aspects of a system by directly calling methods on instantiated objects and examining the values or objects returned as a result of method calls.

We chose to develop the tool as a BlueJ extension for two reasons: firstly, because BlueJ is our IDE of choice for introductory programming; and secondly because the Object Bench allows the auto-generated code to be explored dynamically, allowing the student to develop an understanding of how the collaboration works before modifying it to implement the behaviour which is required for their lab exercise or project. A key strength of BlueJ lies in the simplicity of its interface. Unlike some IDEs, in our experience students can become comfortable very quickly with the environment. We wanted PatternCoder to share this strength, so the design is very straightforward, with help given throughout. Unlike tools such as QuickUML (Alphonce & Ventura, 2003) and Green (Alphonce & Martin, 2006), it is a coding tool only rather than a design-and-coding tool. The motivation was to give students support with code implementation of class relationships and patterns within their familiar coding environment.

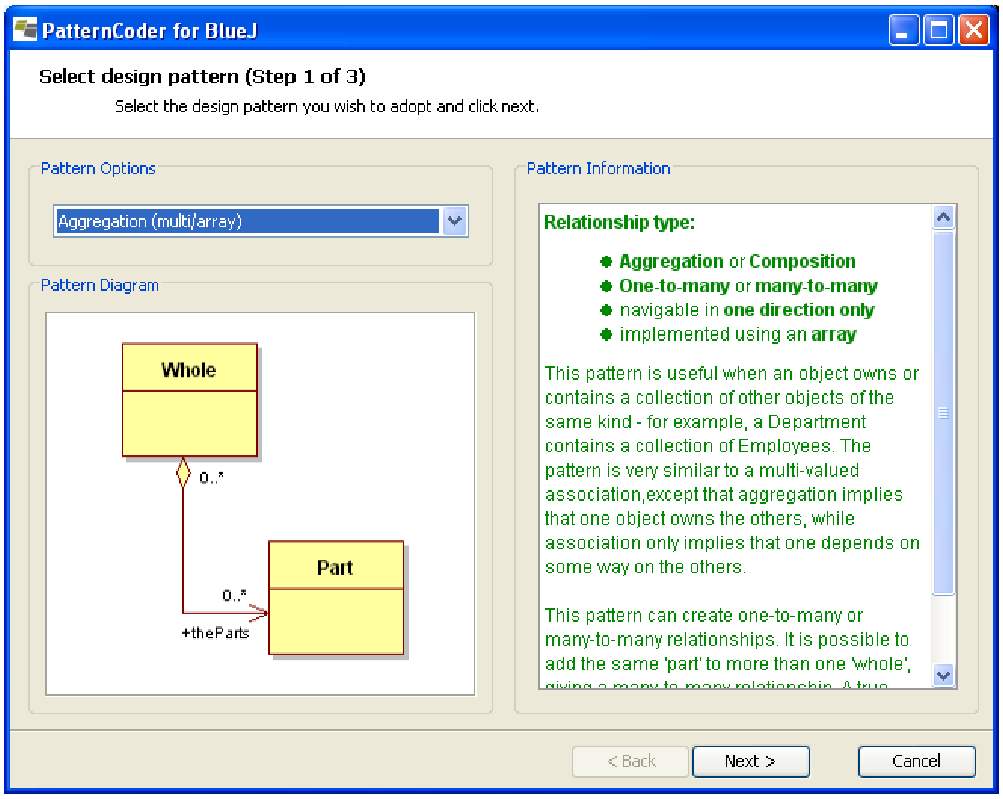
2.1 Working with PatternCoder

Installing PatternCoder adds a new menu option which is available to the user when working with a BlueJ project. Selecting the PatternCoder option presents the user with a wizard interface which includes the steps listed below.

- The user is given a list of patterns (or relationships) to choose from
- On selecting a pattern, a description of the purpose and typical applications of the pattern is displayed, along with a class diagram
- The user is then given the option to rename each class involved in the pattern to match their domain – each class will have a default name which can be replaced
- The specified classes will be created and added to the project, with the appropriate associations between them

The initial step is shown in figure 1. Note that prompts and information about the pattern or relationship are provided at each step.

Figure 1. The PatternCoder dialog (initial pattern selection step)



PatternCoder for BlueJ

Select design pattern (Step 1 of 3)

Select the design pattern you wish to adopt and click next.

Pattern Options: Aggregation (multi/array)

Pattern Information

Relationship type:

- **Aggregation** or **Composition**
- **One-to-many** or **many-to-many**
- navigable in **one direction only**
- implemented using an **array**

This pattern is useful when an object owns or contains a collection of other objects of the same kind - for example, a Department contains a collection of Employees. The pattern is very similar to a multi-valued association,except that aggregation implies that one object owns the others, while association only implies that one depends on some way on the others.

This pattern can create one-to-many or many-to-many relationships. It is possible to add the same 'part' to more than one 'whole', giving a many-to-many relationship. A true

The wizard based approach is similar to that used by some UML design tools, but there is an emphasis here on providing help and relevant information, and of course code is generated at the end of the process.

2.1 Pattern files and class templates

Our intention with this tool was to allow instructors to control the options and information given to students. We wanted instructors to be easily able to add new patterns, and to change, for example, the wording of hints provided to the student, or the default methods provided in the generated classes. To this end, the steps in the wizard are defined by the content of XML documents known as *pattern files*, while the generated Java code is based on text files known as *class templates*. The pattern files have two functions:

1. To provide information about the pattern and define each of the components that must be created as part of that pattern;
2. To define the structure and sequence of the wizard process.

It is possible to edit files easily in any text editor, or in an xml authoring tool in the case of the pattern files. The tool is distributed with a basic starter set of patterns based on those described by

Grand (2002). However, in this paper we will illustrate the use of patterns written to support some basic binary relationships. These are distributed as a separate download, and can be easily installed by copying files into the appropriate folders.

2.2 XML Pattern files

PatternCoder loads each of the installed pattern files when it is selected in the first stage of the wizard. The files are parsed and the data extracted from the files is used to display information about the currently selected pattern. Information about the components and wizard steps is also extracted and used to alter the behaviour of the wizard.

Listing 1 shows a pattern file for a basic relationship ‘pattern’ – this example is a unidirectional multi-valued aggregation, where the relationship between the two participating classes will be expressed in code by one class having an attribute which is an array of instances of the other class.

Listing 1. XML pattern file for an aggregation ‘pattern’

```
<?xml version="1.0"?>
<pattern patternName="Aggregation (multi/array)"
           patternImage="AggregationMultiArray.bmp">
<!—- Description of pattern -->

     <patternDescription><![CDATA[<strong>Relationship
                type:</strong>
           <ul>
                <li><strong>Aggregation</strong> or
                     <strong>Composition</strong></li>
                <li><strong>One-to-many</strong> or <strong>many-to-
                     many</strong></li>
                <li>navigable in <strong>one direction
                     only</strong></li>
                <li>implemented using an <strong>array</strong></li>
           </ul>
           This pattern is useful when an object owns or contains  a
           collection of other objects of the same kind ...

(remainder of descriptive help text omitted here for brevity)

      ]]></patternDescription>
```

```
<!—- Descriptions and dependencies of participating classes -->

     <class classId="1" compType="Whole" defaultName="Whole"
                classTemplate="AggregationMultiArray/Whole.TMPL">
           <classDescription><![CDATA[ This class defines an object
                which contains other objects. In the example
                mentioned in the previous screen, this would be the
                Department class.
           ]]></classDescription>
           <dependantClass value="2"/>
     </class>
     <class classId="2" compType="Part" defaultName="Part"
                classTemplate="AggregationMultiArray/Part.TMPL">
           <classDescription><![CDATA[ This class defines an object
                which can be contained by another object. In the
                example mentioned in the opening screen, this would
                be the Employee class.
           ]]></classDescription>
           <dependantClass value="1"/>
     </class>
<!—Definition of the wizard steps (one per participating class) -->

     <wizard>
           <step stepId="1" type="class" compId="1" nextStepId="2"
                previousStepId="" stepName="Rename Whole"
                stepDesc="Use the text field to enter a relevant
                name for the Whole class.">
           </step>
           <step stepId="2" type="class" compId="2" nextStepId=""
                previousStepId="1" stepName="Rename Part"
                stepDesc="Use the text field to enter a relevant
                name for the Part class.">
           </step>
     </wizard>
</pattern>
```

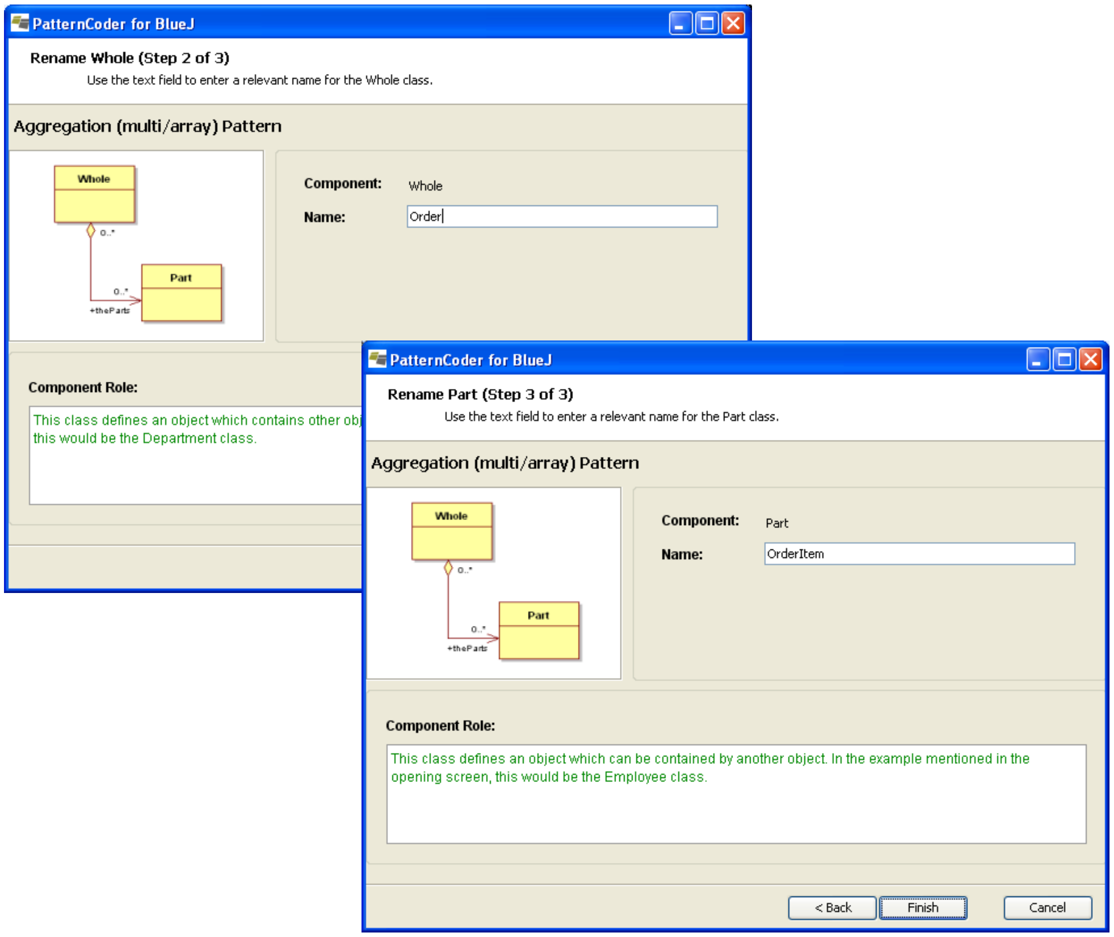
The <*pattern*> XML tag is the outermost element of any source file, and all information about the pattern is contained within this element. The attributes of this element specify the name and the location of an image which is displayed in the opening wizard screen to illustrate the selected pattern, as shown in figure 1. The image is a bitmap, intended to be a visual reference for the user. The image in the figure was created using a UML design tool, but we have also used BlueJ class diagram screenshots for some patterns. The <*patterndescription*> element is used by the extension to show HTML formatted help text describing the nature of the pattern in the initial selection screen of the wizard, as shown in figure 1.

Each participating class is represented by a <*class*> element which specifies the default name (*Whole* and *Part* are the defaults in this example) and the help text associated with the class. Each <*class*> has a unique identifier *classId*. The dependencies on other classes are also defined within the <class> element. In this example, each class is dependent on the other, but the dependencies can be more complex in more advanced patterns. The dependency information is used to ensure that object references to other participating classes have the correct names – you will see how this works when we look at a class template.

The final section of the file defines the steps in the wizard interface. Usually there will be one step for each participating class, and the purpose of the step is to allow the user to change the default name to one which is relevant to a specific scenario. The class help text is shown in the appropriate step. As with the component definitions, each step must be assigned a unique identifier, *stepId*. In the case of the wizard, this is used to uniquely identify each step and link all the steps together to produce an order in which to display them. The attributes *nextStepId* and *previousStepId* correspond to the unique identifiers of other steps defined in the wizard element. If no step is in front or behind, then attributes are left blank, indicating that it is either the start or end of the wizard process.

Figure 2 shows the wizard steps for this example – in this figure, the problem scenario is an ecommerce order system, and the user is changing the default name of the *Whole* class to *Order* and that of the *Part* class to *OrderItem*.

Figure 2. Changing the default class names



PatternCoder for BlueJ

Rename Whole (Step 2 of 3)

Use the text field to enter a relevant name for the Whole class.

Aggregation (multi/array) Pattern

Component: Whole

Name: Order

Component Role:

This class defines an object which contains other obj
this would be the Department class.

PatternCoder for BlueJ

Rename Part (Step 3 of 3)

Use the text field to enter a relevant name for the Part class.

Aggregation (multi/array) Pattern

Component: Part

Name: OrderItem

Component Role:

This class defines an object which can be contained by another object. In the example mentioned in the opening screen, this would be the Employee class.

2.3 Class templates

At the end of the wizard process, the class names entered by the user are used to write new class definitions into the currently active BlueJ project. The class definitions are based on template files. Like the pattern files, class templates can be modified to suit the instructor's preference for the generated code, and new class templates can be created for new patterns. A separate template is required for each <*class*> element in the associated pattern file.

A class template contains the code for the class to be created, with placeholders for the name of the class itself and for any other classes on which it depends. These placeholders are replaced with the names input by the user. Listing 2 shows the class template, *Whole.TMPL*, for the *Whole* class in the aggregation 'pattern'. Listing 3 shows an excerpt from the actual Java class file, *Order.java*, generated if the user enters *Order* and *OrderItem* for the class names.

Listing 2. Class template file for the class playing the ‘Whole’ role in the aggregation

```
$PKGLINE
/**
 * $CLASSNAME.java
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class $CLASSNAME
{
// array capacity - modify to a suitable value if necessary
      private final int ARRAYSIZE = 20;

// $DEPENDANT2s held in an array
      private $DEPENDANT2[] the$DEPENDANT2s;
// the number of $DEPENDANT2s in the array
      private int numberOf$DEPENDANT2s;

// example property - NOT used to define relationship
      private String description;
    /**
      * Constructor for objects of type $CLASSNAME
      *
      * @param description   value for the description
      * example property
      */
      public $CLASSNAME(String description)
      {
            the$DEPENDANT2s = new $DEPENDANT2[ARRAYSIZE];
            numberOf$DEPENDANT2s = 0;
            this.description = description;
      }
    /**
      * an example method which simply prints some information about
      * the aggregated object
      */
      public void printDetails()
      {
            System.out.println("This $CLASSNAME object (" +
                  this.toString() + ") contains the following
                  $DEPENDANT2 objects:");
            for(int i=0;i<numberOf$DEPENDANT2s;i++)
            {
                  System.out.println(the$DEPENDANT2s[i].toString());
            }
      }
     /**
      * adds a new $DEPENDANT2 object to the array of
      * contained objects
      *
      * @param new$DEPENDANT2  the new $DEPENDANT2 to add
      */
      public void add$DEPENDANT2($DEPENDANT2 new$DEPENDANT2)
      {
            if(numberOf$DEPENDANT2s < ARRAYSIZE-1)
            {
                  the$DEPENDANT2s[numberOf$DEPENDANT2s] =
                        new$DEPENDANT2;
                  numberOf$DEPENDANT2s++;
            }
      }
     /**
      * gets the array of contained $DEPENDANT2 objects
      *
      * @return    the array of contained $DEPENDANT2 objects
      */
      public $DEPENDANT2[] get$DEPENDANT2s()
      {
            return the$DEPENDANT2s;
      }
```
...template also includes methods to get a specified object and remove a specified object – omitted here for brevity

```
}
```

Listing 3. Excerpt from a Java class generated by the template shown in Listing 2.

```
    /**
      * adds a new OrderItem object to the array of
      * contained objects
      *
      * @param newOrderItem   the new OrderItem to add
      */
      public void addOrderItem(OrderItem newOrderItem)
      {
            if(numberOfOrderItems < ARRAYSIZE-1)
            {
                  theOrderItems[numberOfOrderItems] =
                        newOrderItem;
                  numberOfOrderItems ++;
            }
      }
```

2.4 Pedagogical considerations

The templates used for basic relationships typically contain several complete methods. The template shown in Listing 2 includes methods for adding to, searching in and deleting from an array of related objects. Similar templates have also been created which use Java Collections rather than arrays for multi-valued references, with equivalent methods for performing the same actions on a *List*. There are templates available to implement unidirectional and bidirectional variations relationships in cases where both are possible. Templates for bidirectional cases include code to maintain the integrity of the relationships. As a result, generated classes may have a significant amount of code which the student will *not* have to modify as their functionality is based on the mechanics of the type of relationship and is independent of the nature of the problem scenario.

So, are we doing too much of the student's work, and hiding too much of the implementation? We argue that this is not the case. The requirements specified for the design of these templates is that the generated classes should:

- Compile 'out-of-the-box' with no further modification;
- Fully implement the appropriate relationship by generating the 'boilerplate code': code which can be reused in different contexts without being significantly modified from the original;
- Contain Javadoc comments and other comments to explain the code;
- Provide a starting point for implementing problem-specific behaviour.

Together these requirements ensure that the student can immediately explore a fully-working implementation of the relationship through instantiating objects in the BlueJ Object Bench and calling their methods. The follow-on task is to implement behaviour specific to the scenario in the lab exercise or coursework being attempted. For example, the class generated by the template in Listing 2 will have a simple string property called *description*, and a method *printDetails* which simply prints some information about the aggregate to the console. Other properties and methods will need to be implemented to produce a suitable solution to the problem. The boilerplate code is created by the tool, but the student still has to devise the interesting code which provides the real behaviour through collaborations between objects.

The benefit of using PatternCoder is not simply that the student saves time by not having to write the boilerplate code. Of course, we want students to understand this code, and to be able to write it when they need to. The generated code is not hidden and can be viewed and modified freely in the BlueJ editor. The tool allows the student to generate an essentially unlimited number of code examples which assist with learning how the relationship works, and to generate and study at different scenarios and look at what parts of the solution change when the class names change. For many students this will be more useful than looking at a single example in a textbook or lecture notes: the student has ownership of the code and can generate and test examples directly within the development environment without the need to search externally for the relevant reference material.

3. Experience with PatternCoder

The initial release of the PatternCoder tool became available for use in teaching during the past academic year. This section describes the initial experiences of using it with students on introductory and advanced object-oriented programming modules.

3.1 Background

The introductory level programming class had already completed most of the module when they were introduced to PatternCoder. The module had a coursework component which involved producing a UML design model from a set of requirements, and implementing a the design in Java. The students at this stage were familiar with UML notation, through a separate module, and with the use of BlueJ.

The design task had already been completed, with feedback given to aid correction of errors. The students were in the process of implementing code solutions to the coursework problem and it was evident that uncorrected faults in the class diagrams would make implementations difficult. With the feedback given, it was hoped that implementation difficulties would be reduced. However, previous experience suggests that students have problems here even if the design is reasonably correct.

3.2 Initial Experience

We introduced at this stage a reinforcement exercise based on a scenario which had been introduced in the UML module. The scenario was designed to include a range of relationships, including one-to-one and one-to-many associations and aggregations, and also generalization. The *Order-OrderItem* relationship used as an example in part 2 of this paper was part of the scenario. The relationships and their directions and multiplicities were discussed in detail. This was followed by lab exercises in which PatternCoder was introduced to the students and they used it to help implement and explore each binary class relationship. The students were encouraged to then apply what they had learned from this exercise and use the tool in their coursework task.
Feedback from the students was very positive. Several students commented that they were now beginning to make sense of the meaning of the relationships defined in UML class diagrams and of how the relationships translate to code. It was encouraging to note from dialogue with students that many had taken the positive step of downloading PatternCoder and installing and using it on their home computers. This may be taken to indicate that the tool is straightforward to install, but more importantly that the students were convinced of the value of the tool. When asked, they confirmed that PatternCoder was a useful tool for investigating class relationships and many said they wished it had been available earlier in the course.
We now want to evaluate whether the tool and this approach had an effect on the performance of the coursework assignment. We are currently analysing the code in the work submitted by these students. In the next session, we will introduce the tool and use it to focus on specific relationships at an earlier stage. We will be looking at the initial designs produced in the coursework to investigate whether a firmer understanding of the relationship types can lead to fewer faults in the initial class designs.

3.3 Experience with Advanced Students

Based on positive experiences with introductory programming class, it was decided to make PatternCoder available to students on the advanced programming module. In fact, although this was nominally an advanced module, students had taken a variety of entry routes to this stage of the course, and many perceived themselves to be weak programmers.
The exercise described in section 3.2 was used here as a revision exercise, which proved useful as it was clear that some students either did not fully recall or had not understood the concepts well when taught initially in the previous year. These students were working on projects where they were required to establish the requirements for their own systems.

After the revision exercise, these students reported that they were able to make more sense of class relationships and how these translated their own various project scenarios. They were then observed to be using the tool and applying the concepts in their projects. Some students went on to explore design patterns, and one made use of GoF patterns in his project.

4. Conclusions

Initial experience suggests that PatternCoder can play a useful part in object-oriented programming courses at various levels. Student reaction has been positive, although it is too early to draw any firm conclusions on any improvements in student performance. Further evaluation will take place over the next academic session. A key approach to this evaluation will be to compare the incidence of common design and implementation faults in previously submitted student projects with that in the work of students who have been exposed to the approach described here.
Further developments are planned to improve the usability of the tool and to expand and improve the relationships and patterns modelled in the pattern files. It is hoped that, in addition to our efforts, new pattern definitions will be contributed to the community via the PatternCoder website, *www.patterncoder.org*.

5. Acknowledgements

This work has in part been supported by the Research Grants Fund of Bell College (now merged with the University of Paisley). The authors are grateful to Michael Nairn for his valuable contribution to the development of the software.

6. References

Alphonce, K and Ventura, P. (2002*) Object orientation in CS1-CS2 by design,* Proceedings of the 7th annual conference on Innovation and Technology in Computer Science Education, 70-74.

Alphonce, K and Ventura, P. (2003*) QuickUML: A Tool to Support Iterative Design and Code Development,* Proceedings of OOPSLA '03, 80-81

Alphonce, K and Martin, B.. (2005*) Green: A Customizable UML Class Diagram PlugIn for Eclipse,* Proceedings of OOPSLA '05, 108-109

Barnes, D.J. and Kölling, M. (2006) *Objects First with Java, A Practical Introduction using BlueJ*, 3rd Edition, Prentice Hall/Pearson Education, Harlow, England.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995) *Design Patterns:Elements of Reusable Object-oriented Software*, Addison-Wesley, Boston MA.

Grand, M. (2002), *Patterns in Java Volume 1. A Catalogue of Reusable Design Patterns Illustrated with UML*, Wiley, New York NY.

Kölling, M., Quig, B., Patterson, A. and Rosenberg, J. (2003) *The BlueJ system and its pedagogy*, Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology, Vol 13, No 4, 249-268.

Paterson, J.H. and Haddow, J. (2006) A Design Patterns Extension for the BlueJ IDE, Proceedings of the 11th annual SIGCSE conference on Innovation and technology in Computer Science Education, 280-284.

Sanders, K. and Thomas, L. (2007) *Checklists for Grading Object-Oriented CS1 Programs: concepts and misconceptions,* Proceedings of the 12th annual conference on Innovation and Technology in Computer Science Education

Thomasson,B., Ratcliffe, M. and Thomas,L.(2006) *Identifying novice difficulties in object oriented design***,** Proceedings of the 11th annual SIGCSE conference on Innovation and technology in Computer Science Education, 28-32.