PatternCoder: a programming support tool for learning binary class associations and design patterns

Paterson, James; Cheng, Ka Fai; Haddow, John

Published in:
ACM Transactions on Computing Education (TOCE)

DOI:
10.1145/1594399.1594401

Publication date:
2009

Document Version
Author accepted manuscript

PatternCoder: A Programming Support Tool for Learning Binary Class Associations and Design Patterns

J.H. PATERSON
Glasgow Caledonian University, Glasgow, UK
J. HADDOW
University of Strathclyde, Glasgow, UK
AND
K. CHENG
Glasgow Caledonian University, Glasgow, UK

PatternCoder is a software tool to aid student understanding of class associations. It has a wizard-based interface which allows students to select an appropriate binary class association or design pattern for a given problem. Java code is then generated which allows students to explore the way in which the class associations are implemented in a programming language. This paper describes the rationale behind the tool, gives a description of the tool itself, and reports on our experiences of using the tool in our teaching.

Categories and Subject Descriptors: K3.2 **[Computers and Education**] – Computer and Information Science Education – Computer Science Education.

General Terms: Design, Langauges

Additional Key Words and Phrases: Java, UML, patterns, associations

---

1. INTRODUCTION

The transition from design to code often represents an obstacle which is difficult for students studying object-oriented design and programming to surmount. They are often taught to follow a process which includes identifying a set of classes to represent the entities in a scenario, and constructing a diagram, typically a UML class diagram, to describe these classes. The diagram will also represent the associations between the classes. UML notation can describe a number of different association types, including aggregation, generalization, and so on. The class diagram can then be a starting point for implementing the classes in a programming language such as Java to create a working solution for the original scenario.

This process presents difficulties at a number of stages. The first area of difficulty is in the design. For example, Thomasson et al. [2006] identified a range of common design

---

faults which occurred in novice designs, including a wide variation in the number of classes identified in response to a written scenario. The second area of difficulty lies in translating a completed design into code. Clearly it is unlikely that a successful implementation will result from a flawed design. However, we have observed that the implementation stage is extremely difficult for many students even when the design is valid or has been improved on the basis of feedback given.

There may be many reasons for such difficulties. However, in developing PatternCoder, we have focused on difficulties arising from a lack of a clear understanding of how classes are associated, and how instances of those classes communicate in order to perform the operations required of the system. In the study of Thomasson et al. a large percentage of the student work exhibited what the authors describe as "non-referenced class faults", where students understood the need for a class to represent a concept but were unable to relate this to other classes. Sanders et al. [2008] studied student understanding of object-oriented concepts. A striking observation in their results is a complete lack of recognition of the concept of message passing, and they comment that the topic of object interaction is poorly covered in many textbooks.

We suggest that placing an emphasis on teaching the specific ways in which classes and objects *can* relate and communicate with each other helps students with a key aspect of their designs as well as with implementing those designs. This has been done by having students explore a simple system designed to showcase a range of association types. The UML representation and the Java implementation were examined. This approach is rather similar to the model-based approach to teaching programming described by Bennedsen and Caspersen [2008] which explicitly includes a conceptual modeling perspective in which the focus is on constructs for describing concepts. The authors note that this perspective is rarely addressed in introductory programming textbooks or in the relevant literature. Their approach concentrates in particular on techniques for the transition from specification model to implementation. *Coding patterns* are introduced for the implementation of classes and of associations between classes.

At a more advanced level, we teach design patterns, based on the work of the so-called "gang of four" (GoF) [Gamma et al. 1995], and again, there is an emphasis on the way that the classes within a pattern relate and communicate. 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 modeling 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 PatternCoder tool is based on the idea of coding patterns, and provides support for learning and exploring these patterns. It goes a step beyond the teaching of specific examples by providing support to students as they work through planned lab activities and as they work independently on their own designs. It also allows them to generate their own examples to help them develop the ability to abstract the features which are common to the use of a pattern within different scenarios.

2. DESCRIPTION OF THE PATTERNCODER TOOL

PatternCoder has been developed as an extension to the BlueJ IDE[1]. We see this as a good fit as we have found that the interactive Object Bench feature of BlueJ is

[1] http://www.bluej.org

particularly useful for exploring the interactions between objects. PatternCoder has been tested with BlueJ on Windows, Linux and MacOS platforms, and current system requirements are listed on the tool website[2].

PatternCoder is essentially a code generation tool which can add a set of one or more classes to a project based on code templates. The student is initially prompted to select a pattern or association from a list, with a diagram and additional textual information given to help identify which is likely to be appropriate for the scenario which he or she is working on. The student is encouraged to think about the issues of multiplicity and navigability which will determine key implementation details. Once a decision is made, generic class names are replaced with scenario-specific names, and the classes are created within the current BlueJ project.

Knowledge of the relationships between classes is encapsulated in the code templates and in XML pattern definition files. These are text files, separate from the executable code, and can be easily customized to suit the approaches taken by different instructors. Examples of the code templates and pattern files are shown in the next section of this paper.

The files supplied with the tool are designed to produce fully working code examples which allow the pattern or association to be explored 'out of the box'. This is illustrated here using an example of a scenario which involves a simple ecommerce system which processes orders. The student has identified that each *order* will consist of a number of *items*, that these are modelled as Order and OrderLine classes and that these classes are associated in some way. On starting PatternCoder (which is done by selecting a BlueJ menu option), the student can explore the available patterns and their descriptions, settling in this case on a Whole-Part (or aggregation) association where the whole can contain multiple parts, as shown in Figure 1.

The following steps, generated by the XML pattern file, allow the generic names Whole and Part to be replaced with the specific names Order and OrderLine, as shown in Figure 2. Finally, the Order and OrderLine classes are added to the current project. These classes do not at this point contain any scenario-specific logic, but they do contain enough code to allow the classes to be compiled, instances to be created, and generic operations such as adding and removing OrderLines to and from an Order to be performed. The student can use the generated classes in the following ways:

- as code examples to explore and understand the code required to implement the relationship. The ability to generate additional code examples may be useful to students who have difficulty in looking at examples given in lectures and extracting the necessary parts of the code to apply the given solution in another scenario.
- as a starting point to develop the complete solution by adding scenario-specific attributes and logic: for example a method to calculate the total cost of an Order. The generation of what is often referred to as "boilerplate" code has the same advantage in terms of saving time as it would for a professional developer, but for the novice has the additional benefit of allowing concentration on the logic without the frustration caused by errors or incorrect implementation of the association.

[2] http://www.patterncoder.org

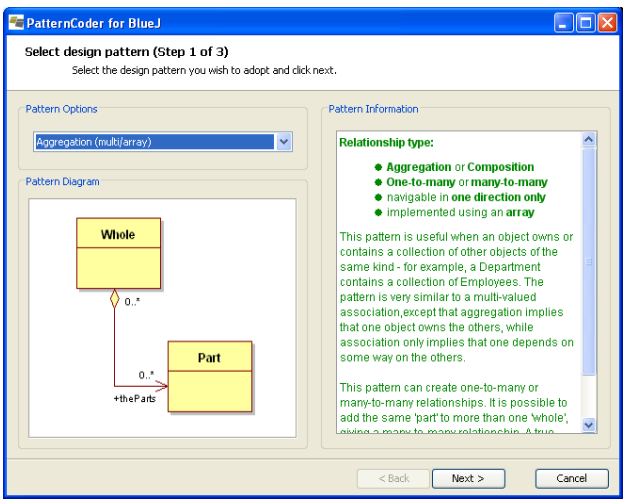

Figure 1. Selecting the appropriate pattern.

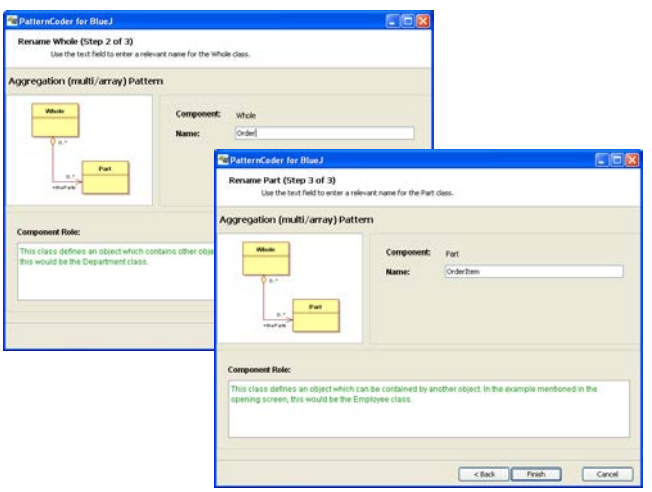

Figure 2. Naming the classes.

PatternCoder can be considered to simply be a conduit for transforming the knowledge encapsulated in the code templates and pattern files into working code. This means instructors are free to use their own knowledge and preferences to modify or expand on the choices and information presented to the students. The tool is supplied with a 'starter set' of design patterns which provides examples of several types of patterns but do not cover the whole catalog of GoF patterns. Instructors are free to add other patterns which they wish to teach. Similarly, a set of binary association patterns is included. These use our own nomenclature and descriptions of these associations, but instructors are able to modify them to suit the way they want to teach. Contributions from the community are welcomed and may be distributed through the PatternCoder website. For example, there is an ongoing project which aims to translate pattern files and templates into Portuguese.

The informational text which is displayed at each step is stored in the XML pattern definition files as HTML code, allowing instructors to create content aligned with the

skill levels of students as they progress through examples, and to highlight specific points in the text, and potentially to provide hyperlinks to more detailed tutorial material. This could, for example, support learning activities based on the idea of scaffolding and fading out in Cognitive Apprenticeship [Collins et al. 1989]. Section 5 of this paper presents an example of a set of activities which embed the use of PatternCoder as a supporting resource, and describes how the support is gradually reduced as the student progresses through the activities.

We have recently used the customizability of PatternCoder to provide an alternative set of binary association patterns [Paterson et al. 2008]. One of the difficulties in learning about these associations lies in the ambiguities caused by the gap which exists between programming languages and UML. The concept of an association does not in fact exist in Java, for example. The association must be expressed in code using the available tools: classes, attributes and methods [Génova et al. 2003]. This requires thought about exactly what is implied about the classes, and understanding of how to map that meaning to code. For example, a simple one-to-one association between two classes may be implemented using an attribute of one class. However, there may be situations where that association is temporary, and is best implemented using a reference contained in a parameter in a method call. The difference is not clearly expressed in the model, but very much affects the implementation details. These patterns, based on the work of Stevens [2002] on the semantics of binary associations, are designed to encourage students to think about this kind of issue. There are many more complex aspects of class diagrams, such as qualified associations [Akehurst et al. 2007] which could be illustrated by creating patterns in PatternCoder, although we would certainly not wish to present these to novices.

3. PATTERN DEFINITIONS

Our intention with the PatternCoder tool was to allow instructors to control the options and information given to students. We wanted instructors to be easily able to change, for example, the wording of hints provided to the student, or the default methods provided in the generated classes, or to add complete new patterns to suit their own course content and teaching approach. 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 basic binary class associations. These are distributed as a separate download, and can be easily installed by copying files into the appropriate folders.

3.1 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 behavior 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. Note that the multivalued attribute could alternatively be implemented using one of the Java Collections Framework classes, for example as an ArrayList. The most appropriate choice here would depend on the students' prior knowledge of Collections classes, and pattern files and templates based on both arrays and Collections are provided. Pattern files are validated by PatternCoder using an XML schema document which is included in the downloads of the source code and of the sample pattern files.

Listing 1. XML pattern file for an aggregation coding 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 above.

Each participating class is represented by a <class> element which specifies the default name (*Whole* and *Part* are the defaults in this particular 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 design pattern examples. It is possible for a class to be dependent on more than one other class in a pattern, which is reflected in the presence of the appropriate number of <dependentClass> elements within the <class> element. The dependency information is used to ensure that object references to other participating classes have the correct names – it will become clear 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 above illustrates the wizard steps for this example – in this figure, the user is changing the default name of the Whole class to Order and that of the Part class to OrderLine. The number of steps required in the wizard depends on the number of classes which participate in the pattern. A binary association coding pattern has two steps as shown in the example, while the number of classes in more advanced design patterns can vary. For example, the pattern file included in the PatternCoder download for the Singleton pattern has one step while the Adapter pattern file includes four steps and four classes.

3.2 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. The placeholder for the class name is identified in the template as *$CLASSNAME*, while that for a dependent class is identified as *$DEPENDENTx*, where x is the *classId* value assigned to that class in the pattern file.

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 coding pattern. Listing 3 shows an excerpt from the actual Java class file, *Order.java*, generated if the user enters "Order" and "OrderLine" for the class names. Comparing the *addOrderLine* method listed here with the *add$DEPENDENT2* template method in Listing 2 shows how the placeholder class names in the code template have been replaced with the class names entered by the user to replace the defaults defined in the pattern file.

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 OrderLine object to the
 * array of contained objects
 *
 * @param newOrderLine   the new OrderLine to add
*/
  public void addOrderLine(OrderLine
    newOrderLine)
  {
    if(numberOfOrderLines < ARRAYSIZE-1)
    {
      theOrderLines[numberOfOrderLines] =
        newOrderLine;
      numberOfOrderLines ++;
    }
  }
```

4. RELATED TOOLS AND WORK

Bennedsen and Caspersen [2008] provide a detailed description of a concrete implementation of a "model-first" course. They include examples of associations, the programming concepts which students require to understand in order to implement the associations, and coding patterns which are used for the implementation. The authors report that this approach has been used successfully with a range of students and teaching styles.

Alphonce and Ventura [2002] describe a design-driven approach to an object-oriented CS1-CS2 sequence. Early programming assignments are based on implementing a given design, while as students progress they learn to create their own conceptual models from a set of requirements. This teaching approach is supported by a software tool, Green [Alphonce and Martin 2005], a round-tripping UML editor plug-in for Eclipse. The aim of this tool is quite similar to that of PatternCoder, but the implementation is significantly different. Green is essentially a UML diagram editor which allows classes and associations to be added interactively to a project. PatternCoder's wizard-driven approach, in contrast, does not try to provide any support for diagramming. Green's knowledge of associations is encapsulated in Eclipse plug-ins, which are themselves written in Java, in contrast to the relatively easily modified templates and XML files used by PatternCoder. There is no tutorial content embedded within the tool to help students decide on the most appropriate association type. Finally, there is no specific support in Green for design patterns.

Patterns+UML [Denegri et al. 2008], like PatternCoder, is an educational tool designed to provide support for the implementation of design patterns. The authors emphasize its use for exploring situations where a class can play roles in more than one design pattern, and they correctly state in comparison that PatternCoder does not support interaction between patterns. The wizard-based process for implementing a pattern appears similar to PatternCoder, Unlike PatternCoder and Green, there is no IDE integration. It is not clear in the reference whether it is possible to customize the patterns offered to the user or whether binary associations are included.

There are many tools designed for professional developers which offer 'round-trip' code generation, in which changes made to a UML class diagram are immediately reflected in programming language code which is automatically generated. To take just one example, in the eUML plug-in for Eclipse[3] you can draw classes and an association between them, and edit the properties of the association. These properties are then reflected in the Java code generated for the classes. Similarly, many professional tools such as IBM Rational Developer[4] have much more sophisticated support for building design patterns into a model or project than PatternCoder offers. However, such tools are not suitable, or intended, for novice developers, are often complex, and do not offer tutorial content.

There is considerable research effort ongoing into the development of model-driven development tools which can automatically map UML associations, including the more complex types of association, into code [Akehurst et al. 2007; Génova et al. 2003]. This is not a trivial task, due to the conceptual gap between model and programming languages discussed in the previous section, and current professional UML tools which

[3] http://www.soyatec.com/euml2

[4] http://www.ibm.com/developerworks/rational

offer code generation support for only a limited range of associations. Génova et al. [2003] have described a prototype code generation tool for UML associations, JUMLA.

Note that unlike most of the above, PatternCoder does not attempt to be a model-driven development tool. It does not analyze a model and its associations in order to generate code. Instead, it asks the student to think about his or her model and to actively make a decision on what type of association should be implemented.

5. USING PATTERNCODER

PatternCoder has been used with students on level 2 and level 3 modules in object-oriented programming and design. Students had some previous or parallel exposure to basic procedural programming concepts and to object-oriented design. They were therefore familiar with the notion of conceptual modeling. Design and programming were taught separately, but the final programming assessments required the production of a conceptual model in the form of a UML class diagram which was then used as the basis of the programming task.

A set of lab activities based on a model-driven approach was introduced into these courses, and the impact that these had on performance on the final assessments was considered. These activities are described here to show an example of a teaching approach which uses PatternCoder as a supporting tool. It should be emphasized that PatternCoder itself does not dictate a particular approach or sequence of activities, but can be adapted to support a wide range of activities by creating appropriate pattern files.
The activities were introduced at level 2 as a key part of the curriculum, and were used as revision at level 3 as these students were observed to be having difficulties with implementation of conceptual models. Level 3 students also had lectures on design patterns and worked on lab activities using PatternCoder to support implementation of a range of well-known patterns. An example of the use of PatternCoder to implement a design pattern has been given in a previous paper [Paterson et al. 2006].

The activities were based around a conceptual model of an order processing system. The class diagram is shown in Figure 3. The model, while simple, showcases a range of association types. For example, there is a one-to-many aggregation relationship between Order and OrderLine, while OrderLine and Product have a simple association which has directionality explicitly represented. Generalization is illustrated by SpecialProduct-Product. Some cases require some thought about exactly what is implied about the classes, and understanding of how to map that meaning to code. For example, the association between Order and Invoice is rather ambiguous. The semantics of this association are discussed, and it is suggested that the association is a dynamic one, following the nomenclature of Stevens [2002], where an Order creates an Invoice and objects are only associated for as long as it takes to create the Invoice.

The sequence of activities built around the model explored each association in turn. The goal was to produce a complete system in which all classes and associations are implemented in Java. The BlueJ IDE was used for the implementation as its Object Bench feature has been found to be extremely useful for testing associations by instantiating objects and directly exploring their interactions without having to create a main program. The level of support in selecting appropriate coding patterns was faded out as the students proceeded through the exercises, as was the support provided by PatternCoder for implementation. For example, in the final task, students were asked to hand-code an implementation based on a coding pattern which they had previously implemented using PatternCoder.

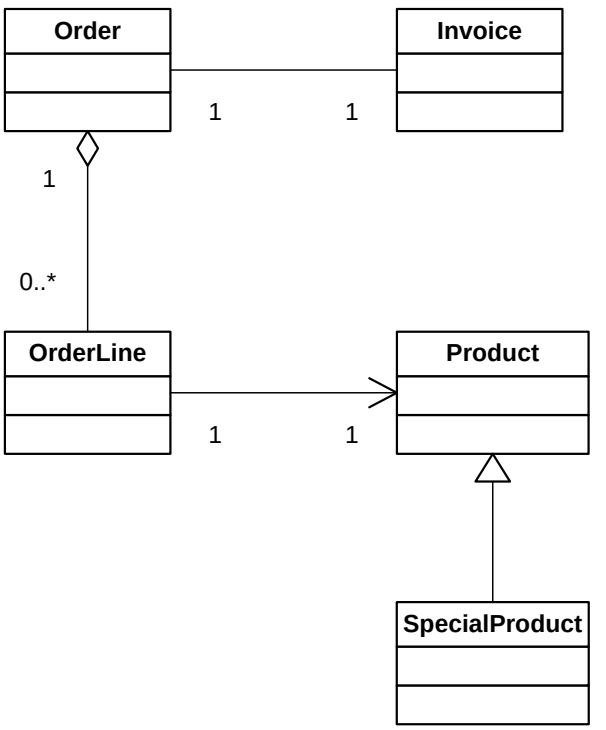

Figure 3. Class diagram for lab activities.

The progression of activities was as follows:

Task 1. Implement the association between OrderLine and Product.
Implications of multiplicities and navigability are discussed and students are guided through implementation of the association using a specified coding pattern with the aid of PatternCoder.

Task 2. Implement the association between OrderLine and Order
Implications of multiplicities and navigability are discussed and students are required to implement the association using a suggested coding pattern with the aid of PatternCoder.

Task 3. Implement the association between Order and Invoice
The semantics of this association are discussed as described earlier in this section, and students are required to implement the association using a suggested coding pattern with the aid of PatternCoder.

Task 4. Implement the association between Product and Special Product
Students are required to implement the generalization association with the aid of PatternCoder in selecting a coding pattern and creating code.

Task 5. Add a Customer class which has associations with both Order and Invoice.
Students are required to create the Customer class and implement its association with Order with the aid of PatternCoder. They are then required to modify the Customer class to implement the association with Invoice using a coding pattern which has been seen previously but without the aid of PatternCoder.

All tasks include the following additional components:

- Examining the code of the basic classes and exploring their interaction by creating and manipulating objects in the BlueJ object bench.
- Modifying the base code created using a coding pattern to implement scenario-specific behavior.

Figure 4 illustrates the exploration of the interactions in task 2. After compiling the classes generated by PatternCoder, the BlueJ object bench has been populated with an instance of Order and three instances of OrderLine. The pop-up menu shows the methods which are available for the Order object. These methods have been generated from the class template used for Order, and represent a basic set of operations appropriate for a class involved in this type of association. The *addOrderLine* method is used to add each of the OrderLine objects in turn to the Order. The result of doing so can be explored in two ways.

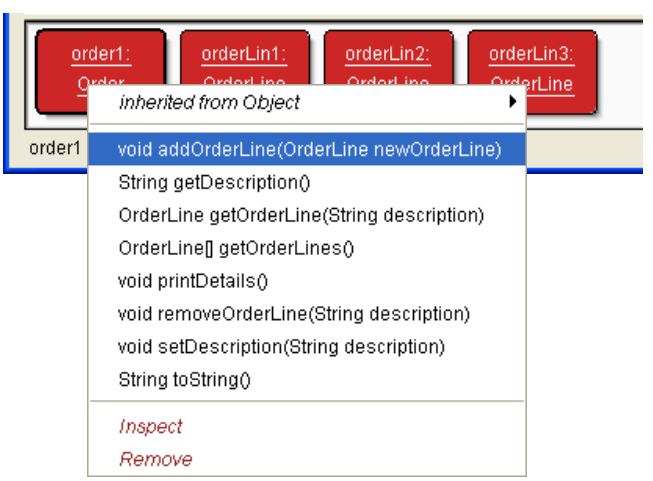

Figure 4. Exploring interactions between instances of classes generated by PatternCoder.

Firstly, BlueJ's capability to inspect objects can be used, as shown in Figure 5. Inspecting the Order object reveals that it has an instance variable called *theOrderLines*, and inspecting this variable in turn reveals an array of OrderLine objects. Inspecting those demonstrates that the array contents are references to the OrderLine objects which are shown in the object bench.

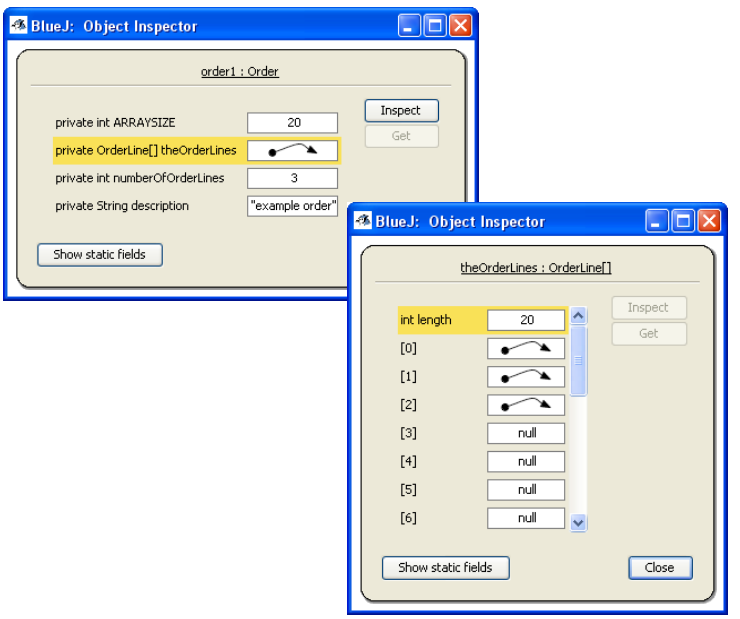

Figure 5. Inspecting instances of classes generated by PatternCoder.

Secondly, the template has generated a method *printDetails* in the Order class, as shown in Listing 4. This simply outputs some text to the console, using the *toString* method of the Order class. An example of the console output is shown in Listing 5. More interestingly, it also illustrates message passing between associated objects by calling the *toString* method of each OrderLine in turn and writing the return value to the console. The *printDetails* method is an example of the way in which a template can include content designed to illustrate and demonstrate the way in which associated objects can interact and how this interaction is expressed in code.

Listing 4. The *printDetails* method in a class generated by PatternCoder.

```
public void printDetails()
{
  System.out.println("This Order object ("
    +  this.toString() +
    ") contains the following OrderLine
    objects:");
  for(int i=0;i<numberOfOrderLines;i++)
  {
    System.out.println(theOrderLines[i].
      toString());
  }
}
```

Listing 5. Example output from the *printDetails* method in a class generated by PatternCoder.

```
This Order object (description: example order) contains the
following OrderLine objects:
description: example orderline 1
description: example orderline 2
description: example orderline 3
```

The final step in this particular task is to modify the basic code to fit the scenario. For example, students are asked to include more appropriate field names to hold information such as an order number and order date, and to add methods to allow an Order object to calculate and return its total cost.

A potential danger with using code generating tools in teaching is that the students are given too much help and do not need to understand the code which is created. Classes generated by PatternCoder 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. We suggest that code templates should be designed such 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 behavior

Together these requirements ensure that the student can immediately explore a fully-working implementation of the relationship and then implement behavior specific to the scenario. The benefit of using PatternCoder is not primarily that the student saves time by not having to write the boilerplate code. 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.

Used to support independent study, the tool allows the student to generate an essentially unlimited number of code examples which assist with learning how the relationship works, and to abstract the key features of the coding pattern. 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.

6. EVALUATION

Feedback from the students on the lab activities 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. The level 3 students found the tool helpful for revising class relationships, which many of them had struggled with previously.

Following on from these activities, a key assessment instrument at each level was an integrated project in which the students, either individually or in groups, were required to analyze a new scenario to produce a conceptual model of a solution and to implement the solution in Java.

We were interested to find out if there was any evidence of the impact of the model-based activities on the students' ability to develop their own conceptual models. The project work, based on a new scenario, submitted by the level 2 students was reviewed to identify the level of incidence of the set of common design faults reported by Thomasson et al. [2006]. It should be emphasized that this is a very small scale review of the work of 20 students who were organized into groups of 3 or 4. The fault types are described as "non-referenced classes" (NRC), "references to non-existent classes" (NEC), "single attribute misrepresentation" (SAM) and "multiple attribute misrepresentation" (MAM). Details of the nature of these faults can be found in the reference. We also identified a further fault, which we refer to as "unspecified association" (UA), where an association line is drawn in the class diagram, but there is no indication at all of multiplicity, navigability or association type. The results are shown in Table 1, together with the equivalent results obtained by Thomasson et al. The latter results were obtained from a much larger pool of students, and the figures shown here are averages from the figures given in the reference for three design exercises involving a total of approximately 450 students.

The striking feature apparent in the table is that *no* non-referenced class faults were observed in the work of our students, in contrast to the high incidence of this fault in the previous study. This suggests that these students at least have a clear understanding that a class must be associated with other classes in order to play a part in a system.

Table 1. Total number of faults and and percentage of designs containing each fault.

fault	**this work**	**Thomasson et al. 2006**
Total number of NRC	0	629
% of designs with this fault	0%	89%
Total number of NEC	4	116
% of designs with this fault	20%	31%
Total number of SAM	31	277
% of designs with this fault	60%	50%
Total number of MAM	4	42
% of designs with this fault	20%	15%
Total number of UA	36	-
% of designs with this fault	80%	-

Although most of the designs exhibited some examples of unspecified associations, the number of these was a small proportion of the total number of associations in each design, suggesting that there is also good understanding of the need to consider the nature of each association in order to implement it.

The results described here refer to design diagrams, whereas the specific role of PatternCoder is to support the coding of patterns and associations. However, part of the rationale of the tool is to develop understanding of the nature of associations and the need to implement them in code, and it is hoped that such understanding will feed back into improvements in design skills. It is not possible to deduce from these results the specific influence of the PatternCoder tool has had. The teaching emphasis placed on associations is likely to be a major factor. The fact that students were working in groups and discussing their designs may also have been significant. These results are based on a very small body of work and so it is not possible to draw strong conclusions. However, it does appear that a teaching approach emphasizing class associations, supported by the use of PatternCoder, has the potential to improve understanding in an area which has been shown previously to cause difficulty for students.

7. DISSEMINATION

The PatternCoder tool is open-source software, and has been made freely available for download since mid-2006. It was originally known simply as the Design Patterns extension for BlueJ. In summer 2007 the *www.patterncoder.org* website was created to distribute the tool and related materials. The download includes binaries and source code, javadocs and a guide to installing and using the tool. PatternCoder works with BlueJ on Windows, Linux and MacOS. The project code is hosted on Google Code[5] for ease of collaborative development. A basic set of teaching materials which we have developed and used are also available for download on the website. A number of papers and presentations have been given on the tool and teaching approaches based on it [Paterson et al. 2006; Paterson et al. 2008]. Approximately 1100 downloads were recorded over a 12 month period to June 2008. Integration with BlueJ, which is widely known and used in the CS education community, has an advantage in terms of dissemination. We are grateful to the BlueJ team for placing a link to our tool in their website, and website statistics show that this link brings a significant level of traffic to the PatternCoder site.

Pears et al. [2007] noted that very few teaching tools have seen widespread adoption within CS education, and identified some possible reasons for this, including the origins of many tools as solutions to local problems, and a lack of development and funding to make tools suitable for use across a wide range of institutions. A further reason may be a lack of readily available teaching materials. One of the most widely adopted tools is BlueJ, and the availability of a textbook [Barnes and Kölling 2008] written by the tool authors and closely based on its use may be a significant factor. We propose to develop a comprehensive set of tutorial materials which will make use of and integrate with PatternCoder.

8. LIMITATIONS AND FUTURE DEVELOPMENTS

The current early release of PatternCoder has a number of issues which warrant further development. It is currently possible only to add new classes to a project, so there is no way of including an existing class within a pattern or association and modifying its code accordingly. This makes it difficult to use PatternCoder to build up a set of associations between multiple classes or to refactor a design to make use of a pattern. The authors of

[5] http://code.google.com/p/patterncoder

the Patterns+UML tool [Denegri et al. 2008] emphasize this limitation in comparison to their tool, and we acknowledge and plan to address this.

Pattern file management could be improved to make it easier to install or uninstall patterns or to let the user navigate and select from within multiple named sets of patterns. Creation of pattern files and templates is done simply by editing text files (templates and XML pattern files). Although this process is documented, it would be helpful to have a tool which would provide a user-friendly interface to help instructors customize patterns and create new ones. Support for internationalization of the user interface has not yet been implemented, although the pattern files and templates can be easily edited and translated into other languages.

PatternCoder has been implemented as a BlueJ extension because we see BlueJ as a good fit for a teaching approach which emphasizes class associations. However, it would be relatively straightforward to create a standalone version, or one which integrates with other popular IDEs such as Eclipse or NetBeans.

9. CONCLUSION

Class associations and collaborations present difficulties for many students of object-oriented design and programming, something which has become apparent through our own experience and through studies reported in the literature. The PatternCoder tool was developed to support a teaching approach which emphasizes the nature and importance of associations between classes which collaborate within design patterns and through simple binary class associations. Initial experience suggests that this approach can produce significant benefits.

One of the problems with evaluating tools such as PatternCoder is that unless the tool is widely adopted, the body of student experience and work available for study is limited. It would be valuable to be able to draw together experiences with the tool from a larger number of institutions. Dissemination of the tool is a key target in order to promote wider adoption and potentially give access to a wider base for evaluation. The tool is being promoted to the community through papers and conference presentations, and a website has been designed to provide free access to the tool for instructors and students. Tutorial materials are also under development to lower the barrier to adoption of the tool. It is hoped that these efforts will drive future adoption, evaluation and development of PatternCoder.

10. WEBSITE

The PatternCoder website is located at *www.patterncoder.org*. The website includes:

- Brief overview and screenshots of the tool
- Downloadable guide to installation and use
- Download of binaries, source code and Javadocs
- List of publications and other resources

.

REFERENCES

AKEHURST, D., HOWELLS, G. AND MCDONALD-MAIER, K. 2007. Implementing associations: UML 2.0 to Java 5, *Journal of Software and Systems Modelling*, Vol. 6, No 1, 3–35.

ALPHONCE, C. AND MARTIN, B. 2005. Green: a customizable UML class diagram plug-in for Eclipse. In *Companion to the 20th annual SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications*, ACM Press, 168-169.

ALPHONCE, C. AND VENTURA, P. 2002. Object orientation in CS1-CS2 by design. In *Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education* (Aarhus, Denmark, June 24 - 28, 2002). ITiCSE '02. ACM, New York, NY, 70-74.

BARNES, D.J. AND KÖLLING, M. 2008. *Objects First with Java. A Practical Approach*, 4th Edition, Prentice Hall / Pearson Education.

BENNEDSEN, J. AND CASPERSEN, M 2008. Model-Driven Programming, In *Reflections on the Teaching of Programming, Lecture Notes in Computer Science* Vol. 4821, 116-129, Springer-Verlag Berlin / Heidelberg,

COLLINS, A., BROWN, J.S. AND NEWMAN, S. 1989. Cognitive Apprenticeship: teaching the craft of reading, writing and mathematics. In L. Resnick (Ed.) *Knowing, learning and instruction: essays in honor of Robert Glaser* (pp453-494). Hillsdale, NJ: Lawrence Erlbaum.

DENEGRI, E., FRONTERA, G., GAVILANES, A., AND MARTÍN, P. J. 2008. A tool for teaching interactions between design patterns. In *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education* (Madrid, Spain, June 30 - July 02, 2008). ITiCSE '08. ACM, New York, NY, 371-371.

GAMMA, E., HELM, R., JOHNSON, R. AND VLISSIDES, J. 1995. *Design Patterns: Elements of Reusable Object-oriented Software*, Addison-Wesley, Boston, MA.

GÉNOVA, G., RUIZ DEL CASTILLO, C. AND LLORENS, J. 2003. Mapping UML Associations into Java Code. *Journal of Object Technology*, Vol. 2, No. 5, 135-162.

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

PATERSON, J. H., HADDOW, J., AND NAIRN, M. 2006. A design patterns extension for the BlueJ IDE. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (Bologna, Italy, June 26 - 28, 2006). ITICSE '06. ACM, New York, NY, 280-284.

PATERSON, J. H., HADDOW, J., AND CHENG, K. 2008. Drawing the line: teaching the semantics of binary class associations. In *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education* (Madrid, Spain, June 30 - July 02, 2008). ITiCSE '08. ACM, New York, NY, 362.

PEARS, A., SEIDMAN, S., MALMI, L., MANNILA, L., ADAMS, E., BENNEDSEN, J., DEVLIN, M., AND PATERSON, J. 2007. A survey of literature on the teaching of introductory programming. In *Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education* (Dundee, Scotland, December 01 - 01, 2007). J. Carter and J. Amillo, Eds. ITiCSE-WGR '07. ACM, New York, NY, 204-223.

SANDERS, K., BOUSTED, J., ECKERDAL, A., MCCARTNEY, R., MOSTRÖM, J., THOMAS, L. AND ZANDER, C. 2008. Student understanding of object-oriented programming as expressed in concept maps.In *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, 332-336.

STEVENS, P. 2002. On the interpretation of binary associations in the Unified Modeling Language. *Software and Systems Modeling*, Vol. 1, No. 1, 68.

THOMASSON, B., RATCLIFFE, M., AND THOMAS, L. 2006. Identifying novice difficulties in object oriented design. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (Bologna, Italy, June 26 - 28, 2006). ITICSE '06. ACM, New York, NY, 28-32.