Design Patterns: Between Programming and Software Design

Christoph Denzler
UAS Northwestern Switzerland
Steinackerstrasse 5
CH 5210 Windisch / Switzerland
+41 56 462 4411
christoph.denzler@fhnw.ch

Dominik Gruntz
UAS Northwestern Switzerland
Steinackerstrasse 5
CH 5210 Windisch / Switzerland
+41 56 462 4317
dominik.gruntz@fhnw.ch

ABSTRACT
In computer science curricula the two areas programming and software engineering are usually separated. In programming students learn an object oriented language and then deepen their knowledge in other languages, algorithms and data structures. On the other hand software engineering starts with discussing processes and then addresses topics like requirements engineering, software design and software architectures.

Design patterns are on the border of these two areas and can be covered from both sides: either as an advanced programming course or as an application of software design and micro architectures. In this paper we present courses on design patterns and on software design which try to bridge this gap.

Categories and Subject Descriptors
D.2.11 [**Software Engineering**]: Software Architectures – *design patterns;* D.3.3 [**Programming Languages**]: Language Constructs and Features – *patterns;* K.3.2 [**Computers and Education**]: Computer and Information Science Education – *design patterns, curriculum.*

General Terms
Design, Experimentation.

Keywords
Design Patterns, Software Engineering Curriculum.

1. INTRODUCTION
A course on design patterns can be regarded both as the top course in the area of object-oriented programming or as the introductory course in the area of software engineering.

When our school adopted the standards of the Bologna agreement, the curriculum in computer science also had to undergo redesign. Before this reform, the design patterns course was more of a course in advanced programming. The software engineering courses did not build upon the knowledge gained in design patterns.

This is somewhat typical for software engineering education. In the first semesters the main focus lies on programming. This discipline is a precise discipline. Students work with text editors, compilers and runtime environments – tools that are not forgiving at all. On the other hand, software engineering is a soft discipline. There hardly exists "the right solution" for any given problem.

Teaching programming courses is about teaching definitions. There is no arguing about how the for loop in Java works. Teaching software engineering courses is about teaching definitions too, but it is also about training to balance requirements and solutions. That is why programming is not an engineering discipline!

In our new curriculum design patterns are lectured in the third of six semesters. At this time the students have enough programming skills to move on towards software design. We wanted the course to bridge the gap between programming and designing (which we consider to be a software engineering skill).

Student assignments play a crucial role in our approach to connect software engineering expertise with the necessary precision of programming. In the following sections we present our approach and discuss advantages and shortcomings.

2. GRAPHICS EDITOR
Deciding on the content of a design pattern course can be narrowed to the question of which patterns to present in the available time. Usually this is also the approach followed in books about design patterns [1,2], where the patterns are explained more or less independently of each other. This has the advantage that for each pattern the best fitting examples can be chosen to explain details and to discuss the pit-falls, strengths and weaknesses. The focus lies on implementation and programming skills.

In order to shift the focus from programming to design, we decided not to issue independent exercises for each pattern but to work on a system that is extended with each assignment. The system is a small graphics editor – shown in Figure 1. It is a single-document multiple view editor which supports the usual editing operations (moving, resizing, grouping, etc.).

Each assignment introduces a new feature to the editor which has to be implemented by the students using a design pattern. The editor can be extended with new figure types.

ICSE'08, May 10–18, 2008, Leipzig, Germany.

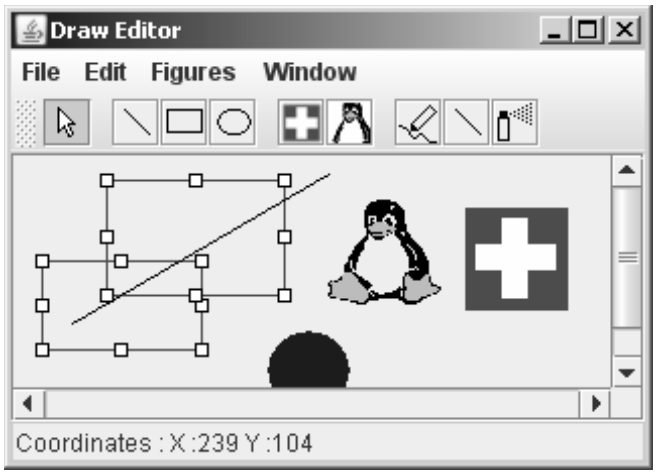

Figure 1. Editor developed during the assignments

We also looked at JHotDraw [4], another editor which is used to demonstrate the use of design patterns. However, we considered JHotDraw as too complex for our needs. Students would be overstrained with its design. Our simplified version still contains enough features to implement the most important design patterns.

The following paragraphs present the design patterns which have to be implemented in the assignments. The editor gets a new feature with each design pattern.

Observer Pattern
The editor is based on a model-view separation. The figures are stored in a model and can be edited through several windows. In order to keep the editor views consistent with the model they are registered as observers in the model. In addition, modifications of figures are notified through the model to all views as well. The model is thus registered as an observer of each figure.

State Pattern
The construction of new figure instances is controlled by a tool implementing the state pattern. The tool defines how the mouse and keyboard events are used to construct a new figure.

Strategy Pattern
The editor supports a grid feature (e.g. snap to a fixed grid or snap to the corners or building lines of existing figures) which is implemented with the strategy pattern.

Composite Pattern
Group figures are an implementation of the composite pattern.

Decorator
The decorator pattern is used to enhance the look and behavior of figures: A bundler prevents the modification of figures, a border decorator adds a rectangular frame around a figure and an animation decorator moves the figure around.

Our design allows discussing the problem of object identities in the context of the decorator pattern: The handles returned by the figures contain a method which returns the figure they belong to. The decorator implementation has to decorate the handles so that these return the decorator as their origin.

Command
The undo/redo functionality is implemented with the command pattern. The composite pattern is used to combine several single commands to a macro command.

Prototype
The prototype pattern is used to implement the cut/copy/paste functions. At least for composite figures deep copy has to be implemented.

Factory/Builder/Singleton Pattern
The Factory, Builder and Singleton patterns are creational patterns used to abstract the construction of objects. Clients do not depend on which concrete instance is created and how it is initialized. In our framework we use creational patterns to construct the cooperating objects (model, views) and the available figure types (tools). The implementation of these patterns is based on the Spring framework [5]. In its core, Spring provides a dependency injection framework which allows to compose the various objects into a working application. It can be seen as a codification of the creational design patterns.

By extending the editor's capabilities with each assignment we achieved several goals:

- knowledge consolidation on the practical use of a design pattern,
- a broader understanding of how design patterns integrate into larger systems,
- a boost of motivation to work on the assignments, as in the end their effort will be rewarded with a small application.

The feedback we get from the students confirms that we chose the right approach. Whereas at the beginning of the semester, student questions often are about simple programming techniques, only after a few weeks, discussions are focusing more on topics such as interface design or extensibility of the system.

3. PROVIDE OR BUILD A FRAMEWORK?

Although – or because – we wanted to outline the fact that design patterns are always embedded in an overall system design, we were confronted with another important decision. What parts of the system – if any at all – should be provided in the beginning? There are many aspects to consider when deciding on this problem:

- student motivation, learning curve and work load
- complexity
- system design skills
- mastery of IDEs and tools
- support and mentoring effort
- compatibility between different implementations

We were weighing three approaches:

1. Providing requirements only: this will probably kill the motivation to start as the work load will be very high in the beginning. Also the learning curve is steep as for many students it is the first large application. They have to master the IDEs support for projects, learn about GUI widgets, libraries, event handling, etc.
2. Providing a simple but running graphics editor with basic functionality. In addition, providing a list of requirements for extensions and refactorings such as the possibility to display a graphic in several windows, to add undo/redo functionality, etc. This would challenge the students to invest thoughts into the interface design and yet the work load would not be as bad as with the first approach. This approach is similar to approach presented by Stuurman and Florijn in [3].

3. Providing a GUI, and interfaces for the main entities of the editor (such as figures, events, tools, commands, views and models). If the interfaces are not altered by the students, they will be able to exchange their implementations, e.g. it will be possible to enhance a classmate's editor with one's own figures.

Having in mind that design patterns bridge the world of programming with the world of system design, the first approach is clearly out of scope. Students do not yet have the skills to start with nothing but the requirements. Deciding between the second and the third approach was more difficult.

Although the third approach seems to fit more a programming than a designing course, we decided to follow it as it has a flat learning curve. Students can concentrate on features of the editor and on the application of the design patterns. They do not have to deal with GUI problems in the first place. For talented students, the second approach is still open, i.e. they can add their own extensions to the editor not yet covered by our framework.

3.1 Provided Interfaces

The fact that everybody programs against the same interfaces has several advantages:

- Figure implementations are exchangeable. Running your figures on a classmate's editor is an incentive.
- Teamwork is encouraged. It is easier for the students to understand each others implementations.
- We can provide test cases which inspect implementation details. For example, one test checks whether information hiding is implemented correctly, i.e. whether mutable objects (e.g. bounding box) returned by a figure are copied. Another test controls whether an observer may remove itself upon notification.
- It is easier for teachers to correct and comment student solutions.
- When students present malfunctions of their implementation we can give an immediate feedback due to our experience with the framework.
- The system and its interfaces can be used as a context for exam questions.
- The effort for assignment support and mentoring is reduced. The teacher does not have to deal with many completely different implementation frameworks.

3.2 Provided Design Decisions

This approach also has a disadvantage: Since we provide the framework in form of interfaces, the challenging question, where and how a pattern has to be applied in order to extend the editor with a particular feature has already been answered.

Consider the grid implementation which is based on the strategy pattern: The most difficult part in using this pattern is to design the strategy interface. With our approach however this interface is given.

One key decision when using the composite pattern is whether the composite specific methods are added to the component interface or to the composite interface (transparent vs. safe approach). Since we provide the figure interface without composite abstractions, this decision has been taken.

The same holds for the model view separation based on the observer pattern. In the framework both the model and the figures are designed as observables (they contain listener registration methods) which have to be provided by the students.

In some cases it is the system itself that prevents the exploration of some aspects of a design pattern. A good example is the problem of testing whether a component is already part of a composite in the composite pattern. This test is necessary if there is a composition relationship between component and composite. The handling of figures in the graphics editor prevents automatically a figure from being part of more than one group, no test is necessary.

3.3 Experience

Although the design is given, it turns out that the correct wiring of the observables and implementation details is still challenging for the students. The relatively flat learning curve and the familiar way of how to work on assignments encourage the students.

We also can meet the diverging needs of weaker and more talented students. In the assignments there are usually questions like: "how can you extend the system with a new feature without having to change the interfaces?" We expect all students to sketch a solution, but the better ones will also implement it.

4. BEYOND DESIGN PATTERNS

At our school software engineering courses had a wide scope on their subject area. They covered the software development process with its various facets like requirements engineering, project management, life cycle management, testing, etc. However, specification was not neglected, but covered in an in-depth UML course.

4.1 Effective Use of Design Knowledge

Such a curriculum seems to cover all the aspects of a sound software engineering education. Still students kept asking "how do I design a complex system?" Didn't we tell them how UML models work, how n-tier architectures are supposed to be implemented? Yes we did, but from the students' perspective we never actually closed the gap between programming and software design.

What happened? The programming world was a hands-on world to the students. In many assignments they had to implement various algorithms and small applications, as e.g. our graphics editor. The software engineering world however was a theoretical world. They gathered requirements, designed systems using UML diagrams. They learned how successful architectures are structured (from JSF's model-view-controller approach to 3-tier architectures up to the design of JEE or .Net frameworks).

Actually the students hardly ever had to implement their designs. Assignments were done on paper. Feedback was at best an interesting discussion with their tutor. They never had to suffer their design!

With the reforms to adjust our school to the European Credit Transfer System (ECTS), the situation improved dramatically as the students have to earn a good deal of their credits with student projects. These projects earn 6 credits corresponding to about 180 hours of working effort per semester. Now they have to suffer

their design. And here they gained similar experiences as described in [3].

Still many students wanted to know how to produce a “good” design. Whereas design patterns help to program “good” code, on the design level we found that there was still something missing.

4.2 Improve Design by Implementing it

A new elective course on Software Design should specifically address the problem of producing “good” design.

The experience with the Design Patterns course let us follow the same track: teach the discipline with plausible, but possibly theoretical, examples and then practice it with a complex system.

Again we used our graphics editor as a sufficiently complex system to exercise our learning matter. An example:

The graphics editor shall be extended to be an UML class diagram editor. Students have to sketch a first draft using CRC cards. Then they have to refine their design using UML diagrams. Eventually they have to implement the extension.

Each assignment will be commented by the teacher. But the students know that the subject is not closed with the teacher’s feedback. They steadily have to continue and build on their own basis. This way the different topics of the course automatically become intertwined. And new questions arise automatically without the need to artificially or theoretically introduce them. Compatibility with older versions, system wide exception handling, refactoring designs that turned out to be dead ends are only a few examples.

5. LESSONS LEARNED

Issuing continuous assignments that accompany the lectures has some consequences that have to be taken into consideration when planning the course.

The assignments are demanding. We counteract this problem by designing them as independent editor extensions. This way, if a student skips one, it has no impact on upcoming assignments. Furthermore we set an interleaved schedule. We issue a new assignment every week, but the students have two weeks to turn it in. Moreover, solution fragments are provided. The provision of complete solutions would demotivate the students to work on their own solutions.

There should be enough time to discuss assignments each week. Often students encounter problems that will be covered later on in the course. It is important to address these topics as they arise and not to block a discussion with a hint on the upcoming subject. This demands some flexibility from the teacher.

Working on the same system during one or two semesters can become boring. To prevent this, the assignments should...

- concentrate on the interesting parts (e.g. no fiddling with complex setup),
- deliver a visible success each time,
- be fun to work on in groups. This enables students to motivate each other.

A complex system allows demonstrating and discussing alternative approaches to a problem. For instance consider the problem of a figure not being allowed to be part of more than one group. As mentioned above, this is prevented already by the way the editor is used. This should be picked out as a central theme.

Students do plagiarize code from previous years. Instead of combating copies, we prefer to ignore them. Students know that the learning effect will be marginal if code is copied but not understood. Some students improve a code basis from previous years, e.g. to implement a more sophisticated undo/redo function.

6. OPEN ISSUES

The Design Patterns course – with the graphics editor assignment – runs now (with modifications) in its seventh year. The Software Design course has only been held twice so far.

Both courses are now adjusted to close the gap between programming and designing. As we first teach Design Patterns and then Software Design, we are bridging the gap from the programming side. Giving first the course on Software Design would open up new possibilities such as designing the whole graphics editor from scratch. However the question of providing or building a framework would arise again: The provided code is large and it would take too much time for the students to develop it themselves. If we provide some code then we implicitly provide a design as well, thus discouraging previous student efforts.

Reversing the order of the two courses would also have impact on the workload of the teacher. It would be more difficult to keep the designs of the students in line, thus increasing the effort of providing feedback.

Due to administrative constraints both courses had to be taught in the same semester last year. This gave us some insight on what impact the order of the courses could have. The experiences with this situation were mixed. On the one hand, there was a mutual benefit as some topics could be taught more efficiently due to the timing. On the other hand there were many synchronization problems and dangling forward pointers.

7. CONCLUSIONS

Teaching software engineering has a long tradition at our school. Still, we realized that students have problems to effectively apply the theoretical knowledge in concrete projects.

Our approach of presenting design patterns in the context of a continuous project mitigated this problem. Students see how patterns are applied in a complete system and gain a feeling about what is “good” design. Moreover, without needing to prompt students, they start discussing their different solution approaches among each other.

Feedback from students is positive. They like the "fun factor" of the project but also report about application of their gained design knowledge in their own student projects.

8. REFERENCES

[1] Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.

[2] Freeman, E., Freeman, E. and Sierra K.: Head First Design Patterns; O’Reilly, 2004.

[3] Stuurman, S. and Florijn, G: Experiences with Teaching Design Patterns. Proceedings of the ITICSE Conference, 151-155, 2004.

[4] JHotDraw, http://sourceforge.net/projects/jhotdraw

[5] Spring Framework, http://www.springframework.org