

Patterns for Teaching Software in Classroom

Axel Schmolitzky
University of Hamburg, Germany
Vogt-Koelln-Str. 30
D-22527 Hamburg
+49.40.42883 2302
schmolitzky@acm.org

Abstract

This paper presents pedagogical patterns for the general context of *teaching software concepts in classroom settings*. These patterns are targeted at people who teach other people, whether in industry or at universities. The patterns are presented in Alexandrian Form, in conformance with the patterns of the Pedagogical Patterns Project. Four pedagogical patterns have been identified: SHOW IT RUNNING, SHOW PROGRAMMING, MAKE IT THEIR PROBLEM and MAKE THEM MAKE IT THEIR PROBLEM.

1. Introduction

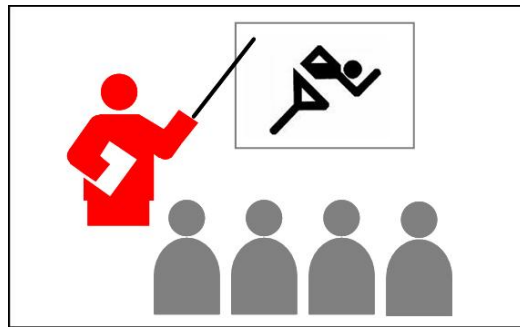
This paper presents four pedagogical patterns for the general context of *teaching software concepts in classroom settings*. These patterns are targeted at people who teach other people, whether in industry or at universities. The teaching persons are called *teachers* in the following, the taught persons are called *participants*. *Classroom setting* means that teachers and participants are together in one place at the same time for some amount of time (the *contact time*), typically for 60 or 90 minutes. Important characteristics of classroom settings are that teachers and participants can communicate directly (an important agile value) and that there are typically more participants than teachers (teaching means multiplication).

The pattern language follows the classical Alexandrian form chosen by Bergin in [4] for *pedagogical patterns* [1]: All patterns are written in the you-form, talking to the teacher. In addition to the pattern name (set in small capitals), each pattern is divided into four sections, separated by ***. The first section sets the context. The second describes the forces and the key problem. The third section outlines the solution, the consequences, limitations and disadvantages. The fourth section complements the discussion of the solution, by providing further information and examples. In addition, for each pattern its thumbnail is identified by setting the core sentences from the problem and the solution section in bold typeface.

Four pedagogical patterns have been identified: SHOW IT RUNNING, SHOW PROGRAMMING, MAKE IT THEIR PROBLEM and MAKE THEM MAKE IT THEIR PROBLEM. They are presented in sections 2 to 5 in order of increasing involvement of the participants. They are also presented in order of increasing complexity, as any of the patterns refers to all patterns presented previously.

2. SHOW IT RUNNING

You are teaching about a software tool or framework you want the participants to use. You have slides that describe the properties (features, advantages, disadvantages, etc.) of the software well, maybe supported by some screenshots that illustrate the usage of the software.



Students tend to forget easily if they just hear about the functionality of a software; hearing somebody talk about using a software can be boring. You feel uncomfortable about the slides being too theoretical on their own, catching not enough interest. But the slides form a good base for learning for the exam at the end of the semester, so you want to keep them.

Therefore, use the software during your presentation. Students remember better if they have seen it working. Use some simple scenario that makes use of the software; the more the scenario shows the particular strength or weakness of the software, the better.

Limitation: Make sure that the time you are investing is paying off. It can be quite time-consuming to work with running software; start-up time can be long, the firewall might need reconfiguration, the web server might not start, the database can be slow on your presentation machine.

If you are discussing layout management of components in a GUI framework, some well prepared resizable example GUIs will be far more instructive than any slide set.

If you want to discuss unit testing with JUnit, a running demonstration producing a red and a green bar is more impressive than pure slides.

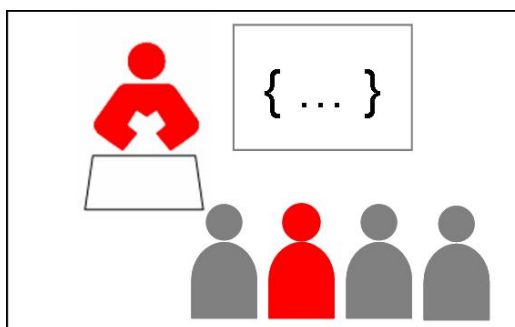
Limitation: Finding and preparing a good scenario can be time consuming; you have to weigh this against the improved learning effect. Keeping a good scenario running over time (with operating system and application updates in between) can be time consuming as well.

Try to make sure that text messages are readable for the audience and that the windows are arranged the right way; make this part of the scenario.

Think aloud while using the software. Make sure that you explain everything you are doing with the software; it is new for the participants and they are not as fluent as you might be.

3. SHOW PROGRAMMING

You are teaching a programming language, a particular programming language concept or a programming technique (such as refactoring, unit testing, or a programming idiom). You are using well-prepared slides that discuss the subject with good source code examples.



If participants ask about variations of the examples on your slides, you can only tell, not show (if you know the answer); if you don't know the answer, you and the participants will feel uncomfortable and unsatisfied after the teaching unit. Even the best slides can be too inflexible for you to react on participants' questions. Quite often participants ask about variations of the examples shown. If you know the answer and tell it, things are good for you but not for the participants; they just hear the answer, they do not see it working. Things are worse if you are not sure about the answer; so you answer "probably" and "try it yourself at home", which implies that participants have to refocus on the question some time later, alone. Most will not do this, either due to time constraints or due to lack of interest.

Therefore, do not just show slides about programming, show programming as well. Start an integrated development environment (IDE) during your presentation. Explain the source code you provide, then show how you apply the concept you are trying to explain. Show how you make use of useful features of the IDE. When participants ask about variations, you answer the question and then show the answer in action. This way, you provide a simple kind of TEST TUBE [6] during your presentation.

You need to be quite fluent in the language you are showing, but you need not be an expert. If you know every little detail about a language, you impress the participants with your deep knowledge; but the participants do not necessarily learn better by this. If you have to try the solution yourself to be sure about the answer, participants feel closer to what you are doing; so sometimes it can help if you fake to not know the solution.

Limitation: You need more preparation time for the lecture. You have to check that the software is running on your presentation machine and you have to program the examples from your slides.

Limitation: It can be quite time-consuming to work with running software; start-up time can be too long, online documentation can be clumsy to use, the web server might not start, the database can be slow.

Limitation: This pattern starts to become bulky as soon as you try to compare language mechanisms in different languages; starting two or even more IDEs can be too much for one presentation.

If you want to discuss unit testing with JUnit, a running demonstration allows a better exploration of variations.

If you want to teach 'test first', doing it in front of the audience will be more instructive than a dry recipe.

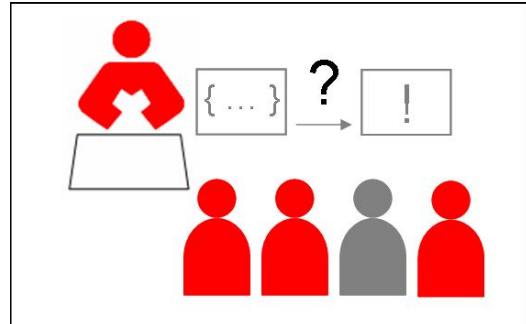
BlueJ [7, 11] is an IDE for teaching Java programming and is well-suited for classroom teaching; it is small enough to be running on any presentation machine and offers, beside other nice features, a code pad for executing simple expressions immediately.

The teacher, if an experienced programmer, can become a role model for the participants, as she can show tips and tricks and can demonstrate best practices in the IDE and/or the programming language (see e.g. [10] for a timely discussion of apprentice-based learning).

Make sure that the source code is readable for the audience (font size, window arrangements, etc.).

4. MAKE IT THEIR PROBLEM

You are teaching a fundamental design pattern or an important programming language concept. You want to make sure that all participants have a thorough understanding of the subject by letting them TRY IT YOURSELF [8] in the classroom, but the group is too large for working in a lab where each participant has its own computer.



Students will not understand well without applying the imparted knowledge; if they apply it on their own they do not get immediate and qualified feedback on their work which can manifest wrong understandings. Typically there is not enough time to set a task that participants can solve offline and then to give each participant individual feedback on the solution. A pure slide presentation, on the other extreme, is the most time-effective way of imparting knowledge, but feedback about participants' individual understanding is typically sparse. You can improve the learning effect by applying SHOW PROGRAMMING, but you still feel uncomfortable about the engagement of the participants; you want them to become ACTIVE STUDENTS [5].

Therefore, set up an environment where all participants of a teaching unit get to know a live running system, then set a problem the audience is supposed and able to solve; let the audience agree on a solution and let them direct you to realize this solution, visible to all. Finally reflect thoroughly on the way the general solution was applied to the specific problem and on other ways or contexts where the general solution can be helpful. You can achieve that everybody can see the initial system and its source code simply by using a single presentation computer connected to a projector.

Try to make sure that the initial system and the problem lead to the demonstration of a *killer example*; a killer example for a design pattern is one which “gives overwhelmingly compelling motivation” for using a pattern [2].

Limitation: This pattern can be oversized for teaching simple programming language concepts, such as conditionals or loops, as the setup of a problem can easily take too much time in comparison to the gain of using the pattern.

Limitation: This pattern might not work with more than 30 participants as it takes more courage for a participant of a larger group to actively join a design discussion.

Introduce the initial system both in its functionality (using SHOW IT RUNNING) and its internal structure (source code) in an IDE (using SHOW PROGRAMMING). Engage the participants by asking which clicks to perform or which class definition to show next. Make sure that everybody has a good understanding of the initial system and feels confident to extend the system; thus the initial system should be as small as possible, but not smaller. Even more as in SHOW PROGRAMMING, you should provide a TEST TUBE [6] for experimentation.

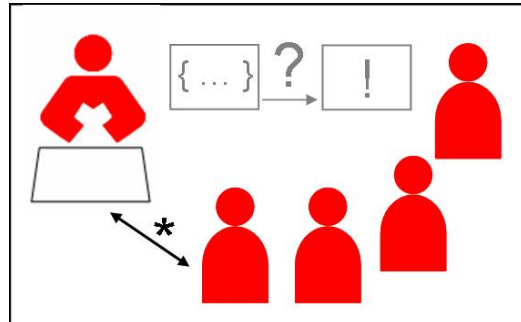
Provide information describing a general solution that can be helpful for the specific solution. You can do this before you set the problem or afterwards, depending on the difficulty of finding a solution for the problem.

Do not fall into SHOW PROGRAMMING, i.e. you being the main person in control of programming; you have to deal with giving up complete control over a teaching unit. The orders for the next programming step should always come from the audience. Ideally, during the design and implementation part of MAKE IT THEIR PROBLEM, you become an INVISIBLE TEACHER [5], while the participants have a lively discussion about different alternatives in the form of a STUDENT DESIGN SPRINT [9] with a lot of REFLECTION [6]. But you should always have a programmed solution up the sleeve that you can show in case you run out of time.

Teachlets, as described in [12], are a specific teaching method that builds on executable code in a teaching unit. It encourages highly interactive classroom settings through introducing a running piece of software in its source code and setting a task to extend this software; the participants then have to find a solution collaboratively and to tell the moderator how to implement this solution in front of the audience. Teachlets have been used so far for teaching design patterns and programming language concepts, but might be applicable to teaching algorithms as well.

5. MAKE THEM MAKE IT THEIR PROBLEM (TEACHLET WORKSHOP)

You are teaching a course on advanced software concepts (e.g. design patterns, advanced concepts of object-oriented programming, advanced computer graphics). The participants have good knowledge of the required prerequisites and are eager to learn.



You want to engage the participants as much as possible and use the time of the course as effectively as possible, but time constraints do not allow you to prepare teachlets for the whole course. You want to make sure that certain topics are covered and well understood. A pure talk would be too much of a one-way street, with only little engagement of the participants. Because the subject of the course is covering a field with rapid change (e.g. advanced computer graphics), the overhead for keeping slides or teachlets up to date is too much on your side.

Therefore, let participants design teaching units using MAKE IT THEIR PROBLEM; let them conduct these units and organize intense FEEDBACK [8] after each. Having to prepare and conduct a teaching unit with slides and running software implies a deeper involvement of the preparing participant with the subject; she becomes an even more ACTIVE STUDENT [5] than an active participant of a MAKE IT THEIR PROBLEM unit.

If a unit of MAKE IT THEIR PROBLEM is ill-prepared or the participant is not a good presenter, an important topic might not get the appropriate coverage. You have to be prepared to give additional background on the topic in the feedback phase. Typically the feedback phase should also include PEER FEEDBACK [8].

Limitation: Designing a unit of MAKE IT THEIR PROBLEM requires some creativity. If participants do not like to be creative, this pattern can be too demanding.

Limitation: Conducting a unit of MAKE IT THEIR PROBLEM requires several soft skills (e.g. the self-assured handling of an IDE, slides and a video projector in front of a group of people, the moderating of design discussions). If participants are not self-confident or experienced enough, this pattern can be too demanding.

If you have some flexibility for your course content, you can let the STUDENTS DECIDE [5] for which learning goals they should build their teaching units.

If participants do not want to be creative, they can do a *teachlet replay*: they take a teachlet from a previous workshop, work it over and conduct it again. This is an implementation of ADOPT-AN-ARTIFACT [5].

If participants are not self-confident enough on their own, you can let them prepare and conduct their teaching units in pairs, as GROUPS WORK [5] often better; this worked well for one pair in the last teachlet workshop conducted by the author.

As weeks in a semester are a restricted resource, the number of (active) participants of a TEACHLET WORKSHOP is typically restricted to 10 to 15 people.

In a postgraduate course on advanced computer graphics at the University of Hamburg (not conducted by the author), students developed teaching material that could be used in a teaching unit as well as for individual offline learning (in a format that could be submitted to CGEMS, a computer graphics educational materials server). Especially here, a TEACHLET WORKSHOP can provide lasting material as STUDENT EXTENDS [4].

Teachlets can be used in a seminar-like workshop, where participants develop new teachlets, conduct these and get feedback on their work in a so called *Teachlet Laboratory*. By being in the teaching position, participants get an even better understanding of the subject to teach. The teachlet concept and the teachlet laboratory have been introduced at the OOPSLA 2005 Educators' Symposium [12]. Several teachlet laboratories have been conducted by the author. Another teacher at the University of Hamburg has adopted and extended the concept for a course to produce learning material for advanced computer graphics topics [3].

In the last TEACHLET LABORATORY conducted by the author, students anonymously graded different aspects of each others TEACHLETs after the FEEDBACK of each unit on prepared ballots. The results were presented at the end of the laboratory as one form of PEER GRADING [8].

6. REFERENCED PATTERNS

In order to make this paper more accessible, the thumbnails of the pedagogical patterns that are referenced in section 2 are provided here in alphabetical order. Their full descriptions can be found via the list of references.

ACTIVE STUDENT [5]

The deep consequences of a theory are unlikely to be obvious to one who reads about, or hears about the theory. The unexpected difficulties inherent in using the theory or applying the ideas are not likely to be apparent until the theory is actually used.

Therefore: keep the students active. They should be active in class, either with questions or with exercises. They should be active out of class.

ADOPT-AN-ARTIFACT [5]

Students try to solve all problems in a similar way, using their individual thinking or problem solving process. But a lot can be learned by understanding and working with an artifact produced by somebody else.

Therefore, ask the students to improve and extend artifacts from their peers. In order to do so, they have to comprehend the way in which their assigned peers have approached their task.

FEEDBACK [8]

Unless the work is assessed and feedback is given, you won't be able to correct any misunderstandings, the students won't know where they are at fault and their learning will be incomplete.

Therefore, give the participants feedback on their performance. The feedback should be differentiated and objective.

GROUPS WORK [5]

You are only one resource for the students. Given the number and difficulty of student questions and concerns you are actually a rather small resource. Your students need frequent feedback on what they do and how they do it.

Therefore, emphasize group work in your courses. Use both large and small groups. Use both long-lived (weeks) and short-lived (minutes) groups.

INVISIBLE TEACHER [5]

Usually, the teacher is the central point of a training environment. Often the students only trust the teacher and (maybe) themselves, therefore, when students struggle, the obvious step is to ask the teacher for help. However, in the work environment the teacher will not be around.

Therefore, make the participants the focal point of the course. If a problem occurs direct them to their peers, to ask their peers for help.

PEER FEEDBACK [8]

Students are knowledgeable and are able to give helpful feedback, but often they are not confident about the relevance of their experience and are unsure about the value of their own knowledge.

Therefore, invite students to evaluate the artifacts of their peers. The students will provide feedback to their peers by drawing on their own experience and because each student will also have produced the artifact for himself or herself, their experience and knowledge will be explicitly relevant.

PEER GRADING [8]

You want to teach your students how to evaluate quality and how to negotiate for it. You want to get them to accept evaluation by peers and to make this comfortable.

Therefore, make it possible for students to provide part of the grade for other students.

REFLECTION [6]

Sometimes, learners believe that the trainer has to deliver all the knowledge, but the students themselves are knowledgeable. Furthermore, students often anticipate that an instructor will solve each and every problem for them, but the knowledge of the instructor is also limited.

Therefore, provide an environment that allows discovery and not one that is limited to answering questions. Let the students uncover solutions for complex problems by drawing on their own experience.

STUDENT DESIGN SPRINT [5, 9]

Students need to solve problems in teams. They also need quick feedback and peer review of early attempts. They eventually need to solve complex problems, but may need help on simpler problems as well. If we don't teach them problem solving they will develop their own ad-hoc techniques that may reinforce bad habits.

Therefore, use some variation of the following highly structured classroom activity. Divide the students into groups of two or three. Give them a problem and have them develop a solution in 15-20 minutes in their groups. There should be a written outline of the solution produced by each team. The instructor can look over shoulders and comment, but few hints should be given.

STUDENT EXTENDS [4]

Students and instructors often find that the provided materials don't meet their needs. In addition, many student activities, such as most homework, have no intrinsic value other than as etudes to get a student to practice.

Therefore, involve the students in improving the classroom materials.

STUDENTS DECIDE [5]

Sometimes it is impossible, to make decisions concerning course material and approach in advance, because the exact skills or interests of the participants are not known.

Therefore, involve the participants in the planning of the course, or suggest some alternatives at the beginning of the course. Give them a voice in choosing among the alternatives.

TEST TUBE [5, 6]

When students encounter holes in their knowledge, we would like for them to seek out an answer. Unfortunately, students often resort immediately to the "easy fix" of asking an authority for the answer. We want students to ask questions, but sometimes they have available to them more effective ways to gain knowledge that they never consider. In many courses experimentation is the one viable method.

Therefore, give the students exercises in which they are asked find the answer to simple questions of the form "What happens if ...?" using experimentation. In a programming course, the machine itself can answer many such questions, for example. Make these exercises frequent enough that students develop the habit of probing the machine for what it does, rather than asking a question or seeking out documentation.

TRY IT YOURSELF [5, 8]

Students usually believe they have understood the topic, but this is often only true in theory. As soon as they have to accomplish a task that is based on this new topic they realize their lack of understanding.

Therefore, take a break in the presentation and ask the students to perform an exercise that requires them to understand the new topic and for which you can give immediate feedback.

7. CONCLUSION

In this paper, four pedagogical patterns for teaching about software in classroom settings have been presented. These patterns have been extracted from several teaching units on software topics such as design patterns, programming language concepts and advanced computer graphics. We further pointed out how these patterns relate to pedagogical patterns previously published. As Bergin notes in [4], some of these patterns might seem obvious, even trivial for professional teachers. But there is a chance that they are a valuable input for educators looking for ways to improve their teaching.

8. ACKNOWLEDGEMENTS

My thanks go to the participants of several teachlet workshops for their great commitment and to my colleagues (both in the SWT Group at the University of Hamburg and at C1 Workplace Solutions GmbH) for several fruitful discussions on patterns in teaching software. Steffi Beckhaus had the patience to listen to my ideas about teachlets and adopted them for her course on advanced computer graphics. Christian Späh supported me with his enthusiasm about teachlets and contributed the peer grading ballots to last year's teachlet workshop.

A previous version of this paper was written for PLoP 2006 and had to be withdrawn for personal and organizational reasons. I have to thank Joe Bergin for his invaluable input as my shepherd during the preparation for PLoP.

This version was shepherded by Peter Sommerlad for EuroPLoP 2007. I very much enjoyed the shepherding session with Peter during the Software Engineering 2007 conference in Hamburg. Being much older than me (at least one year), he provided great ideas for improving the paper, and I hope I managed to consider them all. Thanks, Peter!

9. REFERENCES

- [1] The Pedagogical Patterns Project, <http://www.pedagogicalpatterns.org>, (last visited June 10, 2007).
- [2] Alphonse, C., Caspersen, M. and Decker, A., Killer "killer examples" for design patterns. In *Proc. 38th SIGCSE technical symposium on Computer Science Education*, (Covington, Kentucky, USA, 2007), ACM Press, 228-232.
- [3] Beckhaus, S. and Blom, K.J., Teaching, Exploring, Learning - Developing Tutorials for In-Class Teaching and Self-Learning. In *Proc. EUROGRAPHICS '06 (Education Papers)*, (Vienna, 2006).
- [4] Bergin, J., Active Learning and Feedback Patterns. In *Proc. PLoP '06*, (Portland, Oregon, 2006).
- [5] Bergin, J., Eckstein, J., Manns, M.L. and Sharp, H., Patterns for Active Learning. In *Proc. PLoP '02*, (Monticello, Illinois, 2002).
- [6] Bergin, J., Eckstein, J., Manns, M.L. and Wallingford, E., Patterns for Gaining Different Perspectives. In *Proc. PLoP '01*, (Monticello, Illinois, 2001).
- [7] BlueJ - The Interactive Java Environment, <http://www.bluej.org>, (last visited June 10, 2007).
- [8] Eckstein, J., Bergin, J. and Sharp, H., Feedback Patterns. In *Proc. EuroPLoP '02*, (Irsee, Germany, 2002).

- [9] Eckstein, J., Manns, M.L., Wallingford, E. and Marquardt, K., Patterns for Experiential Learning. In *Proc. EuroPLoP '01*, (Irsee, Germany, 2001).
- [10] Kölling, M. and Barnes, D.J., Enhancing Apprentice-Based Learning of Java. In *Proc. SIGCSE 36*, (Norfolk, Virginia, 2004), 286-290.
- [11] Kölling, M., Quig, B., Patterson, A. and Rosenberg, J. The BlueJ system and its pedagogy. *Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology*, 13 (4), 2003. 249-268.
- [12] Schmolitzky, A., A Laboratory for Teaching Object-Oriented Language and Design Concepts with Teachlets. In *Proc. OOPSLA '05 (Companion: Educators' Symposium)*, (San Diego, CA, 2005), ACM Press.