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 about software, 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, GROUP DESIGN CHALLENGE and LEARNERS DO CHALLENGE.

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, about software. The teaching persons are called *teachers* in the following; the taught persons are called *learners*. *Classroom setting* means that teachers and learners 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 learners can communicate directly (an important agile value) and that there are typically more learners than teachers (teaching means multiplication).

The patterns focus on teaching *software concepts*, such as *using*, *designing* and *programming software artefacts*. Software is "a very special juice" in many respects (beside others, it is immaterial, completely abstract, arbitrarily reproducible, and arbitrarily complex), thus the teaching of software differs considerably from teaching other subjects. Even though the patterns seem to have the potential for being generalized beyond teaching software, the author deliberately kept their focus; they are targeted at people in IT that are looking for ways to improve their teaching.

The patterns are presented in classical Alexandrian form, as chosen by Joe Bergin in [8] 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 positive 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, GROUP DESIGN CHALLENGE and LEARNERS DO CHALLENGE. They are presented in sections 2.1 to 2.4 in order of increasing involvement of the learners. They are also presented in order of increasing complexity, as any of the patterns refers to all patterns presented previously.

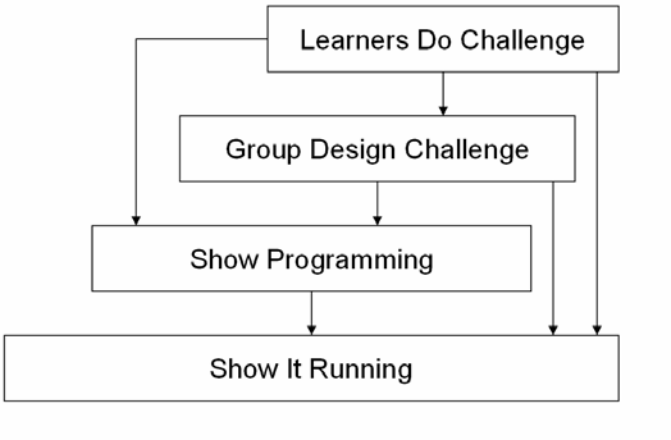


Figure 1: Dependencies between the patterns

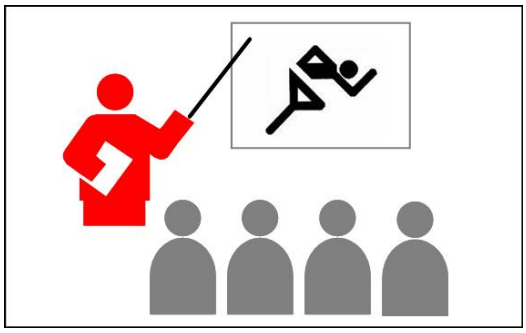
Figure 1 shows the dependencies between the patterns. The size of the box for each pattern also indicates the applicability of a pattern: the larger the box, the more general the pattern. SHOW IT RUNNING is the most general and will be presented first.

In order to make this paper more accessible, the thumbnails of the pedagogical patterns that are referenced in section 2 are provided in section 3.

As an appendix, the thumbnails of the patterns presented in this paper are summarized on one page after the references. One shepherd asked for such a one-page summary, so that he can put it on his office wall as a reminder for his own teaching.

2. The Patterns

2.1 SHOW IT RUNNING



You are teaching about a software concept, a software tool or framework you want the learners 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 it. The slides form a good base for learning for the exam at the end of the semester or course, so you want to keep them for the learners.

***

Slides are mainly static, using software is dynamic; thus slides are typically not well suited to capture the characteristics of software. You feel uncomfortable about the slides being too theoretical on their own, catching not enough interest. Learners will have problems if they just hear about the functionality of software.

***

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

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 (e.g. several semesters, with operating system and application updates in between) can be time consuming as well.

***

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 the interface of a general list (an unlimited collection with duplicates, with the order of elements controlled by the client), showing the usage of a play list in iTunes will be very illustrative.

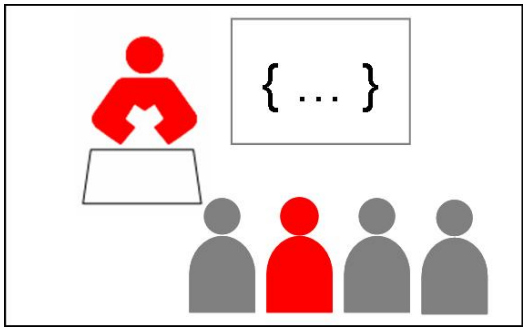
If you want to discuss unit testing with JUnit, a running demonstration producing a red and a green bar is more impressive than pure slides.

Try to make sure that text messages are readable for the audience and that the windows are arranged the right way; consider this while preparing 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 learners and they are not as fluent as you might be.

This pattern is the most basic and most universal of the patterns presented in this paper. It can be applied in any context where software concepts have to be taught, i.e. in a university lecture, a seminar or any course where people are together in the same room.

2.2 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 learners ask about variations of the source code examples on your slides, you can only tell, not show (if you know the answer); if you do not know the answer, you and the learners will feel uncomfortable and unsatisfied after the teaching unit. **Programming is a unique interplay of static properties (at writing time) and dynamic properties (at execution time) that is hard to capture with slides; quite often, subtle variations can have major effects.** Even the best slides can be too inflexible for you to react on learners' questions. Quite often learners ask about variations of the examples shown. If you know the answer and tell it, things are good for you but not for the learners; 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 learners 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. Make sure that the source code is readable for the audience (font size, window arrangements, etc.). Explain the source code you provide, then show how you apply the concept you are trying to explain. When learners ask about variations, you answer the question and then show the answer in action. This way, you provide a simple kind of TEST TUBE [10] (see also section 3 for the thumbnail) during your presentation. Learners get immediate feedback and are encouraged to be more active.

Limitation: You need to be quite self-confident and fluent in the programming language that you want to use during a presentation.

Limitation: You need more preparation time for the presentation, as you have to check that the software is running in the IDE on your presentation machine.

Limitation: It can be quite time-consuming to work with running software inside an IDE; 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 can 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. But Peter Sommerlad reported that he compares C++ and Java programming inside Eclipse in his teaching and that it works very well.

***

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; Till Schümmer experienced this at the XP 2000 conference in a session with Erich Gamma.

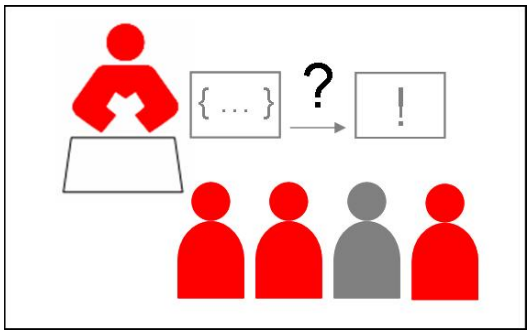
Railscasts [2], the screencasts for *Ruby on Rails* [3], are good examples of showing programming in action, but only in a non-interactive manner.

BlueJ [11, 15] is an IDE for teaching Java programming and is well-suited for class-room teaching; it is small enough to be running on any presentation machine and offers, besides other nice features, a code pad for evaluating simple expressions immediately.

Show how you make use of useful features of the IDE. This way you, if an experienced programmer, can become a role model for the learners, as you can show tips and tricks and can demonstrate best practices in the IDE and/or the programming language (see e.g. [14] for a timely discussion of apprentice-based learning in programming).

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 learners with your deep knowledge; but the learners do not necessarily learn better by this. If you have to try the solution yourself to be sure about the answer, learners feel closer to what you are doing; so sometimes it can help if you fake to not know the solution.

2.3 GROUP DESIGN CHALLENGE



You are teaching a fundamental design pattern or an important programming language concept. You want to make sure that all learners have a thorough understanding of the subject by saying: TRY IT YOURSELF [12] in the classroom, but the group is too large for working in a lab where each learner has its own computer.

***

Learners will not understand abstract design or programming concepts well without applying the imparted knowledge; but if they apply it on their own they do not get immediate and qualified feedback on their work which can manifest wrong understandings. Often there is not enough time or capacity to set a task that learners can solve offline and then to give each learner 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 learners' 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 learners; you want them to become ACTIVE STUDENTS [9].

***

Therefore, set up an environment where all learners get to know a live running system, then set a problem they are supposed and able to solve; let the learners 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 [5].

Limitation: Conducting a GROUP DESIGN CHALLENGE 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 you as a teacher are not self-confident or experienced enough, this pattern can be too demanding.

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 can be time-consuming if reaching a consensus for a design decision is difficult in the group. You as the teacher must have the courage to cut lengthy discussions to keep the schedule, without being insensitive.

Limitation: This pattern might not work with more than 30 learners 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 learners 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 [10] 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.

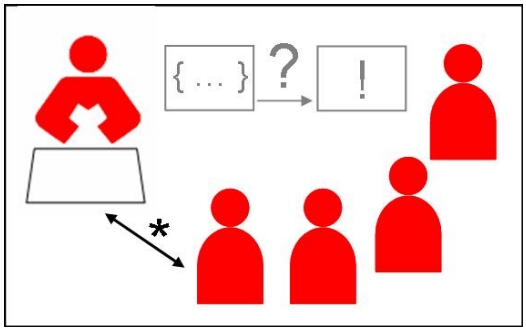
Do not fall into SHOW PROGRAMMING, i.e. you being the main person in control of programming; you have to deal with giving up complete control over a teaching unit. The orders for the next programming step should always come from the audience. Ideally, during the design and implementation part of a GROUP DESIGN CHALLENGE, you become an INVISIBLE TEACHER [9], while the learners have a lively discussion about different alternatives in the form of a STUDENT DESIGN SPRINT [13] with a lot of REFLECTION [10]. But you should always have a programmed solution up the sleeve that you can show in case you run out of time.

There is a strong relation to *pair programming* (one of the primary practices of Extreme Programming [6]) in this pattern. When programming in a pair, two programmers constantly communicate about design decisions that have to be made for the task at hand. GROUP DESIGN CHALLENGE can be seen as an extension of a pair programming session to a larger group of people that work together on a design task that was set for teaching reasons.

Teachlets, as described in [16], are a specific teaching concept 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 learners 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 or database design as well.

Be careful: In the U.S., there is a brand name “teachlet” which is not related to the teachlets described here. The brand denotes “Educational services, namely providing *online*, interactive educational services incorporating graphics, video, and audio, […]”. Thus the brand describes *web teachlets*, which are similar to *thinklets* [4]. Both denote small programs that run inside a web browser and help visualizing concepts in math, computer science, or other academic areas, whereas a *classroom teachlet*, as described in [16], is a highly interactive teaching unit for classroom teaching.

2.4 LEARNERS DO CHALLENGE



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

***

You want to engage the learners as much as possible and use the time of the course as effectively as possible; time constraints do not allow you to prepare GROUP DESIGN CHALLENGEs for the whole course. Furthermore, if the subject of the course is covering a field with rapid change (e.g. advanced computer graphics), the overhead for keeping GROUP DESIGN CHALLENGEs up to date (if you already have some) is too much on your side.

***

Therefore, let learners design GROUP DESIGN CHALLENGEs; let them conduct these and organize intense FEEDBACK [12] after each. By being in the teaching position, learners get an even better understanding of the subject to teach. Having to prepare and conduct a teaching unit with slides, running software and a problem to solve implies a deeper involvement of the preparing learner with the subject; she becomes an even more ACTIVE STUDENT [9] than an active participant of a GROUP DESIGN CHALLENGE.

If a GROUP DESIGN CHALLENGE is ill-prepared or the learner 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 [12].

Limitation: *Designing* a GROUP DESIGN CHALLENGE requires some creativity. If learners are not able to be creative, this pattern can be too demanding. Similarly, if you as a teacher, tutoring the design of a GROUP DESIGN CHALLENGE, are not creative, this pattern can be too demanding.

Limitation: *Conducting* a GROUP DESIGN CHALLENGE 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 learners are not self-confident or experienced enough, this pattern can be too demanding.

***

The classroom teachlets mentioned in section 2.3, as an instantiation of GROUP DESIGN CHALLENGE, have been used in several seminar-like workshops (*teachlet workshops*) where learners developed new teachlets, conducted these and got feedback on their work. The teachlet concept and the concept of a teachlet workshop have been

introduced at the OOPSLA 2005 Educators' Symposium [16]. Several teachlet workshops have been conducted by the author since 2004.

Steffi Beckhaus has adopted and extended the concept for a postgraduate course at the University of Hamburg on producing learning material for advanced computer graphics topics [7]. In this course, 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 course where LEARNERS DO CHALLENGE can provide lasting material as STUDENT EXTENDS [8].

If you have some flexibility for your course content, you can let the STUDENTS DECIDE [9] for which learning goals they should build their GROUP DESIGN CHALLENGEs. In the courses on object-oriented programming concepts conducted as teachlet workshops by the author, students could select from a predefined set of design patterns (such as observer, façade, interpreter) and programming language concepts (such as multiple inheritance, multiple dispatch, etc.).

If learners do not want to be creative, they can do a *replay*: they take a GROUP DESIGN CHALLENGE from a previous workshop, work it over and conduct it again. This is an implementation of ADOPT-AN-ARTIFACT [9].

If learners are not self-confident enough on their own, you can let them prepare and conduct their teaching units in pairs, as GROUPS WORK [9] often better; this worked very well for the last teachlet workshop conducted by the author where all learners worked in pairs.

As time in a course (e.g. weeks in a semester) are a restricted resource, the number of (active) learners of a course applying LEARNERS DO CHALLENGE is typically restricted to 10 to 15 people.

In the last two teachlet workshops conducted by the author, the participants anonymously graded different aspects of each others GROUP DESIGN CHALLENGEs on prepared ballots immediately after each conduction, before the open FEEDBACK round. The results were presented at the end of the workshop as one form of PEER GRADING [12].

3. Referenced Pedagogical Patterns

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 [9]

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 [9]

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 [12]

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 [9]

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 [9]

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 [12]

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 [12]

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 [10]

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 [9, 13]

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 [8]

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 [9]

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 [9, 10]

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 [9, 12]

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.

4. 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 [8], 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.

5. 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 first 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.

The second 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!

This final version has improved a lot due to the comments of the participants of the Writers' Workshop B at EuroPLoP 2007 – thanks to Birgit, Marina, Nicole (skype-ing with us from Hong Kong), Aliaksandr, Amir, Christian, Peter and Till for their great feedback. A special thank you goes to Till for another round of feedback during ECOOP 2007 in Berlin.

6. References

[1] The Pedagogical Patterns Project, http://www.pedagogicalpatterns.org, (last visited January 16, 2008).

[2] Railscasts, http://railscasts.com/, (last visited January 16, 2008).

[3] Ruby on Rails, http://www.rubyonrails.org/, (last visited January 16, 2008).

[4] SHU Thinklets, http://www.cs.shu.edu/thinklets/, (last visited January 16, 2008).

[5] Alphonce, 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, pp. 228-232.

[6] Beck, K. and Andres, C. *Extreme Programming Explained - Embrace Change (2nd Ed.)*. Addison-Wesley, 2004.

[7] 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).

[8] Bergin, J., Active Learning and Feedback Patterns. In *Proc. PLoP '06*, (Portland, Oregon, 2006).

[9] Bergin, J., Eckstein, J., Manns, M.L. and Sharp, H., Patterns for Active Learning. In *Proc. PLoP '02*, (Monticello, Illinois, 2002).

[10] Bergin, J., Eckstein, J., Manns, M.L. and Wallingford, E., Patterns for Gaining Different Perspectives. In *Proc. PLoP '01*, (Monticello, Illinois, 2001).

[11] BlueJ - The Interactive Java Environment, http://www.bluej.org, (last visited January 16, 2008).

[12] Eckstein, J., Bergin, J. and Sharp, H., Feedback Patterns. In *Proc. EuroPLoP '02*, (Irsee, Germany, 2002).

[13] Eckstein, J., Manns, M.L., Wallingford, E. and Marquardt, K., Patterns for Experiental Learning. In *Proc. EuroPLoP '01*, (Irsee, Germany, 2001).

[14] Kölling, M. and Barnes, D.J., Enhancing Apprentice-Based Learning of Java. In *Proc. SIGCSE 36*, (Norfolk, Virginia, 2004), pp. 286-290.

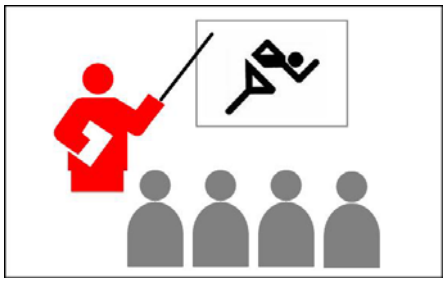
[15] Kölling, M., Quig, B., Patterson, A. and Rosenberg, J. The BlueJ system and its pedagogy. *Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology*, 13 (4), 2003, pp. 249-268.

[16] 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.

Patterns for Teaching Software in Classroom

Axel Schmolitzky
University of Hamburg, Germany
schmolitzky@acm.org

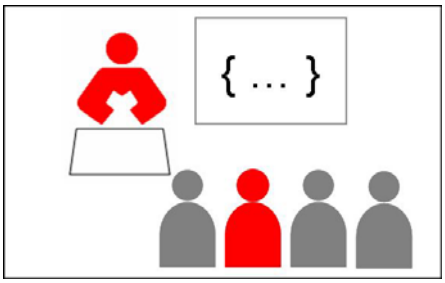
SHOW IT RUNNING



Slides are mainly static, using software is dynamic; thus slides are typically not well suited to capture the characteristics of software.

Therefore, use the software during your presentation.

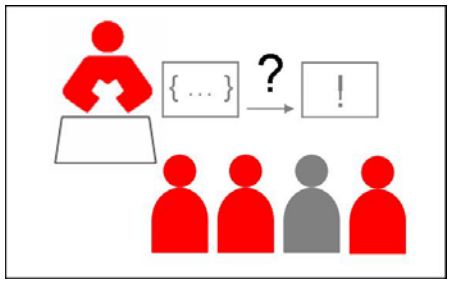
SHOW PROGRAMMING



Programming is a unique interplay of static properties (at writing time) and dynamic properties (at execution time) that is hard to capture with slides; quite often, subtle variations can have major effects.

Therefore, do not just show slides about programming, show programming as well.

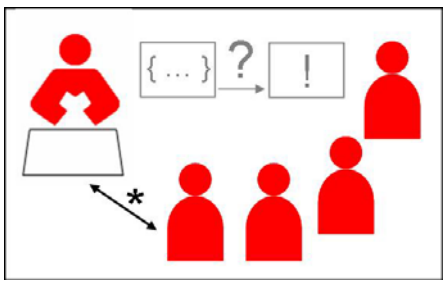
GROUP DESIGN CHALLENGE



Learners will not understand abstract design or programming concepts well without applying the imparted knowledge; but if they apply it on their own they do not get immediate and qualified feedback on their work which can manifest wrong understandings.

Therefore, set up an environment where all learners get to know a live running system, then set a problem they are supposed and able to solve; let the learners agree on a solution and let them direct you to realize this solution, visible to all.

LEARNERS DO CHALLENGE



You want to engage the learners as much as possible and use the time of the course as effectively as possible; time constraints do not allow you to prepare GROUP DESIGN CHALLENGES for the whole course.

Therefore, let learners design GROUP DESIGN CHALLENGES; let them conduct these and organize intense feedback after each.