Patterns for Experiential Learning

Submission to the PPP pattern language
project on experiential learning

Editors: Jutta Eckstein jeckstein@acm.org
Klaus Marquardt marquardt@acm.org
Markus Völter voelter@acm.org

Introduction

In general, teaching is about flexibility. Every teacher should have a set of techniques to create an effective teaching and learning experience. A teacher should collect these effective techniques over time, perhaps creating a personal pattern language or finding new patterns to add to this pattern language.

This pattern language, which is work in progress, collects proven techniques for teaching effectively. For professional educators, these patterns might seem obvious, even trivial, because they have used them so often. But for those newer to teaching, these patterns offer a way for experienced teachers to pass on their tried and true techniques.

However, patterns are not step by step recipes. These patterns offer a format and a process for sharing successful teaching practices, and can be used by a variety of people in many different ways.

Introduction to Patterns and Pattern Languages

This pattern language under construction is for instructors in industry and academia. They will probably not be familiar with patterns and pattern languages. That's why we include a small sections on this topic. If you are familiar with patterns and pattern languages you can skip this section.

The first pattern language was a called „A Pattern Language – Towns, Buildings, Constructions" and was published in 1977 by the architect Christopher Alexander et. al. [CA]. He defines a pattern as follows:

> *Each pattern describes a problem that occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.*

That means, that patterns offer a format and a process for sharing successful practices in a way that allows each practice to be used by a variety of people in many different ways.

Alexander introduces 253 patterns in the architecture domain. He presents patterns for everything from designing independent regions, to cities, to buildings and even to designing single rooms. By relating these patterns within a common problem space he transforms this

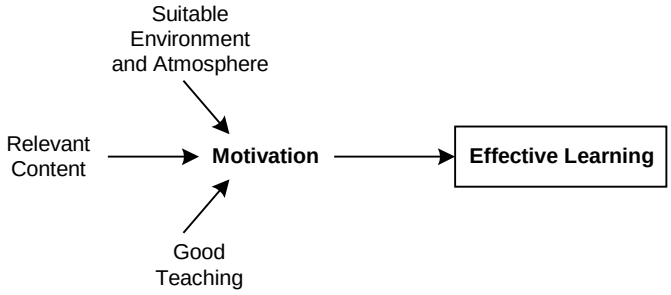
collection of patterns to a pattern language. It provides a consistent way to create a comfortable environment for people to live in.

At the beginning of the nineties the software community started using Alexander's technique to capture and communicate wisdom in software development. The movement began at OOPSLA, a major object-oriented conference. The first book that was publicly available was „Design Patterns" by Gamma, Helm, Johnson, and Vlissides called the Gang of Four [GoF]. It was published in 1994 and presented a catalog of 23 patterns on how to design software systems. In the meantime, domain specific patterns and pattern languages have been created, and the pattern movement has its own set of conferences, namely the PLOP, EuroPLOP, ChiliPLOP and KoalaPLOP (PLOP stands for Pattern Languages of Programs). Many of the patterns presented at these conferences can be found in [PLOP1, PLOP2, PLOP3, PLOP4]

In the more recent past the scope of the pattern languages expanded once again, now including patterns of group working, designing software in groups, and pedagogic patterns, that deal with the problem of how to teach (especially IT topics). This pattern language in progress is intended as another addition to the pedagogic pattern movement, which can be found on the internet at [PPP].

Prerequisites

People are the central focus of teaching. Therefore, the patterns have to deal with biological and social basics that cannot be ignored. Taking these basics into account results in motivated participants, which in turn leads to successful learning. This section highlights some of them.



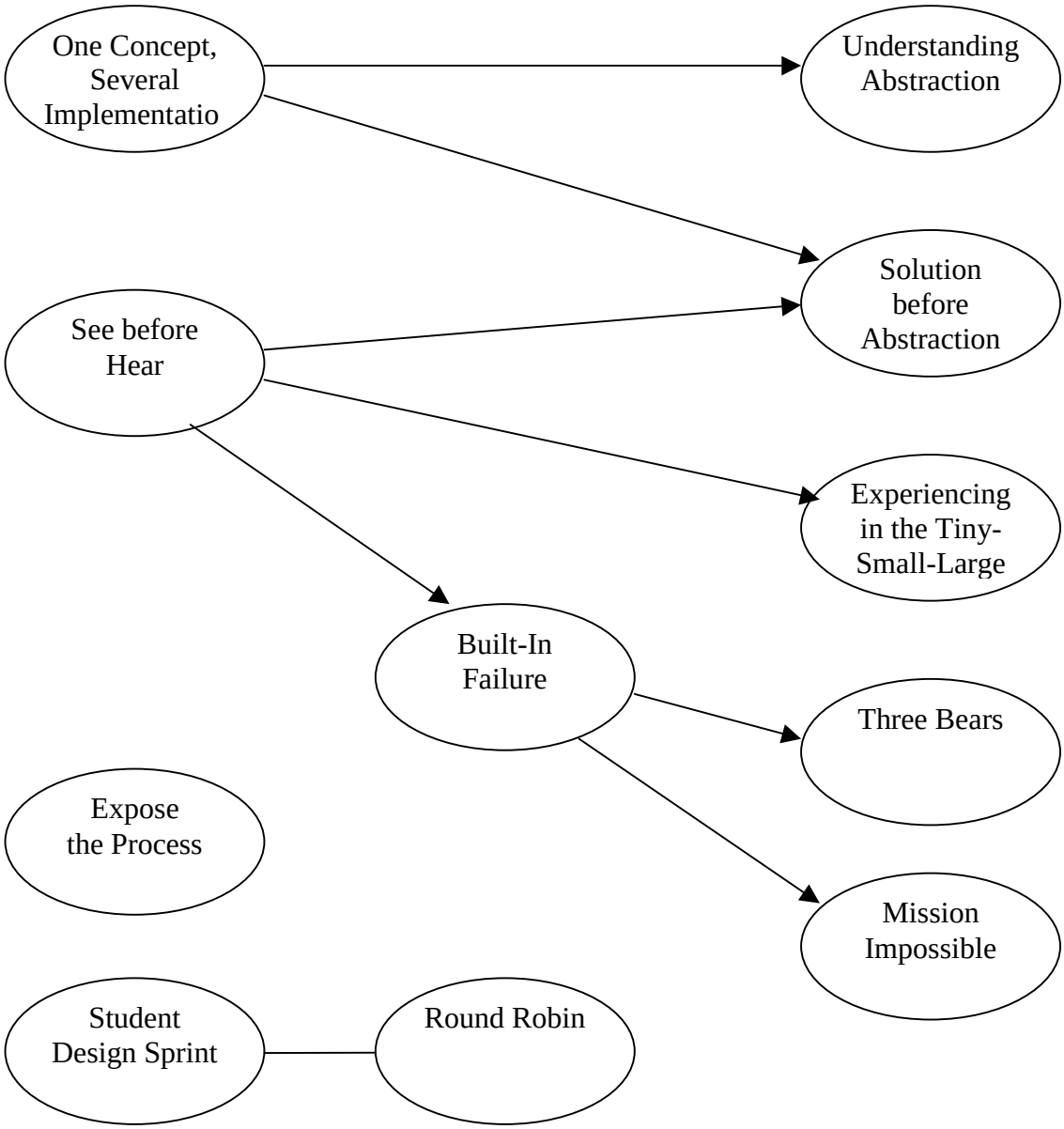
Push and Pulls

This section explores some of the global forces these patterns address. Each pattern tries to resolve these forces in a different way.

- **Learning Efficiency:** As described above, "experiential learning" is one of the most efficient ways of learning.
- **Time Consumption:** Compared to lecturing, this kind of teaching takes up more time in a course or seminar. This might conflict with the amount of topics that have to be covered in total.
- **Teacher work:** Usually, this kind of teaching requires more work and preparation for the teacher.
- **Topic suitability:** Some topics cannot easily be taught this way, for some it's easier.

Roadmap

This pattern language under construction contains patterns from the Pedagogical Patterns effort [PPP] which were refactored and rewritten in Alexanderian form in order to support the integration into a pattern language. The currently available patterns focus on a classroom situation at beginners to advanced level, but their usability is not limited to that. Further patterns will be submitted to future conferences of the PloP series.



ONE CONCEPT – SEVERAL IMPLEMENTATIONS

This pattern is a refactoring of Marcelo Jenkins' Design-Do-Redo-Redo [MJ] pattern, refactored by Markus Voelter.

An abstract concept is hard to understand without a concrete implementation or realization. However, teaching a concept using a concrete implementation might blur the concept itself, because usually, a concept is not realized in exactly the same way, as the concept suggests.

❊❊❊

Therefore, use several different implementations of the concept as examples while teaching the abstract concept. Compare the different implementations afterwards, to re-discover the essence, the abstract concept. You can use this pattern in the form of examples, exercises, group work, etc.

As a consequence, the students learn the abstract concept *and* see several concrete implementations. This allows them to distill the concept itself from the realizations. It is an advantage if the students are already familiar with one of the concrete realizations. If the pattern is used in the form of exercises or group work, immediate feedback is critical, to make sure the students don't implement the concept wrong several times.

❊❊❊

For example, it is hard to teach object-oriented programming concepts without binding them to a specific programming language. To change this problem, let the participants implement a small problem in several languages, and afterwards, let them compare the solutions using a table with several comparison criteria, such as encapsulation, polymorphism, inheritance, memory management, syntax, etc.

UNDERSTANDING ABSTRACTION

This pattern is a refactoring of Gary L. Craig's *Discussion-Activity-Review-Lab-Review* [GLC] pattern, refactored by Jane Chandler.

Concepts that must be understood at two levels of abstraction require time for an iterative approach to learning. However this can be time consuming.

❊❊❊

Therefore, introduce a concept at its highest level of abstraction and use concrete, practical examples or exercises together with reflection on the concept to link the higher level abstraction to the lower level abstraction.

When designing examples or exercises relate the abstract concept to students' concrete experiences (see SOLUTION BEFORE ABSTRACTION) and ensure that students see a number of examples of the concept (as in ONE CONCEPT – SEVERAL IMPLEMENTATIONS).

Consider carefully which of the abstraction levels of the concept to emphasize, e.g., the higher or lower level or the transformation process between the levels of abstraction, as this will drive the level of detail required in each activity.

A consequence of this pattern is that students appreciate the links between concept's different levels of abstraction.

❊❊❊

For example, begin with a class-wide discussion of the concept at its highest level of abstraction without focusing on the details. This will support understanding the big picture. Follow this with small group exercises based around specific (detailed) issues. Next, review the results of the exercises with the whole class, paying particular attention to both common and alternative solutions. Use a second set of exercises to enable the students to produce the lower level of abstraction for themselves. Where appropriate the lower level abstraction, at least in part, should be illustrated through a transformation. Finally, evaluate the results of the second set of exercises and reflect on the connections between the abstraction levels.

This pattern can be used in analysis & design or design & coding courses to show the relationship between the two phases and provide a context for the decisions made. For example, introduce the design issues in an initial, class-wide discussion and follow this with the students undertaking a design exercise. Students then have a sense of "ownership" of the problem and subsequent design and from this position they can then be asked to code the design and finally to review and evaluate their code in relation to their design.

SEE BEFORE HEAR

This pattern is a refactoring of Mary Lynn Manns' *Lab-Discussion-Lecture-Lab* Pattern [MLM], refactored by Mary Lynn Manns.

Learners often find it difficult to convert what they've heard in the classroom into skills they can use outside the classroom. They will remember less of what they hear than what they see and experience. However, in the typical and quite practical classroom lecture format, instructors are often heard saying such things such a "<*this*> *is what will happen when you do* <*this*>". But, a "hear before see" approach is quite abstract, and can make it difficult for the learner to later make use of the concepts in the lecture.

❊❊❊

Therefore, give learners the opportunity to *see* and experience a new concept before they *hear* about it. Encourage learners to record, and to reflect upon, what happened when they are involved in the learning.

❊❊❊

For example:

See: *Give learners the resources to complete a hands-on lab exercise with detailed step-by-step instructions and references to documentation where clarification may be obtained. Include questions throughout to encourage learners to record and analyze their experiences. Periodically allow time for a learner-centered discussion of unfamiliar concepts and problems encountered along the way.*

Hear: *Following the "see" experience, a more traditional "hear" lecture can be delivered to solidify the new concepts that were introduced during the lab. References should be made to the experiences the learners just had during the "see" phase.*

Learners become actively involved in their learning because they are introduced to new concepts as they are using them. Instructors can give less abstract lectures since learners will have had experience with the concepts before a lecture session.

Because of the effort required to develop the hands-on lab for the "see" experience, the prep time can initially be tedious for the instructor. However, the increased level of student comprehension that this approach provides seems to decrease the necessity for extensive follow-up and review periods.

An optional follow-up exercise can be given in the form of a more complex lab that reinforces and tests each learner's understanding of the new concepts. It can then be evaluated by the instructor.

Mistake Pattern *and* Toy Box *[JBx] are* See Before Hear *patterns.*

SOLUTION BEFORE ABSTRACTION

This pattern is a refactoring of Ian Chai's Concrete to Abstraction [IC] pattern, refactored by Klaus Marquardt.

In a typical classroom situation, students may not know what benefit they might derive from the topic. There is the need to keep students' interest even in abstract concepts.

An abstract concept can become the basis for a large number of applications. However, it is hardly considered useful unless it is related to concrete experience.

❊❊❊

Therefore, give the students an example of the problem in a setting that they are comfortable with. After they have found a solution for this example, focus their attention on those aspects that can be applied to similar problems. When your students are inexperienced or you feel that the subject matter is very complex, you should introduce more than one concrete example (see ONE CONCEPT – SEVERAL IMPLEMENTATIONS).

Use the identified transferable aspects to introduce the general, abstract concept of the solution. When your students have understood the underlying principle, you can advance to a more formal description such as abstractions or patterns

This kind of presentation is especially useful for students with little or no experience in the course area. It assumes that students are not familiar with the concept with respect to their profession, so that they need to learn a relation that more experienced professionals probably already discovered themselves. After some abstractions are introduced this way, the teacher may change the presentation form and start with abstractions before applying it to example situations.

❊❊❊

For example, real life experiences can be used to introduce abstract concepts. When two persons have no language in common, and they do not want to learn another language, they need a translator. Between two existing software systems that do not understand each other, you need a component taking a role similar to a translator. This analogy to a real life experience helps to introduce the concept of the Adapter pattern that allows establishing contact to a different program without the need to change it.

EXPERIENCING IN THE TINY, SMALL AND LARGE

This pattern is a refactoring of Billy B.L. Lim's Programming in the Tiny, Small, Large [BL] pattern, refactored by Jutta Eckstein.

Some topics require a number of iterations in various depth to enforce learning by repetition.

A complex concept is difficult to understand unless you have experienced it by example. However concepts are often so complex that experiencing the whole in one step doesn't help either.

❊❊❊

Therefore, introduce the concept in three stages, tiny, small and large, which allow you to monitor the students' progress on a topic-by-topic – *tiny* – basis, to test if the student can combine the topics and apply them in a larger – *small* – setting and to solve a real-world – *large* – problem using all parts of the concept, thus seeing the *big picture* respectively.

Provide a smooth way for the students to get a start into the topic. This is best done by General Concepts First [VF]. Make sure that each of the stages doesn't force the students to make a large jump (as in Digestible Package [VF]). And finally don't forget to give immediate Feedback [VF] to the students' experiences. You could also first let the students experience the concepts as in SEE BEFORE HEAR. If you want to emphasize the iterative part of the pattern more, you can use Spiral [JB] as the context.

As a consequence, students can grasp abstract concepts early in the course through tiny assignments before they embark on a more challenging one. Provide the students the possibility to see the *big picture* without loosing track of the details by working on the individual pieces and combining those pieces together.

❊❊❊

When you teach the basics of object-oriented design concepts assign to the students one tiny *concept, e.g. a single class for which the students should model the responsibilities. The next step would be to let them develop a* small *design, consisting of e.g. three classes, so they will have to focus also on collaborations. The final* large *assignment would then be to design a whole system, where e.g. Design Patterns should be considered. The time frames for the different stages depend on the whole course length. When you are teaching a one week course in industry, the tiny assignment may take only a few minutes, the small assignment less than an hour whereas the large assignment will run throughout the whole week.*

BUILT-IN FAILURE

This pattern is a refactoring of Kent Beck's *Three Bears* Pattern [KB], refactored by Eugene Wallingford.

Learning comes from experience, and much useful experience comes from failure. But a learner who lacks confidence will fear failure, and this fear impedes or even prevents learning.

Confident learners use failure and frustration as investments whose payoff comes in future success. They know that a „wrong answer" offers the opportunity to discover a misunderstanding and to arrive at a better understanding of the topic. These knowledge „repairs" will lead to improved performance over time.

However, many learners do not start out as confident. Traditional schooling typically discourages or punishes failure through grading schemes and recognition of academic achievement. Employees may fear that failure will be seen as a sign of inability by their employers and lead to fewer workplace rewards. As a result, the confidence to fail is rare and hard to develop.

❊❊❊

Therefore, remove the fear of failure as a barrier to learning by making failure a part of the goal.

Create an environment in which failure is an expected and desired outcome of the learning activity. Build an activity that requires learners to reflect on both the „correct" and „incorrect"

answers as a way to better understand the topic. Make sure that all learners will encounter the negative outcome and that no one will be stigmatized by not reaching the right answer.

❊❊❊

THREE BEARS builds failure into the process of learning to recognize a point along a continuum. MISSION IMPOSSIBLE poses a problem that cannot be solved with a naive understanding of the topic, encouraging students to explore the topic more deeply. Mistake [JBx] asks students directly to make and deal with errors.

THREE BEARS[1]

This pattern is a refactoring of Kent Beck's *Three Bears* Pattern [KB], refactored by Eugene Wallingford.

Some problems, inherently sap a learner's confidence. Many problems challenge the learner to find a solution positioned along some continuum. Solving these problems effectively requires that the learner discover a point or a range along the continuum that satisfies the demands of the problem. But finding such a solution requires that the learner have experience with many problems, balancing the demands of each in a particular solution. Until they have sufficient experience, they are likely to be unsuccessful finding the right balance.

How often should a developer refactor a program? How strictly should a musician follow the rhythm of the piece? How often should a point guard shoot the basket-ball?

The process of learning to find such balances creates substantial barriers to the learner gaining experience. The learner will likely be unsuccessful on the first few attempts, unlike many other learning activities. Even if the learner stumbles into the right balance, chances are that the learner will not recognize that the balance has been struck, or why.

❊❊❊

Therefore, ask the learner to create solutions that lie at both extremes, as well as at some balance point. The extreme answers will certainly be ‚wrong' for the given problem, but they give the learner permission to explore the boundaries of the continuum.

First, define the continuum for the learner. The simplest approach is to explain the *reductio ad absurdum* at both extremes.

Second, conduct an experiment that gives the learner a chance to locate the balance for problems whose solutions lie in three different places: at one end of the continuum, somewhere in the middle, and at the other end of the continuum.

Third, conduct a review that gives the learner an opportunity to reflect on the experiment.

The reductio ad absurdum strategy usually gives the learner enough background to begin learning the continuum. You might also pose a set of questions that will be asked of the resulting balance. For example, in reviewing the frequency of refactoring, you could ask ‚Was the team able to get into a good flow while programming?', ‚Was there sufficient time for testing?', and ‚Did the team deliver its product?'

[1] THREE BEARS derives from a German fairy tale in which a little girl encounters a number of situations in which her three choices are 'too hot, too cold, just right', 'too hard, too soft, just right', and so on.

Your experiment should ensure that the learner experiences all three options close enough in time to accurately compare them. Scope the topic to something that can be accomplished in less than an hour, if possible.

Reviewing the experiment is critical if the learner is to understand how well the solutions balance the problem's demands. A useful technique is to have the learner briefly present the three solutions to other learners, and then have the rest of the group guess which was which. This can help learners who have not yet learned the true boundaries of the continuum.

Even still, some learners have difficulty getting past the fact that they 'have to do it right' eventually.

Some topics are more complex. You may find that reducing a problem to a single continuum oversimplifies the topic so much that the learner arrives at a simplistic understanding. In such cases, you will want to follow up this experience with others that address the problem's other facets.

❊❊❊

Walt Disney once used this technique when he found himself dissatisfied with the features of his staff's animations. Finally, in frustration, he told the animators to exaggerate the movement of all their characters. The result was just what Disney was looking for. [Thomas1981]

THREE BEARS has been used to teach requirements engineering. The instructor asks the learners to write stories that will define the system: one too large in scope to be useful, one too small, and one just right.

Similarly, THREE BEARS can help learners to explore the ethical continuum that faces computing professionals. Groups of three are asked to write stories about obviously ethical applications of computing technology, obviously unethical applications, and applications that are still unsettled. Later, the groups share their stories with each other and try to place the stories on the continuum. Interesting discussions usually follow as the groups disagree with one another about the relative placement of their stories.

Many experienced learners routinely use THREE BEARS in their own learning all the time. For example, a Smalltalk programmer might learn the constraint-ish ValueModel framework by deliberately writing systems that use it too much. Some programmers learn object-oriented programming by writing some programs in which every variable is an instance of a different class and other programs that use too few classes and objects.

MISSION IMPOSSIBLE

This pattern is a refactoring of Alan O'Callaghan's [AOC], refactored by Eugene Wallingford.

Many teaching situations are limited in duration. The instructor can choose to omit important concepts in order to fit the available time, but then the learners will not have been exposed to the full subject. Alternatively, the instructor can choose to generalize the material to the point that the whole subject can be covered in the available time. However, such generalizations can oversimplify a rich, subtle topic to the point that the learners think they have mastered it, even though they do not yet have sufficient experience with its details.

Any sufficiently complex topic can be understood at many levels of abstraction. When a generalization is supported by understanding at deeper levels, then abstraction can be a powerful tool. But often new learners arrive at an abstraction not via generalization from a deeper understanding but from a simplification of something they do not yet understand. Such simplistic truths are dangerous, because they lead learners to construct simplistic

solutions that do not really solve problems. Worse, the learners‘ lack of experience prevents them from recognizing the shortcomings in their thinking.

❊❊❊

Therefore, present the learner with a problem that seems straightforward to solve but whose complete solution requires a much deeper understanding than the basic concepts afford.

Choose a problem that at first glance suggests a solution based directly on the general concepts that the learners have encountered. However, a complete solution to the problem should require careful consideration of a number of issues. Indeed, make the development of a complete solution not normally be possible due to insufficient time to study the full range of issue, insufficient information available to the learner, or the lack of any solution at all, despite first impressions.

Follow up the exercise with a brief summary that explains why the problem was *impossible*.

The contrast between the learner‘s initial reaction (‚This is easy!‘) and the result of some study (‚This is a more difficult problem than first we thought!‘) is crucial to the success of this pattern. It creates in the learner a recognition that the subject is more subtle than originally thought. The instructor‘s explanation at the end should make sure that the learner understands both the impossibility of the problem and the role played by their still naive understanding in not seeing it.

Use this technique just after the learner has conquered a logical unit of material. It can be used to form a link between the learning of basic concepts and the more advanced topics needed to master the ‚impossibility‘ of the problem.

You should be able to present the problem in a short form, and yet it should be complete enough that the learner has sufficient information to begin work. The learner should be able to appreciate the unforeseen subtlety of the problem within about 45 minutes, or the learner will begin to lose interest in the problem.

MISSION IMPOSSIBLE makes learners suspicious about their understanding of basic concepts so that they continually question those concepts and improve their understanding of them. Learners occasionally need to be ‚shocked‘ into deeper thinking about what they are doing in order to appreciate subtleties. This becomes even more important when such ideas as ‚objects model the real world‘ can be understood in a naive way that disarms the learner in the face of real problems.

Misused, or overused, the pattern can destroy a learner’s confidence in what she is learning. Some learning contexts create unstated expectations that the student that they will be ‚spoon-fed‘ instruction. Many university students come from schools in which rote learning is the norm. In industry, new ideas are often viewed only as tools or as programming techniques and so require ‚instruction‘, not ‚education‘. MISSION IMPOSSIBLE requires initiative and risk-taking on behalf of the learner, and therefore may not be appropriate in such contexts.

This pattern follows General Concepts First [VF] and is a form of Repeat Topics [VF] that aims for a deeper understanding of the repeated topics.

❊❊❊

In historic tradition, Zen masters pose questions such as ‚What is the sound of one hand clapping?‘ in order to encourage their students to lose their worldly inhibitions and achieve enlightenment.

MISSION IMPOSSIBLE has been used in teaching of object-oriented concepts both to university students and to software professionals. Often, the basic truths about objects that distinguish them from structured methods are expressed in a way that leads the learner to underestimate the intellectual effort needed to master object-oriented concepts. A common example is Meyer's ‚Objects are there for the picking' in response to the question, ‚Where do I find the objects?') Such a situation creates a perfect opportunity to apply this pattern.

For example, a one-hour tutorial [AOC2] uses MISSION IMPOSSIBLE to help learners realize that even relatively simple programs becomes tortuous when they apply a naive notion of object in both analysis and design. This tutorial uses a simple scenario from [Cook 1994] to help the learners see the need for transformation from the objects in the analysis model to those in the design model.

EXPOSE THE PROCESS

This pattern is a refactoring of Byron Weber-Becker's *Expose The Process* Pattern [BWB], refactored by Markus Voelter.

Examples and exercises form a vital part of any teaching effort. However, often examples and (correct solutions to) exercises only show the final result. The process of getting there, including the necessary decisions, dead-ends and backtrackings, alternatives and principles is not obvious. As a result, students get frustrated because they do not find the same solution, or simply do not know how to approach the problem.

❊❊❊

Therefore, when showing examples or "ideal" solutions to exercises, also show and explain the process of getting there. Show the critical decision points to the students and allow them to make their own proposals on how to go on. When asking them to do an exercise, ask them to also document alternative solutions, and why they do not lead to the desired result. When discussing the exercise in class, let the students show and discuss alternative solutions.

Be sure to honor "silly questions" (as in Joe Bergin's Gold Stars for Confusion [JB3]) and honor the students work on a topic, even if the correct solution one was not found (as proposed in Exercise Emphasize Process by Fricke/Voelter in [VF]).

This pattern takes time. You have to reduce the amount of what you cover in a session, however, the things still covered will be more thoroughly understood.

Do not use the pattern when introducing a new topic. It is not very productive to let the students struggle to find a solution without suitable tools – and then showing an elegant solution using a new technique. The pattern works best during a consolidation phase, when students are practicing formerly introduced topics and learning how best to apply them.

Also make sure that you don't bore students. If they are all so good that they see the right solution from the beginning, don't force them to care about other solutions, which they would consider in practice.

❊❊❊

There are several ways to implement this pattern. One is to go into a lecture/seminar with a problem, but without a solution, trying to find the solution together with the participants. This is a bit risky, in case you do not find the solution... Alternatively, you can try to find the solution offline before, and write down all decision points, etc.

You should try to involve students. But be careful: If the good students always suggest the "ideal" solution first, you have to suggest worse solutions to get the variety. This is not ideal, therefore start by showing weaker solution first, then asking the students for improvements.

The pattern has been used in the introductory programming course at the University of Waterloo (Waterloo, Ontario, Canada), and in the books An Introduction to Computer Science using Java *by Kamin et. al. [KMR] and* Designing Pascal Solutions: A Case Study Approach *by Clancy et al. [CLF].*

STUDENT DESIGN SPRINT

This pattern is a refactoring of Joseph Bergin's *Student Design Sprint* [JB2], refactored by Joseph Bergin.

Students need to practice design at all levels. They also need quick feedback and peer review on early attempts. Most educators recognize now that students need to be exposed to design early. Most also recognize the need for team work and for critical analysis. We eventually need to teach system design, but beginners need program design as well. If we don't teach it then students will develop their own ad-hoc techniques that may reinforce bad habits. If you use a Spiral [JB] approach the elements of simple design should come in the first cycle.

❊❊❊

Therefore, use some variation of the following highly structured activity. This activity can take place in a seminar, classroom, or in a lab.

Divide the students into groups of two (or three). Give them a design problem and ask the teams to produce a design outline in 15-20 minutes. There should be a written sketch of the design in that time, perhaps with CRC cards if it is an object design. The instructor can look over shoulders and comment or not, but few hints should be given. Questions should be answered freely.

At the end of 15-20 minutes, the instructor poses a set of questions about the designs without asking for answers. The questions should be such that they cannot be favorably answered by some set of poor designs.

The students are then regrouped by combining pairs of nearby groups, so that you now have groups of 4 or five students and each group has two of the original designs. The task is now modified slightly and the groups are asked to produce a new design.

After another 15-20 minutes the instructor again poses a set of questions for thought, regroups the students again into still larger groups, modifies the task slightly and again puts the students to work.

This can continue for as many cycles as the instructor wishes. At the end, the instructor should evaluate the resulting designs and make comments. It may be enough to show one or two of the best designs and explain why these are better than the others. If poor designs are also to be shown, it might be best if the names of the designers are not attached.

Alternatively, the groups can be required to present and justify their designs and the rest of the class can critique them.

For some situations one cycle may be all that is needed, followed by a discussion of the issues. In this case the instructor can ask the groups which designs had certain characteristics.

❊❊❊

Alistair Cockburn [AC] has a wonderful exercise for students designing a coffee machine in about three or four cycles in which the requirements become more sophisticated each cycle. In the first cycle the machine can deliver coffee for 35 cents. In the second it can also deliver soup for 25 cents.

This can be used in program design in the early phases of a student's learning. The task can be to write a function with a given set of pre and post conditions. The tasks in the later cycles can be to tighten the pre conditions and/or strengthen the post conditions.

Alternatively, the task could be to develop some code with a given invariant and the questions can involve ways that the invariant might be invalidated by a user if the design is not sound.

This pattern can be used when learning data structure design. For example, the students can be asked to design a linked list, without telling them how it will be used. They must design a protocol and pick an implementation strategy. The instructor can then suggest some uses to which a linked list might be put and ask if the design supports that use.

ROUND ROBIN

This pattern is a refactoring of Kent Beck and David Bellin's Round Robin [BB] pattern, refactored by Joseph Bergin.

One of the most difficult aspects of team work is getting everyone in the room to work on equal footing. Both organizational differences (jobs, position, etc.) and personality can quickly and inadvertently lead to a core of speakers and a core of listeners. Moreover, the fact that the listeners are not talking does not mean they are not thinking or that they are in agreement. However, you want to get everyone's participation and input and you especially want to encourage the quieter members to take a more active role.

❊❊❊

Therefore, use a round robin technique to solicit suggestions.

Go around the room or table. As each member of the team contributes an idea, write it down on the board. The facilitator should do the writing since the other members of the team should be watching and thinking. If there is a team member who does not have enough information on a particular problem to contribute to the brainstorming, that person can act as scribe to keep them involved. However, if a team is chosen well, every member should be an important source of ideas. The goal of the round robin is to allow the group to move ahead at an even tempo but to give people enough time to think. Short pauses are fine, but breaks of more than 60 seconds can interrupt the momentum and ideas may be lost. To keep things going you can establish a "pass" policy. If someone is really stumped, they can "pass" for that round, but they should take their regular turn the next time around. The facilitator needs to be sensitive here. If someone is slower to speak, don't cut off their turn too soon. At the same time, keep things moving so that other people do not forget what they want to say. The brainstorming is complete when everyone in the group has to pass.

However, this works better in small groups (6 or so) than large (20). You may need to factor out a subset of the group to use this effectively. Or you can partition the large group into smaller groups and use this in each group. To do that requires a facilitator in each group.

❊❊❊

For example, you can use this in coming up with suggestions for the initial CRC cards for a class design of a new problem being considered by the class. You can also use it to get comments on the flaws in a suggested design. You can also use this in any Brainstorming [DB] session.

References

AC	Alistair Cockburn, *Website,* http://members.aol.com/acockburn/
AOC	Alan O'Callaghan, *Mission Impossible,* http://sol.info.unlp.edu.ar/ppp/pp21.htm
AOC2	Alan O'Callaghan, *The Topsy-Turvy Worlds we live and work in – and the different ways we need to use objects in each of them*, Proceedings of the Educators' Symposium, OOPSLA 1998
BB	Kent Beck and David Bellin, *Round Robin,* http://sol.info.unlp.edu.ar/ppp/pp6.htm
BL	Billy B.L.Lim, *Programming in the Tiny, Small, Large,* http://sol.info.unlp.edu.ar/ppp/pp10.htm
BWB	Byron Weber-Becker, *Expose the Process,* http://www.soi.city.ac.uk/~hsharp/OopslaPATS.htm
CA	Christopher Alexander et.al., *A Pattern Language: Towns – Buildings – Construction*. Oxford University Press 1977
Cook 1994	S. Cook, J. Daniels, *Designing Object Systems*, Prentice Hall 1994
CLF	Clancy, Linn, Freeman, *Designing Pascal Solutions: A Case Study Approach,* 1992
DB	David Bellin, *Brainstorming,* http://sol.info.unlp.edu.ar/ppp/pp4.htm
GLC	Gary L. Craig, *Discussion-Activity-Review-Lab-Review,* http://sol.info.unlp.edu.ar/ppp/pp18.htm
GoF	Gamma, Helm, Johnson, Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley 1995
IC	Ian Chai, *Concrete to Abstraction,* http://sol.info.unlp.edu.ar/ppp/pp1.htm. This references the patterns "Acquaintance Examples" and "Colorful Analogy" from Dana Anthony, Patterns for Classroom Education, Proceedings of PloP'95
JB	Joseph Bergin, *Spiral,* http://csis.pace.edu/ppp/pp32.htm
JB2	Joseph Bergin, *Student Design Sprint,* http://sol.info.unlp.edu.ar/ppp/pp60.htm

JB3	Joseph Bergin, *Gold Stars for Confusion*, http://csis.pace.edu/ppp/pp58.htm
JBx	Joseph Bergin, *Mistake, Toy Box*, http://csis.pace.edu/~bergin/PedPat1.3.html
KB	Kent Beck, *The Three Bears*, http://sol.info.unlp.edu.ar/ppp/pp17.htm
KMR	Kamin, Mickunas, Reingold *An Introduction to Computer Science using Java*, McGraw-Hill, 1998
MJ	Marcelo Jenkins, *Design-Do-Redo-Redo pattern*, http://sol.info.unlp.edu.ar/ppp/pp13.htm
MLM	Mary Lynn Manns, *See before Hear*, http://sol.info.unlp.edu.ar/ppp/pp60.htm
PPP	Pedagogical Patterns Project, http://www.pedagogicalpatterns.org
PLOP1	Coplien, Schmidt (eds.), *Pattern Languages of Program Design*, Addison Wesley 1995
PLOP2	Vlissides, Coplien, Kerth (eds.), *Pattern Languages of Program Design 2*, Addison Wesley 1996
PLOP3	Martin, Riehle, Buschmann (eds.), *Pattern Languages of Program Design 3*, Addison Wesley 1998
PLOP4	Harrison, Foote, Rohnert (eds.), *Pattern Languages of Program Design 4*, Addison Wesley 2000
Thomas 1981	Frank Thomas, Ollie Johnston, *Disney Animation: The Illusion of Life* Abbeville Press, New York 1981
VF	Markus Voelter, Astrid Fricke, *SEMINARS*, http://www.voelter.de/seminars

Authors

Joseph Bergin	can be reached at berginf@pace.edu
Jane Chandler	can be reached at jane.chandler@port.ac.uk
Jutta Eckstein	can be reached at jeckstein@acm.org
Mary Lynn Manns	can be reached at manns@unca.edu
Klaus Marquardt	can be reached at marquardt@acm.org
Markus Völter	can be reached at voelter@acm.org
Eugene Wallingford	can be reached at wallingf@cs.uni.edu

Acknowledgements

Many thanks are due to Martine Devos, the EuroPLoP shepherd for this pattern collection, who had to struggle with a whole committee of authors and editors. Special thanks to Jutta Eckstein who initiated this pattern refactoring effort and convinced everybody that this work was useful and necessary.