Fourteen Pedagogical Patterns

Joseph Bergin
Pace University

One Pace Plaza
New York, NY 10038 USA

jbergin@pace.edu
http://csis.pace.edu/~bergin

The following fourteen patterns form the beginning approaches to a pattern language for Computer Science course development. They might have application to other fields as well. The patterns are not all at the same level of scale. Some speak to the overall course organization and some to very low level things. The general flow is from large structure (semester courses) to small scale (daily activities). A long term goal is to develop them into a proper language. This will require supplementing them with others as well.

Some of the patterns have numbers assigned. These have been submitted to the Pedagogical Patterns Project: http://www-lifia.info.unlp.edu.ar/ppp/. The numbers are assigned by the project. Many educators are getting involved in this effort. You can find the current list of patterns and more information at this site. The others have been submitted but have either not been accepted, or have not had numbers assigned. You might consider getting involved in this effort if you are an educator.

At the end of the list of patterns I discuss how some of these might eventually fit together into a pattern language.

The skeleton form of these patterns is prescribed by the Pedagogical Patterns Project.

Last updated: July 31, 2000 3:26 PM

Pedagogical Pattern #34

Early Bird

(Version 2.1, July 2000)

Organize the course so that the most important topics are taught first. Teach the most important material, the "big ideas," first (and often). When this seems impossible, teach the most important material as early as possible.

PROBLEM/ ISSUE

The typical course has many important topics. Many times they are interrelated. It is difficult to decide how to order topics so that students will appreciate the "big ideas" in the course. If you delay important topics until late in the course, spending much time on preliminaries, students may get the wrong idea about relative importance. You will also not be able to reinforce the big ideas frequently through follow up exercises and discussions.

AUDIENCE/ CONTEXT

This has very wide applicability to almost every domain.

FORCES

Students need to see where they are headed. They need to see that detail presented early in the course will relate to important ideas.

Students need to know what are the few Big Ideas from each course. They need to be able to separate the key concepts from the detail that support them.

Students often remember best what they learn first. This can be both positive and negative, of course. Important (big) ideas can be introduced early, even if they can't get complete treatment immediately.

SOLUTION

First identifiy the most inportant ideas in the course. "Mine" the course for its most important ideas. These ideas become the fundamental organizational principle of the course. Introduce these big ideas, and especially their relationships at the beginning of the course and return to them repeatedly throughout the course.

Here we order class topics in order of importance and find ways to teach the most important ideas early.

If design is more important than programming, then find a way to do design as early as you can. If functions are more important than if-statements in programming then do them first. If objects are more important than functions, then do them first.

DISCUSSION/ CONSEQUENCES/ IMPLEMENTATION

The most important things in a course or curriculum receive more focus from the instructor and the students. Students can be made more aware of what is paramount.

Implementation is difficult. Often only simple aspects of an important idea can be introduced early. Sometimes it is enough to give important terms and general ideas. Some "big" ideas are thought of as advanced. It is difficult to introduce some of these early. Hard thought and preparation are needed in curricular design. Sometimes a really big, but difficult, concept can be introduced incompletely. Then as other material that relates to it is covered, the relationship to the big idea is carefully explored.

However, if you can't introduce the big idea at the beginning, then make certain that nothing you do early is inconsistent with the big idea or impedes its easy learning. For example, if you teach procedural decomposition early in a course, it will impede the learning of object-orientation. This is because the thought processes that lead you to a procedural decomposition are different (and inconsistent with) the ideas of OO.

You need to be able to analyze deeply what are the consequences of developing material in a particular order. It is often helpful here to have a forum in which ideas can be discussed and refined. It is also often necessary to develop your own materials, which requires time and effort.

SPECIAL RESOURCES

Time and deep thought are clearly required. Discussion groups with other educators who share similar ideas about the most important concepts in a domain are very helpful.

RELATED PATTERNS

It may be necessary to *Spiral* to give some needed background on the important topics.

A Lay of the Land example can be used to show the students an example of a big idea in action. If there are many important ideas it can be *Larger Than Life*.

A *Fixer Upper* can be a good way to get started. It must emphasize the big idea of course.

If the idea is complex, use a *Toy Box* example to introduce it.

Interrelated ideas can often lead to components of a *Tool Box*.

Note: This pattern is actually recursive, as patterns themselves are a really big idea.

EXAMPLE INSTANCES

Teaching objects first (or at least early). Teaching design first. Teaching concurrency first in operating systems. Teaching user requirements first in Database. Teach recursion before loops. Of course, these are my definitions of what is most important. You may disagree, but then it is your course, so discover and implement your own "firsts."

The book Karel the Robot, by Richard Pattis was designed with this pattern in mind as a way of teaching procedural programming (procedures first). Its successor, Karel++, attempts to do the same with Objects (classes first).

CONTRAINDICATIONS

It may be a mistake to try to use this pattern with material that has clear prerequisite ideas to the important ideas. This would be especially true if the relationship between the prerequisite idea and the big idea is especially subtle or if the prerequisites are especially difficult to master. Then again a clever use of *Toy Box* or *Lay of the Land* might let you do what seems difficult in presenting topics early.

REFERENCES

Karel the Robot, Richard Pattis, Wiley, 1981

Karel++, Joseph Bergin, Mark Stehlik, James Roberts, Richard Pattis, Wiley, 1997.

Pedagogical Pattern #32

Spiral

(Version 2.1, July 2000)

Topics in a course are divided up into fragments and the fragments introduced in an order that facilitates student problem solving. Many of the fragments introduce a topic, but do not cover it in detail. Just enough detail is given initially so as to form a basic understanding that can be applied to problem solving. Additional cycles contain reinforcing fragments that go into more detail on the topic.

PROBLEM/ ISSUE

Topics in a course are often interrelated. Too often lots of different topics are required for students to have enough tools with which to solve interesting problems. If we try to do the topics in any "logical" order we tend to get bogged down in details and leave the students bored. Students need to be empowered to solve meaningful problems early in the course.

AUDIENCE/ CONTEXT

Any course in which there are a large number of concepts that must be mastered together.

This pattern can be used in several courses, primarily at the early stages of the curriculum. It can be used (at least) in programming courses, analysis and design courses, and special courses in object technology. A variation of this pattern is often used in the compiler construction course.

FORCES

Many fields can only be mastered by individually mastering a large number of different techniques that must interact.

Large topics such as programming and design require many parts and much detail to master. Developing these in a sequential manner leaves the students without interesting exercises, as they have not seen enough of the breadth of the topic to do interesting things.

Students like to build things and they like to see how the pieces fit together. They get bored easily if instruction is repetitive and if the instructor spends too much time on one topic or a set of closely related topics. Students can also get bored if exercises are artificially contrived to illustrate arcane details.

Courses do not need to be organized like reference material. Nor should textbooks be.

SOLUTION

Organize the course to introduce topics to students without covering them completely at first viewing so that a number of topics can be introduced early and then used. This can get students working on interesting problems earlier as they have more tools to use, though they have not, perhaps, mastered any of the tools. The instructor can then return to each topic in turn, perhaps repeatedly, giving more of the information needed to master them.

On each cycle of the spiral topics are covered in more depth and additional topics are included. The sequencing of the "fragments" is done with an eye to providing students with problem solving skills. Anthony's *Mix New and Old* suggests the important of mixing new material into what is already known.

The course cycles around to a given topic several times during the term, each time to a greater depth.

DISCUSSION/ CONSEQUENCES/ IMPLEMENTATION

This pattern results in the topics of a course being more fine grained, with just enough of a larger topic introduced at each stage to permit problem solving with other tools/topics which are also, as yet, not completely covered. For example, "iteration" is a big topic. The "while loop" is a small topic, especially if initially introduced with only simple loop tests. Yet the while loop can be used effectively with other constructs to build meaningful programs before iteration is understood in all of its aspects.

To start, the instructor should extract a subset of the material covering several topics that interact. Only simple cases should be introduced at first. The instructor and class move quickly through the topics until an understanding of how the topics interact can be gained. Students then can work with the tool subset on problems. Then more of each of the original topics, with perhaps simple cases of new topics are introduced to deepen understanding of the topics and of their interactions. Students then work on a richer set of problems. This can be repeated as often as necessary.

Students will get a feel earlier for how the pieces fit together. A potential negative consequence for some students, at least, is that some of their questions (What if...?) may need to be deferred (see *Test Tube*).

SPECIAL RESOURCES

The instructor needs a plan, showing the order in which the topics will be introduced and what will be deferred to later cycles.

The instructor must extract subsets of each of the many topics in which the tools introduced can work together in problem solving. Several, increasingly large subsets must be designed. Problems using most of the features of each subset need to be designed. One way to design the subsets is to start with the problem and extract a minimal set of tools necessary to solve that problem and similar problems. The next larger subset can often be designed by thinking about how the original problem could be expanded and its solution generalized.

RELATED PATTERNS

Early Bird can get you started on the first cycle.

Test Tube can be used to avoid getting bogged down.

Toy Box can be used to provide a sequence of increasingly difficult exercises.

Lay of the Land and Larger Than Life can be used to provide an overall vision.

Fixer Upper can be used to introduce new material for each cycle.

This pattern can make Early Bird and Lay of the Land work well together.

Dana Anthony has several patterns that inform this one. In particular, *Mix New and Old, Seven Parts, Example Lasts One Week*, and *Visible Checklist* can be used to design each cycle around the Spiral.

EXAMPLE INSTANCES

This pattern can be used to teach low level programming structure. For example integer data, assignment statements and simple forms of if and while can be introduced. Problem solving using these topics only can be introduced allowing students to solve simple programming problems. In the second cycle, more can be discussed on each of these topics (else clauses, infinite loops,...).

In analysis and design, simple analysis techniques and tools can be introduced (simple use-case) and then the class can move to simple designs (CRC cards). Students can thus get a feel for the whole process, solving simple problems. The second cycle can introduce simple features of more sophisticated tools as well as somewhat more complex problems.

In object technology courses, simple inheritance can be introduced and used before polymorphism (dynamic binding) is discussed.

CONTRAINDICATIONS

This pattern cannot be used in a small way. A commitment needs to be made to it. If this is not possible or desirable, avoid it entirely.

REFERENCES

The book *Ten Statement Fortran Plus Fortran IV* by Michael Kennedy took a spiral approach.

Karel the Robot, by Richard Pattis (Wiley, 1981) was a first cycle in a Spiral approach to Pascal. Its successor, *Karel++,* by Bergin, Stehlik, Roberts, and Pattis (Wiley, 1997) tries to do the same for an object-oriented language like C++ or Java.

Dana Anthony's patterns were presented in PLoP '95.

See http://st-www.cs.uiuc.edu/~chai/writing/classroom-ed.html

Pedagogical Pattern #

Consistent Metaphor

(Version 2.1, July 2000)

When teaching a complex topic outside student's normal experience, find a complex and consistent metaphor for the topic being taught. The basis of the metaphor needs to be known to the students.

PROBLEM/ ISSUE

Especially when teaching beginning students, it is easy to get lost in the details of the current topic. Students then may not see how this topic is related to larger goals. It is also difficult for many students to see quickly how things fit together and make correct predictions about how the technology "should" behave.

AUDIENCE/ CONTEXT

You are teaching a complex topic, such as object-oriented thinking. The topic has many parts, some of which are quite detailed. The students need a way to think of the topic as a whole, but the topic is highly technical and outside their experience.

FORCES

You want to give your students a powerful and consistent shorthand for thinking about some complex topic. The shorthand should relate the topic being taught to things within their experience.

Students may get lost in the detail easily and fail to see the big picture and how the parts relate to each other. This is especially true when the details themselves are unfamiliar and new to them.

It is helpful when learning new topics to relate new ideas to already familiar ideas.

It may take you a long time to teach all of the elements of the topic under review. (But, see *Spiral*.)

You want students to have an idea about what is happening within a system that permits them to make valid inferences about what should happen.

SOLUTION

Create a metaphor that is consistent with the topic being taught, and with the same basic elements that interact in the same way. Give this to the students as a way to think about the topic. The metaphor must allow students tomake valid inferences about the topic by thinking about the metaphor.

DISCUSSION/ CONSEQUENCES/ IMPLEMENTATION

The instructor must know the limits of the metaphor and communicate these to the students, so that they don't make improper inferences.

A metaphor can be used for a small element of a topic or to give a view of the overall landscape. It is most useful when valid inferences can be drawn from it.

The basis or the metaphor itself must be well known to the students.

SPECIAL RESOURCES

Discussion groups are useful in exploring metaphors among educators and for finding the limits of particular metaphors.

RELATED PATTERNS

Audio Object Analogies (#31) is a specialization of this pattern.

Role Playing (#5) makes use of an instance of this pattern.

EXAMPLE INSTANCES

(Human Powered Vehicle) One of the most powerful ways to teach about objects and their properties is to imagine that each object in a system is a person. The metaphor shares the following with an object system:

a. People and objects are autonomous actors.

b. People and objects respond to messages.

c. How a person or an object handles a message depends on the receiver and not on the sender of the message.

d. There are different kinds of people and objects (Doctors, Lawyers…). Just because a person or an object can respond to the same massages as another object, doesn't mean that it is necessarily of the same kind.

e. Different people and objects of the same kind respond to a given message in the same way.

f. People and objects have a current "state" that can change over time and may effect how a message is handled.

g. The only way to get a person or an object to carry out a request is to ask (send a message).

h. A person or an object may receive a message requesting some service and may in turn request help from another person/object, so that a "server" in one transaction may become a "client" in another.

On the other hand, the metaphor breaks down in a non-concurrent system as objects interact sequentially, while people always interact concurrently.

(White Box) In systems building, it is sometimes possible to build in such a way that information system components directly model components of the business process being handled. In this case, the real world system becomes a metaphor for the information system being built. While this may not be the dominant mechanism of object-oriented system building today, it is still useful for beginners who need help initially in finding the objects in a system being modeled.

(CPU) Modern computer systems are seldom exactly like the memory model that many instructors give their students to think about the nature of computation in imperative languages. The simple model, however is usually good enough for most purposes. Notice that this is an abstract metaphor.

A workshop was held at OOPSLA '97 on Non Software Examples of Design Patterns. The results can be used to motivate a number of ideas of software design. http://www.agcs.com/patterns/papers/tutnotes/index.htm

Pedagogical Pattern #35

Toy Box

(Version 2.1, July 2000)

The intent of this pattern is to give the students broad historical and technological knowledge of the field by letting them "play" with illustrative pedagogical tools.

PROBLEM/ ISSUE

Students often have no real concept of the breadth of application of computer science or of its basic theoretical underpinnings such as Turing Machines. Since we have to teach a lot of things, it is often difficult to give this breadth to the students if we want to give them the required depth of understanding.

AUDIENCE/ CONTEXT

The pattern can be used in several courses and at several levels. It can be used very early in programming courses and in teaching upper level courses as well.

FORCES

Students must deal with a great amount of detail. Sometimes it obscures how the detail is to be used.

Many applications of computer science are very complex and outside the skill level of novices to absorb completely.

While we are teaching programming, it is nice to let the students program with things that will teach them some simple ideas from courses that they will encounter later in the curriculum.

Students have to program with "something." Often it is just integers and floats at the beginning. It could be things more exciting to them and which teach them on a variety of levels simultaneously.

We would like to be able to use our contact time with the students as efficiently as possible. On the other hand, students need to work on problems.

SOLUTION

Prepare application skeletons, each of which is from some key area such as as database or spreadsheet. Each skeleton forms the framework for student activities and exercises and each embodies in the simplest possible way the key idea of that key area. While the framework embodies the key idea in a simple way, these can build on each other so that they can become quite complex overall if they take advantage of lessons learned earlier.

Students examine and interact with these specially written, scaled down, examples of realistic applications such as word processors, database management systems, and spreadsheets. These applications are reduced down to their simplest form possible.

Choose student exercises to give the students a rich set of experiences about what can be done and what is important in computer science. These exercises are supported by a library of instructor provided materials that make learning fun. Finite Automata and Turing Machines can be introduced this way in the first courses.

Give students a library of classes that can be used to implement some complex functionality. These are the building blocks. They then use them to build some artifact. Instead of programming with integers and arrays, they program instead with logic gates, for example. You need to prepare some classes in advance to enable this, however.

Distribute a class library that implements the basics of some functionality. The students may either use this unchanged or extend it and use the extended version to build some project. Care must be taken that the hierarchy is soundly built, demonstrating excellent techniques, and excellent structure.

DISCUSSION/ CONSEQUENCES/ IMPLEMENTATION

In object oriented programming courses this is especially useful, as classes provided by the instructor can and should be used by students in any case. These can be carefully chosen to emulate larger systems.

This pattern allows students to actively work with larger programs than they can develop completely themselves. If the tools are chosen correctly, they can also gain breadth of understanding of the entire field. If the class hierarchy is well built it also serves as a good model for students building their own classes and hierarchies later, though the intent is not to teach OOP specifically.

SPECIAL RESOURCES

Fairly rich hierarchies of classes implementing the tools must be supplied by the instructor. This is labor intensive, but resources can easily be shared.

RELATED PATTERNS

This pattern can provide examples for cycles around a *Spiral*.

A collection of these can be *Larger Than Life*, especially if closely interrelated.

A well chosen collection could also demonstrate *Lay of the Land*.

EXAMPLE INSTANCES

A set of classes that implement logic gates and circuits. Gates can be connected together to form circuits.

An assembly language simulator that lets students get familiar with goto/register programming without all the details of a real machine.

An extension to the above that shows some of the problems with concurrency. A set of classes that let students experiment easily with readers/writers conflicts, for example.

A simple game with complete information that plays against the user but learns from its mistakes.

A simple spreadsheet like program. The program can store "programs" in cells using the simple assembly language of earlier "toys".

A set of classes implementing a simple relational database that can be queried with a simple language.

Some of these things are complex, but with appropriate scaffolding, even beginning students can use and extend the tools appropriately.

Decker & Hirchfield's Analytical Engine is a good use of this pattern.

CONTRAINDICATIONS

This requires a fairly long lead time and a lot of effort on the part of instructors, creating and finding materials. Don't try to use this pattern without commitment of time and resources.

REFERENCES

The Analytical Engine : An Introduction to Computer Science Using the Internet, Rick Decker, Stuart Hirshfield, PWS, 1998

Pedagogical Pattern #36

Tool Box

(Version 2.1, July 2000)

The intent is to let the students build a tool kit in early courses for use in later courses. If well thought out and implemented, it can be a wonderful guide to reusable software. Students become apprentices in the same sense that young people once served as apprentices in medieval guilds. There they spent their early years building tools they would need if they were to achieve master status in the guild.

Students build things in early courses that will actually be used later in the same course and in later courses as well.

PROBLEM/ ISSUE

Too often students work on a problem and write a program to solve it and then discard the code and work on other problems. Many times students wind up re-writing the same code very frequently. This is desirable if it enforces key ideas, but is often just drudge work that detracts from the key ideas under discussion.

AUDIENCE/ CONTEXT

The pattern is used most heavily in the early programming courses, especially the data structures course. However, it can be added to as well as used in later courses such as database, AI, operating systems, compilers, and the like.

FORCES

Students in later programming courses make use of knowledge from earlier courses, but in the past made little use of the actual programs built there. This gives them the lesson that things are continually rebuilt in software projects. In reality, a working professional seldom builds a stack class, instead reaching into a tool box to pull one out.

Having a personal tool kit of reusable software components can be a wonderful help in a difficult project.

Building a tool kit stresses different, but important, skills. Tool kits need to be maintained. Their elements need to be built with an eye to generality. Encapsulation is more important in toolkits.

SOLUTION

Student exercises have multiple parts. One part of each exercise is to build a general tool that might be useful in other projects and to take some effort in its proper formulation for reuse. The design for reuse must be explicit and must be discussed by the students and commented on by the instructor. Groups of students can combine individual designs of the tools, discuss the relative merits of each and then build a common implementation that improves each of the individual designs.

DISCUSSION/ CONSEQUENCES/ IMPLEMENTATION

Students gain skill in the early courses building reusable components. To the extent that they fail, the lessons are reinforced in the later courses when they need to rebuild parts of the tool kit for use in the projects of those course. Languages supporting reusability need to be used. Most of these languages are either object-oriented or functional. Scheme, C++, Eiffel, Java, Ada are good choices. There are a few others such as Modula 3, Beta, Oberon, CLOS, and Standard ML. Languages supporting genericity (templates) are especially useful.

Instructors can provide some tools (data structures and algorithms) in the form of class hierarchies or other libraries. Students complement this collection. Time should be spent by students and instructor evaluating student built tools for correctness, of course, but also for the potential for reusability. Instructors in later courses must be aware that students have these personal tool kits and should use exercises and projects that exploit the tool kits and permit their extension.

Coordination with later courses is an important element of this pattern.

SPECIAL RESOURCES

Give special thought in the design of the early courses as to what tools are broadly useful, but especially which tools will necessarily be useful in later courses. Platform independence of the tools should be one consideration. You need an overall plan and must be able to provide implementations of those tools that the students will need, but will not build themselves. For example, give them a singly linked list and have students build a doubly linked list. Both would be included in the toolkit.

RELATED PATTERNS

Fixer Upper can be used by the instructor to provide partially flawed tools for the students to comment on and repair.

Tools can be built and refined as part of each cycle of a *Spiral*.

EXAMPLE INSTANCES

Kernighan & Plauger's Software Tools books were a good use of this pattern. This book should probably be read by anyone who wants to implement this pattern. It shows how a high degree of reusability can be achieved with extremely simple tools.

Many data structures, with only slight modification, could be taken as early examples of this pattern.

A Stack class is often built early. It doesn't take a lot of extra work to build it originally in such a way that it is generally useful in many projects. Any extra work can be impotant in student understanding in any case. For example, a Stack should not simply write to standard output when an attempt is made to pop it when empty. A discussion of layers of concern (functionality vs. error handling) is not wasted.

Lisp, ML, and Scheme depend heavily on libraries of functions. These can be built in early courses and used later.

CONTRAINDICATIONS

This pattern is probably not for use in theoretical or "concept" courses.

REFERENCES

Software Tools in Pascal, Brian Kernighan, P.J. Plauger, Addison Wesley, 1981

Pedagogical Pattern #50

Lay of the Land

(Version 2.1, July 2000)

Students are given some early experience in examining a large artifact, beyond their ability to produce, with the intent of showing them the complexity of the field they are about to study.

PROBLEM/ ISSUE

Often we teach courses that cover a lot of ground. If students don't see the big picture fairly early, they may never see it while lost in a sea of detail. We would like to show students the breadth of a large topic so they have something to relate to and don't get lost in the details as the course progresses.

AUDIENCE/ CONTEXT

This has very wide applicability to almost every domain. It is especially useful in teaching topics with a lot of parts that must fit together in certain ways. Teaching programming is one example. Teaching design methodology is another.

FORCES

Teaching is often incremental, with topics introduced one after the other. This may not be the best way to learn, however, especially if you try to impose a strict ordering discipline on ideas. Constructivist educational theory is at odds with such strict ordering, which may not be meaningful or useful to the individual student. The mind is not a blank slate into which you pour ideas.

Students need to see the big picture too, as well as the detail.

Early on, they can produce only simple artifacts, but they can examine, if only in a superficial way, a complex artifact. Most people can read and understand something much more complex than they can themselves produce.

Seeing the big picture can give them motivation for the study of the parts as they have an idea of how they might be used. This is especially true if you can make the big picture compelling with a really interesting example.

SOLUTION

Give students a large artifact to examine early in the course. They can see what it is that they are supposed to be about in that course and what kinds of things they will be expected to master.

The artifact should have the complexity of something you would like them to be able to produce at the end. Spend time examining the parts and their interactions. You may spend more time on trade-offs inherent in the design or not, but you should mention the notion of design tradeoffs at least. The artifact should include most of the elements that are the proper study for that course. It is good if the artifact has some subtle points. If questions arise initially on these points they may be deferred, but it may mean you have better students than you think. Return to the artifact throughout the course and reveal and discuss its subtle points.

DISCUSSION/ CONSEQUENCES/ IMPLEMENTATION

Students get to see a target for their study. They also have a model on which they can base their own work.

The artifact must be prepared ahead of time. Student projects from prior years are a good source, though they may need to be modified somewhat to emphasize points you believe are most important. If the artifacts are large, a good way to transmit them to students is via the Internet, especially the Web. This way other educators can use them also.

SPECIAL RESOURCES

Good solutions to larger projects are needed. These must exhibit excellent structure and style, as we are hoping that the students will emulate their structure and style.

RELATED PATTERNS

One of these could easily be *Larger Than Life*.

A *Spiral* could be used to examine parts of the artifact on successive cycles. This is especially true if the parts are tightly coupled.

The artifact could also be a *Fixer Upper*.

If chosen carefully, it is also an *Early Bird*.

Test Tube can be used to let the students answer their own questions about the artifact.

See also Kerstin Voigt's Big Picture on a Small Scale at
http://www-lifia.info.unlp.edu.ar/ppp/pp46.htm

EXAMPLE INSTANCES

A large Object-Oriented program with a few classes interacting in an interesting way is a good choice in a first programming course using an object-oriented language. A complete design with many documents can be used in a systems design course. A complex database design with entity relationship charts and tables can be used in a database design course.

Some compiler course instructors give the students a complete compiler for one language and then ask them to provide a compiler for another over the course of the term.

Pedagogical Pattern #30

Fixer Upper

(Version 2.1, July 2000)

Giving a student or group of students a large artifact that is generally sound but with carefully introduced flaws can both introduce a complex topic early and serve as a way to introduce error analysis and correction. Students are asked to repair and discuss the artifact.

PROBLEM/ ISSUE

Too often students work on only "toy" problems because they may not have the experience or skill to build large artifacts from scratch and there is only just so much time. But all realistic problems are large and the day in which small problems were interesting is about past. We want to get students to work on large artifacts without overwhelming them. On another front, students also have difficulty when unexpected errors arise in their own work. Compiler and run time error messages, for example, often leave them lost.

AUDIENCE/ CONTEXT

The pattern can be used in several courses and at several levels. It can be used very early in programming courses and in teaching analysis and design. It can also be used to show the overall structure of a solution methodology.

FORCES

We often need to introduce students to a new field requiring mastery of several topics. Students often fail to see how the topics fit together when introduced sequentially. They also often fail to have a grasp of the means of locating and correcting errors.

Fixing a larger artifact than can be created by students is generally within their grasp. It gives them a better sense of scale of interesting problems and permits them to integrate a number of issues into the solution of a single problem.

Students can benefit from seeing larger problems than they can solve at their current state of development. They also need critical analysis skills and the ability to evaluate programs, designs, etc. (See *Lay of the Land* and *Larger Than Life*).

SOLUTION

Give students are artifact, such as a program or design. The artifact proposes to be the solution to a problem, but while generally correct, the instructor has purposely introduced flaws into the program, design, or whatever. The artifact should be fairly large and should contain a number of flaws. Most of the flaws should be simple and obvious to most readers. There should be one or two deeper flaws.

Students are asked to find and correct the flaws. They can also be asked to discuss the nature of the flaws found and the reasons for their changes. Finally, they can be asked to discuss the overall structure of the artifact and draw inferences from it.

DISCUSSION/ CONSEQUENCES/ IMPLEMENTATION

This pattern allows students to actively work with larger artifacts than they can develop completely themselves. They benefit since finding flaws in their own work is a valuable skill. In programming, students see lexical, syntactic, and semantic errors. In design, they can see the effect of incorrect partitioning of responsibility.

It is important that the overall structure of the artifact be sound. If it is a program it should be well designed and written, with good choice of identifiers. If it is an analysis or design document, its overall structure should be sound with a clear map to the problem statement.

The best way to develop such an artifact is to start with an excellent solution to a problem and then doctor it by introducing flaws. There must be different kinds of flaws, but probably not structural flaws if you are dealing with novices. This latter rule can be broken if the artifact is introduced later in the course rather than at the beginning, at a point at which structure is the main issue. When used as an *Early Bird* instance, however, concentrate on flaws of detail, rather than structure.

Follow up. Most such exercises cause the students to generate questions that can be a fruitful source for classroom discussion. If the instructor is careful to introduce certain flaws, the student can be led in a desired direction to further explorations.

Note that in the United States, a "fixer upper" is a house for sale that is in poor condition. They are sold to people, mostly young, with more energy and enthusiasm than money. The concept is that you need to repair it (perhaps extensively) after purchase.

SPECIAL RESOURCES

A problem and a well designed and implemented artifact that you can manipulate to provide the necessary errors. These can come from industrial quality designs and professional books. They can also come from projects from previous years, provided that you are willing to modify and improve the artifact so that it is truly of high quality prior to the introduction of flaws.

RELATED PATTERNS

When the artifact is introduced at the beginning of the course and introduces the key ideas of the course, then it is also an *Early Bird*

When the artifact is very large then it is also an instance of *Larger Than Life*

If the artifact is also illustrative of some important topic, then it is also an instance of *Toy Box*

When the artifact illustrates the overall topic of interest in the course, it demonstrates *Lay of the Land*

If the repaired artifact is useful in some larger context it can also be a *Tool Box* instance.

This has some of the same goals as *Mistake*, though it is approached differently. Here we find errors. In *Mistake* we make specific errors.

EXAMPLE INSTANCES

1. Beginning programming. Here the artifact is a program illustrating a number of syntactical constructs that have not yet been introduced in class. (It has been used as the first assignment.) The program can be large enough that its structure is not obvious. Two or three classes with several short methods each is about right. One part of the program might be more complex. Together with the driver, there should be three or so pages of code. The errors can be mostly syntactical and lexical, so that the compiler can find them. One or two semantic errors should also be introduced, so that the program does not perform as expected. More serious and perhaps for more advanced students is the failure to fulfill a precondition contract. Ten to fifteen errors is about right if most are easily caught.

Even a single class can be introduced that has a flawed public interface. Students can be asked to analyze the consequences of this in relation to the likely current and future use of the class.

2. Introduction to design. Here a problem is presented and a design for the solution. Six to ten major elements in the design is about the right scale. The design should have a few simple flaws, such as missing message paths or missing functionality within a class. Improper partitioning of responsibility in a small region of the design is relatively easy to introduce. One subsystem could be left out. If the design requires several documents, there might be an inconsistency between the documents. Five or six flaws is probably enough.

3. Some (Linda Rising) have used this pattern with very large artifacts -- large enough that they can't be understood by a single person at the level of the students and therefore require teamwork.

4. Linda also suggests having students type in large examples. They will always make mistakes. Then have them find and disucss the corrections.

CONTRAINDICATIONS

This must be carefully used if student honesty is an issue. It is easy for one student to point to the locations of errors in C++ programs, for example. One way to address this is to use large artifacts that require teamwork. Another is to ask questions concerning the structure as well as the errors. Finally, the students can be asked to examine the artifact before they are given the full set of questions that will be asked about it.

Some students are frustrated by such large artifacts. The instructor must be prepared to provide support and encouragement that the real world really is like that and that it is ok to initially (a) be frustrated and (b) lack knowledge.

ACKNOWLEDGEMENTS

Kyle Brown (the pattern shepherd) and the participants in the OOPSLA '98 Pedagogical Patterns Rewriting BOF were very helpful in improving the presentation of this pattern.

Pedagogical Pattern #53

Larger Than Life

(Version 2.1, July 2000)

Students of Object-Oriented programming and design can and should examine and use computing artifacts much earlier than they can design and build them themselves.

PROBLEM/ ISSUE

Realistic artifacts worked on in this field are large and complex. Students also often are made to work alone, when we know that most of the real work they do will be in teams. Finally, much real work in the world of computing is not to build an artifact from scratch, but to modify an existing one. Most real projects in the Object Oriented world have a *piecemeal growth* life cycle in which a large artifact grows from a small, well designed, core.

AUDIENCE/ CONTEXT

Students at all levels, from raw novices to quite experienced students.

FORCES

Students cannot initially produce programs and other software development artifacts of any complexity. However they need to be introduced to the complexity of real systems. They can, in fact, examine and learn from programs and other artifacts far beyond their ability to produce. Much time is spent in class on quite low level details, yet students need to see the big picture.

If the students build everything that they use, they will get a very stilted view of how real software is produced in the world. They are also unlikely to have a real appreciation for the complexity of real systems, or the requirements of software that will be used by people other than its author.

In courses other than computer science, students regularly work with texts and other artifacts far larger and more sophisticated than they could produce themselves. A literature or history course is a good example. The same occurs in sociology courses. These artifacts teach the student what is best in the field and should be emulated.

SOLUTION

Give the students access to large programs and designs well before they have the ability to produce them. These artifacts can be used as the basis of exercises. Students can make small modifications to large programs and they can extend them in simple ways early on. They can also make corrections to flawed large programs if the flaws are of modest difficulty (see *Fixer Upper*).

For example, students are often asked to study a compiler for one language while writing the compiler for another in the compiler course. Even in the first course, a fairly large skeleton program (Grocery Store Simulation) could be used for study and as the basis for exercises.

Use only artifacts of high quality design, so that students will have a template to emulate in their own work. An exception is made if the goal of the exercise is to improve a large but poor design.

DISCUSSION/ CONSEQUENCES/ IMPLEMENTATION

Unlike the days in which languages were simple and libraries non-existent, students today need to be able to use libraries written by others. These libraries are usually more complex than the students could create. They may also contain much code of generally low conceptual content, so that it isn't worth course time to develop them from scratch. Their design, however, may be of some interest.

SPECIAL RESOURCES

Large computing artifacts of good design.

RELATED PATTERNS

If the artifact requires repair then it is also a *Fixer Upper*.

If it stresses the big ideas and is given early it is an *Early Bird*.

If it is generally useful it is a *Tool Box* entry.

If the artifact covers most of the main concepts in a course it is also an instance of *Lay of the Land*.

EXAMPLE INSTANCES

The Standard Template Library of C++ is such an artifact. So is the AWT of Java. Design documentation for large systems are examples. Complete programs such as compilers and operating systems are also useful. One standard technique in a Compiler course is to give the students a complete compiler for one language at the beginning of the course. This artifact is then examined over the term and the students build a compiler for a different language.

In any object-oriented course, students will benefit from the use of libraries prepared by others, either instructor written, or perhaps standard libraries.

Linda Rising gives an example of a software design project that is both *Larger Than Life* and a good *Lay of the Land*. It has some features of a *Fixer Upper* as well.

Owen Astrachan discusses the idea of a *Software Cadaver* which is an artifact for study by students. It often exhibits Larger Than Life characteristics.

REFERENCES

Rising, "Removing the Emphasis on Coding in a Course on Software Engineering", SIGCSE BULLETIN, February 1989.

Pedagogical Pattern #

Student Design Sprint

(Version 2.1, July 2000)

Students need to practice design at all levels. This pattern gives them quick feedback and peer review on early attempts.

PROBLEM/ ISSUE

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.

AUDIENCE/ CONTEXT

This pattern applies to courses for novices in programming and in design. It can also be used with experienced developers who are new to a topic.

FORCES

Students need to practice design, both program design and system design. But design is hard.

They need feedback on early attempts. The quicker the feedback the better.

They need to see good designs and to critique poor designs.

They need to see the consequences of poor designs.

SOLUTION

This activity can take place in the 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 minutes. There should be a written sketch of the design in that time. 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 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 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.

DISCUSSION/ CONSEQUENCES/ IMPLEMENTATION

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.

If the exercise is to design a system using CRC cards, then a good follow up is to apply the *Role Playing* or *Incremental Role Play* pattern. Have the students with a good, but not perfect, design explore the communication paths implied by their design in a search for bottlenecks.

This pattern is not restricted to analysis and design. It can be used for program design and data structure design as well.

RELATED PATTERNS

This can lead to a *Role Play* pattern.

EXAMPLE INSTANCES

Alistair Cockburn 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 first and second courses. 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 in data structure design in the data structure's course. 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.

REFERENCES

The coffee machine can be found on Alistair Cockburn's web site: http://members.aol.com/acockburn/

Role Playing, David Belin, http://www-lifia.info.unlp.edu.ar/ppp/pp5.htm

Incremental Role Play, Jutta Eckstein, http://www-lifia.info.unlp.edu.ar/ppp/pp7.htm

ACKNOWLEDGEMENTS

Alistair Cockburn's Coffee Machine Design exercise.

Pedagogical Pattern #33

Mistake

(Version 2.1, July 2000)

Students are asked to create an artifact such as a program or design that contains a specific error. Use of this pattern explicitly teaches students how to recognize and fix errors. We ask the student to explicitly make certain errors and then examine the consequences.

PROBLEM/ ISSUE

People make mistakes. Students often don't know how to interpret the error messages provided by their tools or what to do to solve problems that are diagnosed by the tools. Debugging is an essential skill, whether done with a sophisticated debugger, or just by comparing actual outputs with expectations. Students don't initially know what occurs when errors are made and so don't know what to do when they see program diagnostics and incorrect outputs.

AUDIENCE/ CONTEXT

This is very applicable to the early stages of learning programming. Syntax and semantic errors are frequent and students need to become familiar with the messages produced by compilers and run-time systems and what they indicate about the program.

The pattern could also be used in an analysis or design course in which certain specific, but fairly common, errors are required to be part of the design. The instructor can provide a part of the design, which is flawed in some way, and the students are asked to complete the design without changing the instructor's part. (See the *Fixer Upper* pattern also.)

This pattern is often used in Database and Operating Systems courses where the students are asked to explore the conditions leading to deadlock, by specifically causing it. Similarly, in learning concurrent programming, students are often asked to write programs in which race conditions between processes lead to the corruption of data.

FORCES

Students, like professionals, make errors. Professionals generally know what the indications of an error are, but students do not.

We usually help students avoid errors, but they occur anyway. Students should have a way to explore the effects of errors so that they can recognize them by their effects.

SOLUTION

Ask students to produce an artifact with certain specific errors (usually a single error). The effect of the error is then explored.

For example, students are given an assignment in which they are instructed to create and run a program with certain specific errors. They are then asked to comment on the diagnostics produced and/or why no diagnostics were produced for the error.

DISCUSSION/ CONSEQUENCES/ IMPLEMENTATION

Students become more familiar with errors and how to correct them. Having seen specific errors, they can learn to avoid making them in the first place.

You should carefully prepare exercises using this pattern and be sure to do the exercise before the class does, so that unanticipated occurrences can be prepared for and you can later provide explanations of precisely what happened.

Follow up. It is important that the students get a chance to discuss interesting things that occur while making these errors.

One especially useful technique is to ask students to make errors that, when they occur accidentally, are especially hard to diagnose when made accidentally. One example of this is errors made in C++ template instantiations.

SPECIAL RESOURCES

The instructor simply needs knowledge of common errors.

RELATED PATTERNS

In *Fixer Upper* the instructor makes the errors and the students correct them.

In *Test Tube* we ask for explorations. Here we ask for explorations of specific errors.

EXAMPLE INSTANCES

In addition to those mentioned above, this pattern can be used effectively to teach students about pointers in languages like C or C++, by having them make all of the common pointer errors purposely. This particular use is somewhat dangerous on computers that have memory mapped i/o and unprotected operating systems. Both syntax and semantic errors can easily be explored using this pattern.

One exercise from an old book [Teague] was to write a program that produced every diagnostic mentioned in the manuals for a given (Fortran) compiler. This is, not surprisingly, very difficult to do. Impossible, for some compilers, as the documentation and the compiler are not completely parallel.

CONTRAINDICATIONS

This pattern can be over used. It is best used early and in response to student questions.

REFERENCES

Computing Problems for Fortran Solution, Robert Teague, Canfield Press, 1972.

Pedagogical Pattern #52

Test Tube

(Version 2.1, July 2000)

Students can answer their own "What if…" programming questions with access to a computer. Sometimes this is quicker and more effective than formal documentation. Students can also explore the undefined parts of some computer languages.

Also Known As: **Try It and See**.

PROBLEM/ ISSUE

Students are often naive about what resources are available to them. They often ask questions they could answer for themselves. While this is not bad in itself, it would be liberating to them to know they can find their own answers quickly. Students need to learn to be effective in answering their own questions about programs.

AUDIENCE/ CONTEXT

Beginning programming courses.

FORCES

Students can take up a large amount of class time asking questions at a very low level of detail. They can also become very frustrated if they cannot find sufficiently detailed documentation to answer their questions.

SOLUTION

Give the students several exercises in which they are asked to write small programs to get the computer to answer simple questions of the form "What happens if…". These exercises should be frequent enough that students get in the habit of probing the machine for what it does, rather than the documentation.

DISCUSSION/ CONSEQUENCES/ IMPLEMENTATION

Some languages are underdefined. In these languages, the instructor will need to point out a few places where there are no rules. In particular, the order of evaluation of parameters in C++ is not defined in the standard. Therefore, the most a program will tell you is the effect of using the compiler at hand. It can't give you a general rule as there is none. Students need to be aware of these situations so that they can program defensively and not draw misleading conclusions.

SPECIAL RESOURCES

Student questions can be a good source for exercises of this kind. The instructor can take a student question and turn it into a very short term (overnight) exercise. If a laboratory is available, students can immediately test out hypotheses about how things operate.

RELATED PATTERNS

This can be used in conjunction with *Fixer Upper*.

If the purpose of the exercise is to generate errors, then this is an instance of *Mistake*. This pattern is more general than Mistake, however.

This can be used effectively to make cycles around *Spiral* move quickly without getting bogged down in too much detail.

EXAMPLE INSTANCES

The meaning of for loops in C++ can be explored in a sequence of exercises in which the initialization, test, and "increment" portions of the loop are varied.

When a while loop exits (between executions of the body) and when it does not (as soon as the condition becomes true in the middle of the body) can be explored in exercises.

The difference between value and reference parameters and the meaning of const can be explored.

Pedagogical Pattern #

Fill in the Blanks

(Version 2.1, July 2000)

Students can often learn a complex topic by building several small parts of a larger artifact. This aids both their reading and writing skills.

PROBLEM/ ISSUE

Beginning students need to work on larger projects than has been typical in the past, yet they are unsophisticated and have only a little knowledge and skill at the begining. On the other hand, they can learn by reading as well as by doing. How can you get the students working on larger artifacts without overwhelming them?

AUDIENCE/ CONTEXT

This pattern is intended for introductory programming courses that attempt to move students quickly to difficult material. It is probably possible to adapt this to design as well.

FORCES

Students have a difficult time developing complex programs from scratch. The programs that they can build early are usually quite boring.

Students should see how the work that they do fits into a larger context.

Students can learn to read programs earlier than they can learn to write them. But, they should not be permitted to be overly passive in their reading.

SOLUTION

Prepare a very well designed program or part of a program and remove a few pieces of the code. Give the result to students with instructions to fill in the missing parts. The missing parts need to be well specified. It is also best if the result will be put to some use immediately so that students can see the effect of their work.

DISCUSSION/ CONSEQUENCES/ IMPLEMENTATION

The overall design of the artifact must be excellent.

The missing pieces should be carefully selected so that students can deduce something about the missing parts from the supplied parts.

In some cases the missing parts can be carefully specified with pre and post conditions. In others, the specification step can be left to the students. They could even be created in class in a *Student Design Sprint*.

The missing pieces could require different skills to complete. Some missing parts might require little beyond basic syntactic knowledge, while some might require deep semantic analysis. If used repeatedly (as in a *Spiral*) the successive uses could require deeper analysis each time.

SPECIAL RESOURCES

Any well designed artifact that can be modified will do.

RELATED PATTERNS

In *Fixer Upper*, students repair a carefully broken artifact. Here they complete an artifact carefully left incomplete. The artifact can also be *Larger Than Life*, or an *Early Bird* or *Lay of the Land*.

EXAMPLE INSTANCES

A simple example is a class definition in a language like C++ or Java in which some of the method bodies have been omitted. Slightly more complex is one in which part of the interface itself has been left out.

Even simpler, leave out a single line of code somewhere, though you must provide appropriate comments so that students don't need to guess where these blanks are.

You can provide an application that requires certain library code (stack, queue,…) that the student must provide.

At an extreme end of the spectrum, a major component of a project could be left to the students.

ACKNOWLEDGEMENTS

Mail from Peter Henderson (SUNY Stony Brook) reminded me of this. He uses it extensively in CS I.

© Copyright 2000-2006, Joseph Bergin

Pedagogical Pattern #

Gold Star

(Version 2.1, July 2000)

Students should get praise for what they do well.

PROBLEM/ ISSUE

Sometimes students amaze us in class or in their turned-in work. Normally the reward structure is very private, which is fine, but then we lose the opportunity to show other students what we value most highly.

AUDIENCE/ CONTEXT

All students at all levels.

FORCES

Students look up to professors as role models. They want and need our praise. Praise can be a prime motivator. Expecially when it comes from someone who is respected.

Students work best when they feel good about themselves.

Students, like their professors, like to feel appreciated.

SOLUTION

When students do something well, praise them for it.

Give a token of appreciation for work well done. This can occur either publicly or privately. It can be a simple few words spoken in private or it can be an insertion into the student's permanent academic record.

The forces at play here also suggest that you should never belittle a student even for poor work.

DISCUSSION/ CONSEQUENCES/ IMPLEMENTATION

This works for young children as every elementary school teacher knows. It is often unexpected by young adults, and this alone can account for some of its effectiveness.

EXAMPLE INSTANCES

My compiler course is seen as very difficult. As an added incentive, I publish the "Gold, Silver, and Bronze Medal" winners each year on the internet.

When bumping into a student in the halls or elevator, you can mention how much you liked the solution they provided in the previous assignment (if indeed you did). Simple things like this can (and do) change lives.

Dartmouth College had (has?) a policy that a professor can insert a special note into any student's permanent file when they do something out of the ordinary. This might be an especially good job on a project or a special contribution to a class or to campus life in general. These notes are highly valued by students.

An extravagant example of this is when a professor publishes a student's best work on a course web page.

The name, of course, comes from a fairly common practice in elementary school in which a teacher will put a gold star on well done papers. Some teachers have special ink stamps made with special sayings for their students. I actually pass out a few gold stars in each course (available in any good stationery store), mostly for asking really key questions. I make this very public.

CONTRAINDICATIONS

If you are going to do this, then don't forget to be consistent.

Pedagogical Pattern #55

Grade It Again Sam

(Version 2.1, July 2000)

To provide an environment in which students can safely make errors and learn from them, permit them to resubmit previous assignments for reassessment and an improved grade.

PROBLEM/ ISSUE

Students are in our classes to learn. They make mistakes. Sometimes they don't take risks because they fear they will suffer because of the grading structure. We want to help the students to learn, even when they struggle, and to reward them for thier success, especially when hard won.

AUDIENCE/ CONTEXT

This pattern can be used ubiquitously--in every course. The only real exception is a course with a major project due at the end and which is graded only then.

FORCES

A college or university is supposed to be a safe place in which to learn. This implies making mistakes.

Students shouldn't come to a course to prove they don't need to take the course.

We all make mistakes, novices most of all. Students should have an environment in which they can learn from their mistakes.

Professors are busy and have a lot of things to do besides teaching.

SOLUTION

When an assignment is turned in and graded, permit the students to resubmit it with changes for a reevaluation and a new grade. The grade will be higher than the original, but there will be some small penalty charged so that a "perfect score" isn't attainable on a reassessment. A limit can be put on the number of reevaluations that can be done or not. In a large class (more than 30, with no teaching assistants) perhaps one reevaluation is all that can be managed. In a smaller class there may be no need to limit it at all.

A variation is to permit the student to resubmit the work on their lowest grade to date, rather than just dropping the lowest score as is commonly done. This permits the student to think again about the material covered.

DISCUSSION/ CONSEQUENCES/ IMPLEMENTATION

Some students take advantage of the system, but enough benefit greatly from it so as to make it worth doing. Students should turn in all previous editions of an assignment with the new work so that the professor can easily see from previous comments why points were lost. It can also help if students mark the changes (change bars or hiliter). It also helps if work is neatly done and submitted in a manila folder.

You may also need to cut off additional editions for a student or for the class under certain conditions. Sometimes the potential for improvement is small and the student's time is better spent elsewhere. Sometimes the professor doesn't have time for the reevaluations. At the end of the course it will probably be necessary to set an absolute deadline for resubmissions.

The purpose of this pattern is to permit a student to spend additional effort on material with which they have special difficulty.

Note that the name is a play on the film Casablanca, in which Rick (Humphrey Bogart) repeatedly tells his pianist to "Play it, Sam." Many people remember it as "Play it again, Sam," though he never said that. Woody Allen has a film with this name also.

SPECIAL RESOURCES

This is time and labor intensive.

EXAMPLE INSTANCES

I use this method in all courses and have done so for ten years. I don't know of other instances. My experience is that there are seldom more than three editions, though I don't limit editions. Some students have grown tremendously with this method. You work more with the students that really need the help.

Neil Harrison mentions a technique his brother in law, Allen Bramwell, uses. He gives the students a final assignment with lots of time left. He tells them they can turn it in at any time before the due date, and he will return it with comments in 24 hours. Some students get two or three rounds of feedback, and naturally get the highest grades. Those who don't get any feedback do not fare well.

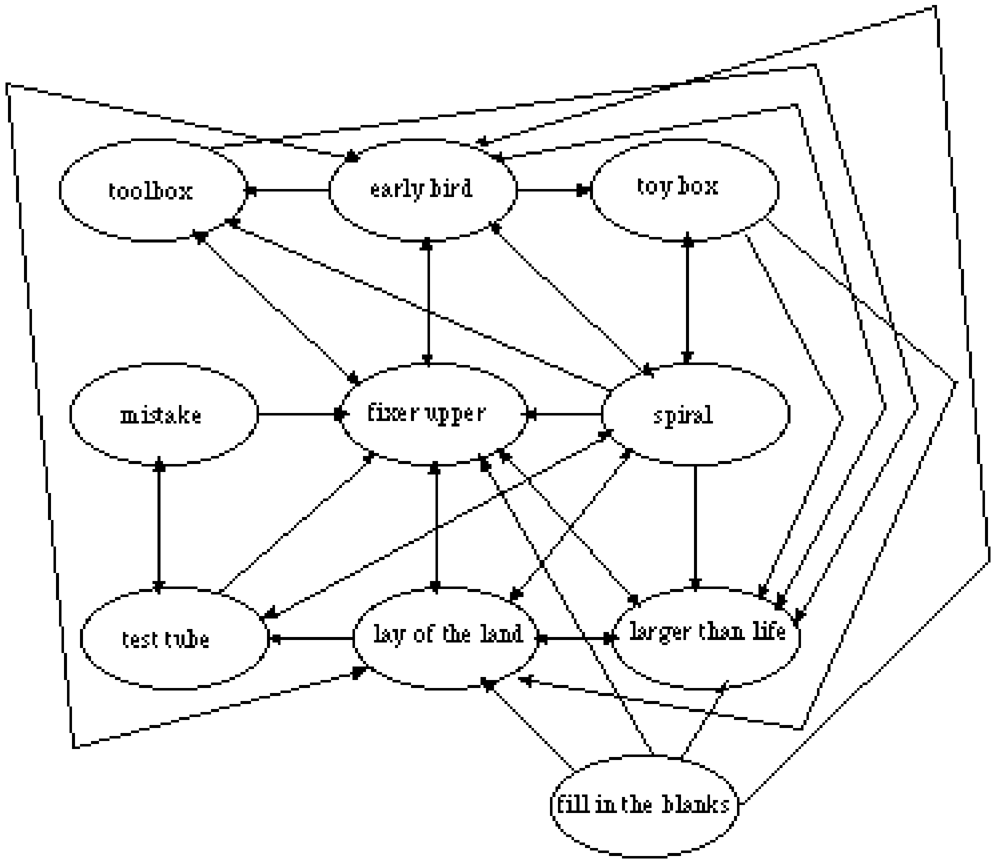
CONTRAINDICATIONS

If students are sloppy they may turn in especially poor work the first time, thinking that there is no risk. I use a 10% penalty on regrading to avoid this somewhat. The penalty is charged only on the difference between the initial score and the perfect score.

If students are overly conscientious then they may spend too much time on new editions to the extent that they neglect other work. To help avoid this, I never give the last three percent on rework.

Synergy Among Some Of These Patterns

These patterns work well together. They are the result of many years of teaching. It is object-orientation, however, in which they find their greatest manifestation. The first ten exhibit the following interconnections. At this point the arrows are just cross references within the patterns. I have begun to sequence these with others into a proper language for course development. See http://csis.pace.edu/~bergin/patterns/coursepatternlanguage.html



Back in the late 1970's when U.S. educators were moving from Fortran to Pascal we made a number of serious errors early on. Many of the books of the time were better as reference works than as texts for students. Most of the early books relegated functions to a very late chapter, often the last. Types weren't used especially well and ADT's were non-existent.

I remember what a revelation it was to find the book Karel the Robot, by Richard Pattis. Here were functions done first, not just early. Here was a very simple starter that showed the Lay of the Land of programming. It was simultaneously a Toy Box and the first cycle of a Spiral. Very powerful stuff, and very useful for both educators and students. The algorithms developed with Karel had direct impact on learning Pascal. For example, a robot searching for a beeper has exactly the same structure as a program searching a sentence for its period.

I fear we are making the same errors in moving from procedural programming to object-orientation. I've read papers by educators who claim that the "received wisdom" is that object-orientation should be taught late, as an add on to procedural programming. This goes counter to the experience of professionals in the object-oriented world who have seen how much differently one must think to be successful in this new world. If you want to produce object-oriented developers, they should learn this first (Early Bird).

The field is complex and has many parts. These must be shown to fit together in comprehensible ways (Lay of the Land). The programs we build now are much larger and more complex than was true ten years ago. Students need to be introduced to this scale (Toy Box). Reuse is more important now (Tool Box). Languages are more complex (Test Tube, Mistake).

Software is built in teams, and maintenance is often the dominant cost. Programs are too big to be understood by individuals, and they grow from a well-designed core (Larger than Life, Fixer Upper).

We are teaching new and better topics and tools than we did a few years ago. I believe that our methodology must change as well. When teaching Pascal, few relied on pre built libraries of code with which the students would interact, since most Pascal programs are built largely from scratch. Most C++ and Java programs are not built that way, however, so students need different experience than many of their instructors had. This implies that the instructor must be prepared, not only with knowledge and pedagogy, but with tools and libraries. It took me 25 years to know what I do about computer science. I expect my students to get there in less than a third of that time. Their experiences need to be very different from mine to make this possible.

In fact, object-orientation itself can provide these tools. Well designed classes are inherently reusable. They can be used to build toys which give the students both examples of object-orientation that they can emulate, but also tools to work with and extend. We must learn to use the tools to teach the tools. And we must match our pedagogy to the exciting new things that we are trying to teach.

NOTES

One of the themes that run through all of these patterns is the requirement that the artifacts that we show students be of the highest quality. Programs must be beautiful as well as correct. Designs must be excellent. Our programming and diagramming style must be of the highest order. Students look up to professors for guidance and emulate what we do. It is occasionally ok to show students work in development, but they should also see the final products. Perhaps I should have written this as a pattern (Quality is Job One).

Another common theme here is that students need to get early practice in working with software development artifacts much larger than they are capable of producing themselves. This is very common in other fields as mentioned above. More to the point, engineering and architecture students work with large engineering and architecture artifacts long before anyone would dare let them build much of anything. This has multiple benefits, among which is the ability to show students the best of both historical and contemporary work in their field. It gives students something to strive for and something to emulate in their own work. A professor of architecture will come to class with not just ideas, but with a corpus of work that the students will be asked to examine. Students will see the architectural drawings, and perhaps the engineering drawings as well. With luck, they may be able to visit one or more of the buildings of the architect. Computer science students need this same exposure to great works.

The reader interested in this work should also read the paper, Patterns for Classroom Education by Dana Lynne Goldblatt Anthony which was first presented at PLoP '95 and was published in Pattern Languages of Program Design 2, by Vlissides, Coplien, and Kerth, Addison Wesley, 1996. The above link is to an online version. Many of the patterns here and in Anthony's paper complement each other.

Acknowledgements

Neil Harrison of Lucent was the Pattern Shepherd for EuroPLoP 2000 and suggested many improvements. I'm always amazed at how much this process improves presentation and tightens up ideas. Thanks Neil. And thanks to the workshop at EuroPLoP. I learned a lot and had a wonderful time. I especially want to thank Linda Rising for her help on the style.

Permission is granted to Hillside Europe e.V. to publish and distribute this paper as part of the EuroPLoP conference proceedings.