



Capstone Courseware, LLC

33 Boylston Street
Jamaica Plain, MA 02130

877-227-2477
capstonecourseware.com

Design Patterns in Java Software

Will Provost

Instructor's Guide

Revision 5.0



Revision Notes

Revision 5.0 is the initial revision.





Course Overview and Philosophy

This course is not like most of our Java courses: it does not teach a specific technology or API, but rather patterns, best practices, and other such fuzzy notions. On one hand, students (and instructors) may find it a breath of fresh air compared to relatively dry API-focused material – on, say, the DOM Element interface. Patterns are fun, and subjective, and intellectually challenging. On the other hand, after a while students may be pining for some concrete detail. We've tried to go heavy on the examples, but even so: a discussion of design patterns that immediately “drops down” to the code level seems to miss the point.

The best offerings of this course have been to experienced Java programmers who have been in the trenches enough to appreciate the value of patterns without much “selling,” and have been driven by lively group discussions in which students bring their own experience to the learning of a pattern. Encourage this, go overboard in drawing people out. It's always nice to have good questions from students, but in a training on a specific specification or API that's all it is – nice. With this course it is really essential – without it the instructor will start hating the sound of his or her own voice sometime during day two.

This may go without saying, but it is also essential – and again, more so than for a raw-technology course – that the instructor be able to bring his or her own design and coding experience to these discussions. This class is the right place for “war stories” that might not go over in other classes, because patterns really are about those sorts of stories: “how I got it wrong, saw the light, and got it right, and what you can learn from my mistakes.” This is part of the idea of focusing as much as we do on “warning signs” for patterns: these things tend to resonate with students (of a sufficient level of experience, anyway) and are often the most successful part of a pattern discussion in getting students engaged.

Not all students will know OOAD, and especially UML, coming into the course. Our approach to this, rather than to back off of the use of UML and OO terminology, has been to embrace it fully. We stick to simple class/interface-diagram notation, and we use it everywhere. What we've seen is that even students with no prior UML and even those who are a little hostile to the notation do pick it up by example quickly enough. And some notation is needed – if weren't UML we'd have to invent something else. Be quick to jump to Appendix B to introduce the basic notation, or to answer a question about another diagram.





Hands-on work in this course is split between design and coding exercises. The design exercises can be pursued (a) individually, (b) in small groups, or (c) as a single group. The small-group approach has borne the best fruit thus far, and we recommend giving groups of 3 or 4 students separate breakout rooms, if possible, to minimize crosstalk. Visit each room at least once early, to clarify design instructions or requirements as necessary, and at least once near the end of allotted time, to give a hint or two. Have one group put their solution on the whiteboard, and let everyone discuss. (Having each group present is usually too time-consuming.) If you go with the full-group approach, try to step out of the discussion as much as possible, even as a moderator. Stick to clarifying requirements and let (insist that) students lead themselves.

In any configuration, the trickiest part of the design exercises is clock management. These will almost always run longer than expected, but even so the students usually consider the time to be well spent. It may be worthwhile to sit in with one or more students or groups and help them along here and there. As with coding exercises, there's a balance to be struck between spoon-feeding and students feeling overwhelmed and hopeless; with group design discussions, you have more of an opportunity to tune to the best balance by listening and (judiciously) participating.

As to the coding exercises, they are certainly advanced, and somewhat long. This class in general is no place for junior coders, and these labs are almost prohibitive for those with little coding experience coming into the class. The prerequisites make this clear, but if you are faced with an audience of non-coders or less-experienced coders, it's probably best to treat the labs as demonstrations – and to go very slowly. Another option, also mentioned in the course outline, is to reduce the schedule to two days, skipping the code-level labs. We offer an alternate timeline suggestion for this purpose.





Timeline

Day 1

Chapter 1	Recognizing and Applying Patterns
Chapter 2	Creational Patterns
Chapter 3	Behavioral Patterns (get started)

Day 2

Chapter 3	Behavioral Patterns (complete)
-----------	--------------------------------

Day 3

Chapter 4	Structural Patterns
Chapter 5	J2EE Patterns

Alternate Timeline (Without Coding Exercises)

Day 1

Chapter 1	Recognizing and Applying Patterns
Chapter 2	Creational Patterns
Chapter 3	Behavioral Patterns (halfway or more)

Day 2

Chapter 3	Behavioral Patterns (complete)
Chapter 4	Structural Patterns
Chapter 5	J2EE Patterns





The Eclipse Overlay

Some students and instructors may prefer to install an IDE and use it in working through the course exercises. Capstone Courseware provides an optional package of workspace and project files for Eclipse 3.1 for this course. (See the course Setup Guide for download URLs.) Instructors, use this package on your own initiative and at your own risk. You should have experience yourself with Eclipse before using the overlay package in the classroom. The workspace and projects have been tested lightly with the course but are not part of the standard product.

That said, this overlay should save a good deal of work for those who wish to use Eclipse instead of the text editor and command-line tools that are standard for the course. See the file `c:\Capstone\Java\Eclipse\ReadMe.html` for general notes on how to use the Eclipse overlays for Capstone courses. Be prepared to walk students through the first few exercises in Eclipse; the notes in this file are for experienced Eclipse users, and will not be clear to many students on their own.

One foible of Eclipse is that workspaces cannot be arbitrarily relocated without confusing the workbench a bit. To wit, if you choose not to install the workspace to the default, `c:\Capstone`, you will probably find that projects either don't build when they should, or that when you try to run them the Eclipse launcher "can't find the main class." The bottom line is the projects all need to be refreshed if they are opened from a path other than the one at which they were last open. So in this case the best process is to have everyone begin by opening, refreshing, and then closing all projects; this should clean up any odd behavior due to the "surprise" location.

Again, Capstone Courseware can only offer complete technical support on the standard course, and while we hope this overlay is convenient, it is not as thoroughly tested as the core lab image at this time. If a given exercise is giving trouble, please be certain to build and run it from the command line, using the SDK tools as prescribed in the student guide, before contacting Capstone.

We always welcome feedback on our courseware, and especially with this new undertaking we would appreciate whatever comments and criticism you might have. Eclipse, like all IDEs, tries to promote ease of use by providing many different ways of doing the same thing; this is convenient for users but does leave a lot of questions as to what's best practice. At this time we believe consensus is still forming around a number of basic practices, and we'd like to hear how you use Eclipse for training situations and how this overlay works out for you. Please contact Will Provost at provost@capstonecourseware.com.





Chapter 1:

This chapter is the “sell” – why should I use patterns, what’s good about them? How do they fit into my current practice as a developer? The title indicates the real focus here: it’s not about any specific pattern, but about learning how to recognize patterns and then to apply them. The example/exercise at the end should be helpful in driving the point home; you can certainly treat this as a small-group design exercise, but it (more than later ones) works well as a quick individual exercise as well – give students maybe ten minutes before discussing solutions.

Chapter 2:

The challenge of this chapter is probably dealing with the likely resistance to “factories for factories.” That is, factories in general sound great, but to really make Abstract Factory work the factory method for the factory itself is important, and even then it gets tricky to really enforce object-creation policies. It starts to seem like a bit of a quagmire, with visibility issues, hard-coded factory names, etc. It’s a good idea to forward-reference the provider approach and the use of system properties, as this is where it all seems to come together for many students. The lab doesn’t get this far, it sort of climbs most of the hill and stops short, but the examples right after the lab should really make the case for abstract factories or cascading factories and for strong enforcement of object-creation policies.

Chapter 3:

By contrast to factories, the first few patterns in this chapter sort of sell themselves. If anything, Strategy is too simple, almost feels like it doesn’t belong with the others. But the pairing of this and Template Method is compelling, and the first lab of this chapter is simple and makes a good case for Template Method. Observer is usually a hit, too, and here again there’s a lab devoted to reinforcing this pattern and best practices around it.

MVC is where things usually get tougher. It’s a great and very successful pattern, but it’s challenging to really understand at first: it’s more complex than most, it’s hard to understand that it can scale up or down in granularity, and the relationship to Observer also takes some consideration. It’s not easy to capture this pattern in a dead-simple example, and so both the example and ensuing lab are quite detailed, dealing with many layers and classes and relationships in the HR case study. The instructor should make a point of studying the HR application in detail before trying to cover this material; in fact if you prepare on one thing for this class, make it the case-study code and its MVC system. You’ll be lost in the details otherwise.





Note also that the style of code listings in the MVC example is unusual for our courses: because the source files are so large and complex, we take a different approach to providing context for highlighted code. Each listing shows a hierarchy of top-level class, inner class(es), and method name, before including the key lines of code. This should be simple enough to understand once the notation is recognized, but it's a departure and is probably worth pointing out to students at the beginning of the example.

The first of the design exercises at the end of the chapter – called “Auditing” – is about recognizing patterns. This makes instructions a bit tricky; we want to give hints but not say too much and give it away! If a student or group seems stuck, you might drop this hint: comb through the chapter looking only at the discussions of warning signs, and one of the warning signs will leap right out at you. (Just between us chickens, it's the fact that a window is directly triggering a refresh of another window after changing application state – the second warning sign mentioned for MVC.)

Chapter 4:

A bit like the previous chapter, this one starts with a couple of patterns that (usually) are immediately appealing, and then hits a toughie. Composite has already been covered, and Adapter has a nice practical feel to it. Decorator is another story. Some students take to it immediately, and the Streams example certainly proves that the pattern is worthwhile. But Decorator is a challenging one, to recognize and to apply well: when it's right, it's really right, but when it's wrong it's horrid. It's not a bad thing to tread lightly on this one, especially if teaching to less experienced programmers.

Note that this chapter's one lab goes heavy on collections and generics. If the students know Java 1.4 but not 5.0, this lab will be the hardest for them, by far. On one hand the lab is supposed to be about Adapters, not generics! On the other, in Java 5.0 it's not really good form to use the old weakly-typed collections, so we went with full use of generics, including wildcards in the code that students have to write. There's some spoon-feeding for the trickiest parts. By this point in the class you should have a good read on students' readiness for this. One fallback approach is to encourage them to do the lab full-out, but to leave the type parameters out of it. For example, instead of creating a new class `cc.util.ConsolidatedList<E>`, leave it at `cc.util.ConsolidatedList`; where the lab instruction says to define a `List<? extends List<? extends E>>`, just use `List`. This will trigger all sorts of warnings in the Java-5 compiler, but the code will ultimately still compile and run correctly.





Chapter 5:

This chapter departs from the flow of the course so far, jumping to J2EE and distributed systems, and also moving into more of an overview mode. We don't want to assume J2EE skills coming into the course, although it's common to see some Web application or maybe EJB experience among well-traveled Java programmers. We don't require a J2EE application server as part of the setup, and so no working J2EE examples are possible. Also, as the student guide points out, many J2EE patterns are implemented by containers or add-on frameworks such as Struts or Hibernate, so for instance a lab in implementing a Front Controller seems like a poor use of time. So we take our examples where we can find them, and the HR case study is helpful here, since many of the patterns in this chapter are really about multi-tier architecture, regardless of specific APIs like servlets and JSPs.

You should find that the final exercise at the end of the chapter is a nice way to wrap things up. Without it the course sort of ends with a whimper, but that final "big picture" seems to make each of the patterns covered in this chapter more compelling by showing how they interact: it's really a picture of a classic J2EE application – minus any vertical details.





Errata

This section lists issues that have been raised with the course since the publication of the latest version. These will be addressed in the next revision of the courseware.

- In Chapter 4, p. 127: the constructor for the enum is shown incorrectly at public visibility; it must be private.
- The **Files** and **Health** examples use the JCE to sign text content. The keys in the prepared keystores for these examples will expire someday! If you find that the examples don't work, try regenerating the appropriate key (with names and passwords as given in the example instructions and code) using the **keytool**. We'll replace these stores with commands in the **build** scripts that create them fresh in the classroom, to fix this problem in the next release.





Feedback

We truly do welcome feedback, both of a specific nature (pointing out mistakes) and general suggestions. For the former sending email with a numbered list of corrections would be most helpful.

Please send feedback to:

Will Provost
Capstone Courseware
<mailto:provost@capstonecourseware.com>
www.capstonecourseware.com

