



Capstone Courseware, LLC

33 Boylston Street
Jamaica Plain, MA 02130

877-227-2477
capstonecourseware.com

Spring-MVC Web Applications

Version 2.5

Instructor's Guide



Overview

This course combines four main parts: an initial overview of Spring as a whole; several chapters of study on the Spring core module; a module on Spring MVC; and coverage of Spring persistence techniques.

The first module develops fluency with basic instantiation, configuration, and assembly of objects of various stripes using bean factories and application contexts. It also introduces other concepts that are useful to the Spring-MVC developer, such as lifecycle hooks, annotation post-processing, component scanning, and the validation framework. Lecture material considers practical applications in the web context, but exercises in this module are simple, standalone Java SE applications.

The next module lays out features and usage for Spring MVC. We start with an overview and introduction to the **DispatcherServlet** and its main henchmen, the **HandlerMapping** and the **ViewResolver**. A second chapter digs into those latter two choices and illustrates a few options in more detail. Then we tackle the controller hierarchy, gradually, over the next three chapters, with a final unit on advanced techniques including declarative error handling and interceptors.

In this version of the course, we are frankly somewhat caught between two possible practices: call them the **Controller** and the **@Controller**. Most of the world, as of this writing, still seems to be using the Spring-2.0 **Controller** hierarchy, with heavy use of **SimpleFormController** and some other feature-rich options. But the Spring framework, for better or worse, seems headed for more emphasis on annotation processing, and the Spring-2.5 **@RequestMapping** will be one highly visible expression of this trend. For now, we stick with the 2.0 approach as the main line of our presentation, but give a good bit of time to the 2.5 technique, up to an including a parallel version of the capstone lab such that students can choose to explore **SimpleFormController** or **@RequestMapping** approaches to get to the same outcome.

The final module treats persistence in Spring. There are two different targets here, pursued in different ways. For JDBC we emphasize a classic template-and-configuration strategy, while for JPA we recognize that Spring's main value-add here is in transactions and so we focus on a JPA-first approach that allows simple integration of JPA and Spring. Then a final chapter shows the advantage of Spring transaction management to either persistence technology.





Timeline

The following breakdowns are approximate, and every class will vary.

Day 1

2 hours	Module 1, Chapter 1
2½ hours	Module 1, Chapter 2
2 hours	Module 1, Chapter 3

Day 2

2 hours	Module 1, Chapter 4
2½ hours	Module 1, Chapter 5
2 hours	Module 1, Chapter 6

Day 3

3½ hours	Module 2, Chapter 1
1½ hours	Module 2, Chapter 2
3 hours, spans days	Module 2, Chapter 3

Day 4

1½ hour	Module 2, Chapter 4
3½ hour	Module 2, Chapter 5

Day 5

1½ hours	Module 2, Chapter 6
1½ hours	Module 3, Chapter 1
1½ hours	Module 3, Chapter 2
1½ hours	Module 3, Chapter 3

If time is tight, the best candidates for skipping or skimming are probably Chapters 2, 4 and 6 of Module 2; but with a course this wide-ranging of course it will usually depend on the student group and where their main interests lie.





Crimson and the XML Validator

For those classrooms not using Eclipse, it will be worth the trouble to introduce students to a simple XML authoring environment deployed with the course software. The setup guide requires that the Crimson text editor be installed to `c:\Capstone\Tools\Crimson3.70`; then, the lab installer bundles an XML validating tool from Capstone Courseware (just a command-line interface to Xerces, really) and a single file that installs custom hotkeys into Crimson. The end result is that Crimson acts as an integrated editor, parser and schema validator for XML files. Hit **F9** to parse for well-formedness and **F10** to validate against a schema that's referenced in the XML document using the **xsi:schemaLocation** attribute.

Students may find this helpful when creating and editing their beans XML files and web context configurations. Especially if you sense that students' XML experience is slight, take a few minutes to show them this environment, and suggest that they make a habit of saving with **F10** rather than simply **File|Save**; this will trigger schema validation of their XML files and alert them to any of the common syntax or validity mistakes that can otherwise become frustrating as they don't surface until after the whole application's been run or deployed to Tomcat.

Two unfortunate limitations: one is that, on the one hand, Spring's bean factories insist on a **schemaLocation** attribute, or they fail to load the XML. This is non-compliant with XML Schema, but fairly harmless as we prime all our files with this attribute. On the other hand, web application contexts will choke if the **schemaLocation** is there and can't be resolved at runtime! So for safety's sake we leave them out of the context configurations in all the web projects. This in turn means that **F10** won't be useful for these files; students can still hit **F9** to check basic syntax. If one suspects a schema-validity problem but can't quite track it down by eye, one can always add the **schemaLocation** phrases for hand-editing the files, remembering to comment them out before attempting deployment of the web application.

Ant Build Process

Though most students will be happy to let the **ant** command take care of things, some students (and most instructors) will want to understand the inner workings here a little better. Each web project in the course has its own **build.xml** and **build.properties** files; these rely on the master environment variable **CC_MODULE** to import targets and properties in a central directory, **%CC_MODULE%\Ant**. Information here defines a routine for building a Spring application; we won't document this in detail here, but rather summarize the interesting tasks.

Some projects – everything in the first module and some exercises in the third module – contain only standalone applications. In these, the Ant build will:

- Clean out and create a **build** directory
- Compile Java source files to **build\classes**

Then, the provided **run.bat** in each case will launch Ant again, but this time with a specific target that launches the application. Properties **app-class** and **args** inform this launch, though for many of the projects the settings are the same: the application class is simply “Controller” and there are no arguments. Some applications do accept command-line arguments, and in this case the **run.bat** will pass those from the actual command line along to the launched class. The necessary Spring module JARs are included in the class path in all cases.

Starting in the second module, and for some projects in the third module, Ant will build and deploy web applications to a Tomcat server. For these projects Ant will:

- Clean out and create a **build** directory, which becomes the root of a web application (cleanup is a little tricky since Tomcat likes to lock certain files)
- Compile Java source files to **build\classes**
- Create a staging area for assembly of a web application
- Copy resources from **docroot** to that area – this may include HTML and JSP, the **web.xml** and Spring configuration files, message resources, and custom tags – and copy compiled Java classes to its **WEB-INF/classes** folder
- Copy the appropriate Spring module JARs into **WEB-INF/lib**
- Build a WAR file from the assembled tree

Some of these projects have standalone applications in them as well, and in those cases there will be a **run.bat**.



Eclipse Overlays

Capstone Courseware provides an optional package of workspace and project files for Eclipse Ganymede for this course – also using WTP 3 and Spring IDE 2.2. (See the course Setup Guide for download URLs.) Instructors, use this package on your own initiative and at your own risk. You should have significant experience yourself with Eclipse, WTP, and Spring IDE 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 documentation inside the workspace itself for general notes on usage: when you first open the workspace you'll see a **ReadMe.txt** that directs you to deeper, HTML-based documentation. 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.

See also the module notes file in each workspace – e.g. **Spring Module Notes.html** – for specific matters related to using Eclipse WTP with this course.

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.





Teaching Notes – Module 1

Chapter 1

This chapter should accomplish two things: give students a straightforward overview of Spring, and prove the machine setups by building one each of the two primary types of applications in the course, which are a standalone application that uses a bean factory and a web application that uses a web context configuration. Along the way students can get familiar with the layout of course exercises. Try not to let inquisitive students bog you and the rest of the class down during the example; they'll see it again almost immediately in Chapter 2 and there will be plenty of opportunity to dive into the details.

Chapter 2

Here we start building the foundation of Spring IoC, which will gradually become central to course exercises over the remaining chapters, as it is central to most Spring development. At first the beauty of the bean factory, inversion of control, and dependency injection may not be obvious to everyone. By the end of the class it probably will, but in the meantime, try to promote the idea that while Java EE gives a set of IoC features to a select few types (servlets, EJBs, etc.), Spring is egalitarian: it gives these same features to any Java class. How nice to be able to configure any class you write with properties that might not be known at development time! It's the end of the properties file, ad-hoc system properties, and so forth, and much more elegant and powerful than any of those hoary mechanisms.

An interesting side note that you might make during or after the Singleton vs. Singleton exercise: in order to enforce singularity (or prototype behavior) even over the rules set by a Java developer, Spring has to carry out some privileged actions. For this to work, either there must be no security manager in place, or it must have a policy that allows the Spring codebase some special permissions. A rough list is here:

```
permission java.io.FilePermission "", "read";
permission java.io.FilePermission "-", "read,write";
permission java.util.PropertyPermission "*", "read";
permission java.net.SocketPermission
    "www.springframework.org:80", "connect";
permission java.lang.RuntimePermission "accessDeclaredMembers";
permission java.lang.reflect.ReflectPermission "suppressAccessChecks";
```

The lab in this chapter is not all that challenging, but it's the first one and should get everyone rolling without much trouble and build familiarity with the <bean> construct.



Chapter 3

The previous chapter focuses on instantiation, but does bow to the necessity to allow some basic configuration via <property>s in order to make the exercises at all interesting. In this chapter, we expand on both ideas, with options to configure via constructor arguments and also new ways to customize object instantiation using existing factory implementations. This starts to tumble out as a promotion of all the different ways in which we can build objects using Spring. It may seem a bit scattered after a while, or like overkill, but it's good to note that Spring is really just trying to be as adaptable as possible to various common object-model designs: existing factory methods and abstract factories, immutable objects that can only be configured via constructors, and so on.

Chapter 4

Now we start to perform dependency injection for real. With single-value configuration as we've seen it in earlier chapters already understood, this feature should come very easily. By the end of this chapter we should be able to construct arbitrarily complex graphs of objects.

This chapter also introduces two features that may land with a bit of a thud in some classes: autowiring, and auto-detection of annotated component classes. The previous chapter did introduce component scanning, so that we could consider **@PostConstruct** and **@PreDestroy** hook methods. But now we look at both of these features in detail.

Autowiring is controversial: used heavily by many in the Spring community but reviled and labeled outright bad practice at many companies. We try not to advocate one way or the other here, but simply to cover the feature completely and accurately.

Component scanning and annotation processing is a different choice entirely, although this and autowiring are often practiced in concert. Like autowiring, component scanning seems to tear away some of the declarative benefit of Spring; annotation processing is another matter but where the kingpin annotation seems to be **@Autowired** we do wind up in the same boat. It might be worthwhile to point out that other sorts of annotation post-processors can enable features that even the anti-autowire crowd would find more appealing, such as **@RequestMapping** for the web tier or **@Transactional** for DAOs.





Chapter 5

Here we expand on our configuration and assembly capabilities, with an emphasis on configuring collections and maps but also some further consideration of reuse mechanisms including **util:property-path** and **util:list/set/map**.

One note on the Policy demo: to accommodate the different concrete paths at which the code is deployed, code in other courseware modules often rely on the **CC_MODULE** environment variable, and various beans will on their own initiative expand the phrase “`{env.CC_MODULE}`” found in their property values by looking up this variable. But that seems a bit much for this exercise, and so the exact path of the code source must be entered when configuring the policy. This is simple enough and the path is shown in the demo instructions. But the demo answer will have a path from the **Examples** tree, so any copy-and-paste solution will need some additional editing. It also means that course deployments to non-standard roots (other than **c:\Capstone**) will require editing of the XML file, even for the answer code. Finally, note that this syntax is demanding: the trailing slash matters, and even incorrect case for the drive spec (“C” vs. “c”) will result in a failure to match the permission. Read and type carefully!

Chapter 6

Validation is indeed a core-module feature in Spring – but it is also clearly distinct from the IoC/DI/factory material in the previous four chapters. In the materials we applaud Spring’s recognition of validation as a core feature: not something tied to the web tier, for instance, though Spring MVC does a nice job of extending the core validation mechanism for HTTP request parameters, HTML forms, and web controllers.



Teaching Notes – Module 2

Chapter 1

This chapter attempts to introduce the Spring web module as simply as possible – which is not very simply, but still it should be manageable. Successive chapters flesh out the web module, but in this one we establish a “kernel” of the dispatcher servlet, handler mappings and controllers, view resolvers and views. The initial exercise refactors a servlet application – which means we don’t see much in the way of new features from the application during the demo, but it also helps us start on familiar ground and see what Spring MVC is at its most basic. The example at the end of the chapter should serve as a good summary of the basic concepts; this seemed necessary since we’ve thrown so many new and important pieces at the student in quick succession. The lab might be a bit grueling but it’s important to do that first Spring web app more or less from scratch, and by the end students should have a pretty good handle on the basic concepts and construction techniques.

Chapter 2

We’ve separated this chapter from the previous one to isolate what are really not fundamental techniques but options and variations. On the other hand, it is sensible for a Spring developer to at least be aware of these choices – different **HandlerMappings**, **ViewResolvers**, etc. – before getting too deep into a development effort. This is why this chapter appears here and not closer to the end of the course. Still, it is probably the best candidate for cutting, if students might benefit from the additional time spent on other chapters. Most of this material could be derived by API study after class – though the redirect techniques are probably worth a quick look in any event.

You may well have your own (and perhaps contradictory) advice to offer at the end of the chapter, in the “Which Way to Go?” discussion; so much the better. This is entirely subjective stuff, and if there were one best way to map URLs to controllers or to manage the dependencies between controllers and views, then presumably Spring would stick with that and leave out the others.





Chapter 3

This chapter and Chapter 5 are tightly linked, and it was tricky to figure out exactly how to divide and organize these topics. Basically we're trying to establish command objects first, and the full flow orchestration that comes with form controllers later on. Then, the **MultiActionController** needs a home, and this seems like the right place, especially as it gives some exercise in dealing with command objects and validators more or less manually.

MultiActionController is a bit of an odd duck, actually, living off to the side of the main line of inheritance in the servlet-MVC package. This and Chapter 5 are mostly a long walk down that inheritance tree, and that makes multi-action controllers a sort of side road.

And, they will seem that much less necessary once **@RequestMapping** is introduced, which it is at the end of this chapter. This is the first exposure to the Spring-2.5 style of annotated **@Controllers**. It's salted in here as a first look, and more complete treatment develops in Chapter 5. But from this point forward students will be aware of a basic shift in Spring web development practice that is underway already and will probably accelerate when Spring 3 is released.





Chapter 4

Here we focus on binding and validation in isolation from command controllers. We can't really segregate these topics completely, but at least conceptually this chapter should be self-sufficient. The only real concession is that, for binding to work in both directions as one would expect, a form controller and the custom tag library both need to be in play. We leave this as a clear hole in the story, forward-referencing the next chapter and even leaving this chapter's lab exercise in a less-than-complete state pending the introduction of Spring custom tags.

It's probably fair to say that the failure to handle lowercase values in the LandUse demo on editing enumerated types is a bit of a straw man. HTML `<select>`s and `<option>`s allow for labels vs. visible text and this is probably how the problem here should be handled. However, the ensuing lab exercise makes a more compelling case for taking control of enumerated-type binding, and so it seems like a good flow, though the sharper students may challenge that first demonstration.

One interesting side note for this chapter concerns JSF, which is of course a big competitor for Spring MVC. In JSF, "converters" not editors are used, and on one hand they seem to throw away JavaBeans' **PropertyEditor** standards. On the other hand, one can perhaps see why: JSF converters don't rely on a base class like **PropertyEditorSupport** and sport just two simple conversion methods, one for each direction. Arguably, Spring's property-editor registrars and more general IoC features make configuration of property editors more manageable than JSF's converters, which are easy to configure as part of a view but then this system doesn't scale up as well for large applications.

Spring vs. JSF validators is more a question of scope, since Spring validators target objects and JSF ones target individual values. Each approach has its advantages; I happen to prefer the object scope, and for practical reasons, but it's very nearly a matter of taste. The arrow does point to Spring regarding message localization and management. Finally, any discussion of web validation should probably include a mention of the Struts Validator – neither Spring nor JSF has anything quite as ambitious, and integrating it may be a matter of interest to advanced students.





Chapter 5

In this chapter, we try to tackle just one main job, which is managing HTML forms with form controllers – but it's a big job. The understatement of the whole coursebook may be that **SimpleFormController** isn't really simple. On the one hand this class is the crown jewel of the MVC library, and has great facility once you get control of it; on the other hand it's a remarkably hard beast to tame. The demos are intended to walk students through the many necessary steps so as to make the task not too daunting. Otherwise, this is one of those tools with which it's best to work from a successful example and start varying things, rather than build from scratch and watch things fail and fail until suddenly they work. Be sure you know these demos very well before tackling them in class.

We also revisit **@RequestMapping** and now give it a little more room to run. A point I try to make as clearly as possible at this point is that, with Spring 2.5, Spring MVC has introduced two new techniques at once: annotations and a new mechanism for making dispatch decisions. These are both rolled into the **@RequestMapping**, and it's important to recognize that the new mechanism is the real feature here. It happens to be delivered solely by way of a set of annotations, which is unfortunate. Ideally there will be an XML equivalent to this set of annotations, because the model itself is really elegant and manages to make the whole **MultiActionController/SimpleFormController** hierarchy seem quaint.

By the way, this may be the right place to point out a custom tag that's been in use in the course examples all along. There's enough going on in that chapter that it seems best to leave `<cc:springBean>` undiscovered for a little while; but if students ask, or once you reach this point in the course, you might want to dig into it and explain what it's doing and how it's doing it. The idea here is to bring Spring beans into a JSP – even if that JSP doesn't enjoy the services of the Spring MVC framework. So for the database wrapper in LandUse, and similarly in the Wholesale application, the JSPs use this tag to get access to an object that all the other components in the application load using the application context.

Chapter 6

This chapter closes out coverage of the web module, and it may be seen as a bit of a hodgepodge of advanced topics. But there is a clear theme – or, at least, there are a few! Exception handlers and interceptors more or less of a piece, and the context-and-lifecycle section seems to put the finishing touches on the idea that any JavaBean can have the first-class treatment of a contained object. But ultimately this is the stepping-off point, and towards the end of the chapter we're mostly telling students where they might look next as they continue to learn about Spring.



Teaching Notes – Module 3

Chapter 1

We introduce Spring's persistence capabilities in this chapter, and then focus on JDBC as a high-value example of the DAO-support and template approach taken to this and many other persistence technologies. The value-add of Spring persistence templates will be highest here, and probably at its lowest in the next chapter when we look at JPA. This is just an inverse measure of how sophisticated is the target technology. But JDBC templates do show a nice "win" in overall coding effort and especially in being much less error-prone.

Chapter 2

JPA is a good example of a trend noted elsewhere in the coursebook, by which Spring was originally intended to improve the experience of developing Java-EE software, but Java EE itself has improved over time, frankly leaving Spring less to do. JPA all by itself is pretty slick: the provider performs object/relational mapping; it's configurable to work to raw connection properties or server-configured data sources; the provider is pluggable; and it supports container-managed contexts and/or transactions. A far cry from JDBC!

So in this chapter we approach Spring's JPA support mostly with simple integration in mind. That is, where we might use Spring to make JDBC coding better/easier/stronger, we don't really get those advantages by using Spring for JPA. Transaction support, as shown in the next chapter, will be a "win," but this is more a matter of Spring standing in for an application server – such as where we'd use Spring for transaction management in a web server such as Tomcat. So there's no lab for this chapter, just a demo, and this really sets the stage for the lab in the following chapter.

Chapter 3

Here we get Spring's transaction managers involved, and this certainly is a big part of the value proposition of using Spring for persistence. The demo shows the basic usage over JDBC, and the lab does the same thing for JPA. The effects are different, though: in the JDBC case we see that everything functions okay beforehand, but recovers more cleanly from a key violation with transactions. The JPA application simply won't write to the database except with transactions in force, and so this application actually starts working properly only at this stage of the game.





Revision History

Version 2.5 updates the course for Spring 2.5, and expands to five days in the process. Major additions include:

- Coverage of component scanning and annotation processing via bean post-processors, including exercises in **@Component** and **@Autowired** annotations.
- Coverage of **@RequestMapping** and related annotations, in Chapters 3 and 5, including a new lab that is a parallel of the old Lab 5B: so 5B is now done in the Spring-2.5 style, and 5C is the same old lab using **SimpleFormController**.
- New Spring/JPA chapter and more on transactions as applied to JPA DAOs.

Version 2.0.1 is the initial release of the course, and is created with Course 117, version 2.0.1, as a basis. In responding to instructor feedback on that course, we've added several examples, demos, and labs, and have attempted to ease the way for students who found too many steep ascents in 117. Major changes are as follows:

- The DOS scripts and Ant build files now rely on the variable **CC_MODULE**, not **CC_HOME**. This is in keeping with other recent Capstone courses and has simply proven to be a more potent and maintainable approach. You shouldn't see any complications in this switch, as in both old and new versions use of the variables is baked into the existing files.
- Several example trees have been renamed and/or expanded. **Ellipsoid** is now **Ellipsoid/Web** so that we can explore basic IoC techniques in **Ellipsoid/Beans**. **Beans** was not real descriptive and is now **Bank/Beans**, and there is a small **Bank/Web** example as well. **Factories** is now **Parsers**, to be more in line nominally with **Transformers**. **Collections** now has two steps, with **Step1** being the old example and **Step2** bringing in some interesting techniques using the utility schema.
- The **Bank/Beans** example now has two additional steps, and these form the first exercise in the use of factories, with a new **BankManager** system in place.
- The old demo on factories was a bit much for some students, since it used JAXP parsers and not everyone knows those; this has been refactored to an optional lab.
- The **Transformers** example has been explained in some more detail, and extended considerably to create a major new lab in Chapter 3.



- The old lab on validators is now the second of two, with a new lab on building the basic **OrderValidator** as the primary exercise. Students mostly found the **ListOfBatchesValidator** to be too much all at once; now it's additional exercise in basic validation and also illustrates nested paths – which have gotten more treatment in the lecture material as well.
- The initial web demo in refactoring the **Ellipsoid** application has been enhanced a bit to make it more obvious how things are changing as Spring features are introduced.
- There is a new, very simple web application in **Bank/Web** that should help to cement the basic concepts of Spring MVC. This is an example at the end of Chapter 5 and provides work in property editors later as well.
- We've moved away from Apache Derby and now use MySQL 5.0 as the RDBMS for the last chapter on persistence. This brings the course in line with other Capstone materials, notably our JDBC and Hibernate courses. It also removes any need to restart Tomcat during that chapter's exercises; Derby could only support one connection in embedded mode as we were using it, and this made some of the later demos rather clunky.





Troubleshooting

If you run into any trouble with code exercises, the first and best thing to do is to double-check that the classroom machines have been set up precisely according to the course setup guide. Especially, the wrong version of a tool can cause significant problems; don't wander off-book in this way unless absolutely sure you can support the software that you prefer and that we haven't tested. Check environment variable settings carefully, too; these are the cause of a great many classroom glitches.

Below are some specific pitfalls that have come up in previous offerings of the course:

- If you set the **CC_MODULE** environment variable in an individual command console (and not globally, e.g. through the Windows Control Panel), you must be sure to start Tomcat from that console or from one that also has the environment variable defined correctly. This is necessary because some Java classes running in the deployed web applications will attempt to resolve the value of that variable, as a way of finding resources outside of their own web contexts. A good strategy is to set your environment in the console where you'll do your Ant builds; **start** another console, which inherits that environment, and change to the Tomcat **bin** directory in that second console. Use the second console to **startup** and **shutdown** Tomcat as necessary.
- Attempts to validate Spring beans XML files against the schema will succeed only if there's an Internet connection, since the **schemaLocation** attribute points to the public schema. If the classroom is not networked, you can reset this attribute to the schema cached locally in **%CC_MODULE%\..\Tools\Spring2.5\dist\resources**. (This change would have to be made for each XML file, which is a bit clunky, but once you've done it, it's easy to do again by copy-and-paste.)



Errata

The following issues have been reported since the most recent release of the course. They will be fixed in the next revision.

- In Module 2, Chapter 5, the “User Feedback” demo:
 - Step 2, the code has a typo: the **command** attribute should be **commandName**.
 - Step 14, the output is not exactly correct: the full property name is `org.springframework.web.servlet.tags.form.AbstractFormTag.commandName`





Feedback

We very much appreciate whatever feedback we can get on our courseware – especially from the instructor's perspective. Naturally, the more specific, the better, and we strongly encourage you to make notes on issues you may encounter in the classroom, whether they're typos, missing files, or suggestions for clearer language to explain a concept. We can't guarantee that we'll act on every suggestion, but we're aggressive about stamping out problems and try to be highly responsive. Hopefully this means that when you give us good feedback, you get a better course the next time you need to teach it.

Please direct all courseware feedback to

Will Provost
Capstone Courseware
<mailto:provost@capcourse.com>
877-227-2477

For anyone who's interested, we have a very informal defect-tracking system, based in Excel spreadsheets with columns to capture defect location, nature, status, and author feedback. Ultimately, feedback goes into these sheets, so if you want a template, we'll be happy to provide one, to facilitate the reporting process.

