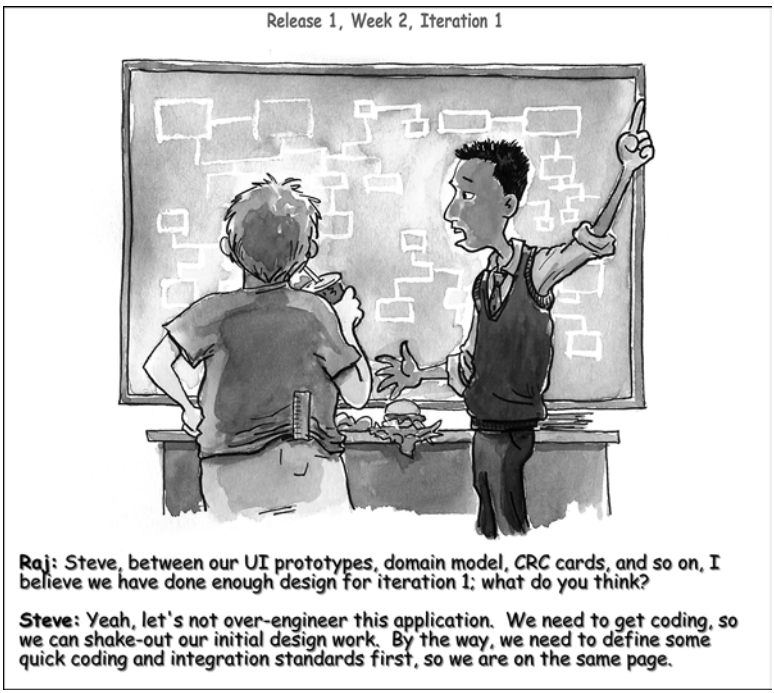
3

XP and AMDD-Based Architecture and Design Modeling



IN THIS CHAPTER, WE FINALLY BEGIN to get into the technology side of things, so now begins the fun part.

In a truly iterative development environment, all the architecture and design issues would not necessarily be finalized up front. Refactoring (improving code without impacting its functionality) plays a big role in constant improvement to the initially established design because invariably you will find better ways to do something when you are actually coding. Furthermore, while the scope of the project can be defined up front, the user requirements can continue to evolve from iteration to iteration versus having everything locked-down up front. With requirements, the idea is to have a lot of interaction with the stakeholder and be able to ask ad hoc questions.

Although some work can be done up front, such as the user stories, high-level architecture, user interface prototypes, domain model, standards and so on, other design issues can be resolved in the iteration they are applicable to. Furthermore, as we will see in Chapter 5, "Using Hibernate for Persistent Objects," and Chapter 7, "The Spring Web MVC Framework," writing tests first can also help with the design of your classes, so you don't have to have all the fine details of your classes figured out up front; in other words, you can take a just-in-time approach to design, so to speak.

However, some upfront design is bound to happen, perhaps in iteration 0 (perhaps when you are trying to demonstrate a proof-of-concept, which shows that the chosen technologies can work end-to-end, from the user interface to the database, for example).

Note

Also, in iterations 1 and 2, perhaps fewer user stories get coded because of the extra time required for design and environment setup work; this can include a domain model (explained later), definition of business objects, Java naming conventions, build/integration process/scripts for the team, and so on.

In this chapter, I hope to provide you with an end-to-end approach using modeling and process guidelines provided by Agile Model Driven Development (AMDD; agilemodeling.com) and Extreme Programming (XP; extremeprogramming.org).

What's Covered in This Chapter

In this chapter, we will accomplish the following architecture and design objectives for our sample application, Time Expression:

- Develop a free-form architecture diagram
- Explore objects using CRC cards
- Assemble an artifact I like to call an application flow map
- Develop class and package diagrams for Time Expression
- Establish our development directory structure and look at some sample file names (we will create in later chapter)
- Look at the steps we will follow in the upcoming chapters for end-to-end development of our screens
- List advanced concepts we will need to consider as our sample application evolves: exception handling, scheduling jobs, transaction management, logging, and more

Design Approach and Artifact Choices

In the previous chapter, we looked at an XP-based approach to defining business requirements and working with the customer. In this chapter, we will drill down into some minimal architecture and design to help us get going with building Time Expression, using popular technologies such as Hibernate, the Spring Framework, the Eclipse SDK, and many other related tools such as Ant, JUnit, and more.

If you have come across the myth that XP programmers don't design or document, I hope this misconception will be cleared up by the end of this chapter, because it couldn't be further from the truth. Let me give you a preview of what I'm talking about.

Take a look at Figure 3.1, which shows some possible artifacts you can produce at the release or iteration level. Release-level artifacts are ones you produce prior to a new release; iteration-level artifacts are ones you produce prior to each iteration. These aren't all mandatory for every project, so we can pick and choose the ones we need. However, between Chapter 2, "The Sample Application: An Online Timesheet System," and this chapter, I have chosen to demonstrate as many of these as possible, and practical, for Time Expression. At the end of this chapter, I will show you another diagram that will tie together all the artifacts produced as a result of our efforts between the previous and this chapter (but don't cheat by looking now, because it is a detailed diagram and I don't want to overwhelm you at this point).

At the very least, what you will see in this chapter will give you one perspective. This process might or might not work for you. However, there must be some things good about these methodologies, because developers love them, and I have seen many successful projects as a result of these methods. Also, in our case, the artifacts we will produce in this chapter are essential to the rest of this book, and this process will help get us there.

As you can see from Figure 3.1, we have a few artifacts to produce in this chapter, so let's move forward. However, before we do, I want to provide two perspectives from real-world users of XP.

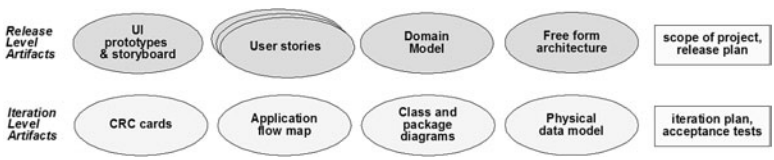


Figure 3.1 XP/AMDD style choices for artifacts to produce at release and iteration levels.

A project director working at a Fortune 50 company told me recently, "When we kick off an iteration, the first day of the iteration is usually spent reviewing stories and breaking them into tasks. The exercise of breaking them into tasks is truly a design session. What we ended up observing is that something like 20% of the developer's time, during an iteration, was spent in design. If you add all that time for all developers, across all iterations, it was a large number—which truly debunked the 'no design' comments."

To give you another perspective on the XP style of working, consider this statement from a senior architect at a well-established IT solutions company that has deployed more than a dozen successful projects using XP and AMDD techniques: "There is also another level of design that happens on an XP project which is at the daily level. Refactoring is a design activity. Although the iteration-kickoff design is an important step, it is the design work after the code is written that makes the difference between an OK design and a truly elegant one."

The difference with the XP approach, is that the architecture and design happens throughout the application's release cycle, not just up front. In other words, the application continues to evolve through the various iterations. The benefit of this approach is that the design is actually applicable to what you are building, not three to six months into development when the requirements could have changed—something that is certainly possible in our fast-paced and ever-changing world today.

Free-Form Architecture Diagram

Figure 3.2 shows the high-level architecture for our sample application. Note that this has been converted to an electronic version from the whiteboard version we saw at the end of the previous chapter. Converting it to an electronic format is a personal preference; you could just as easily take a digital picture of the whiteboard version, but I personally like clean and readable diagrams.

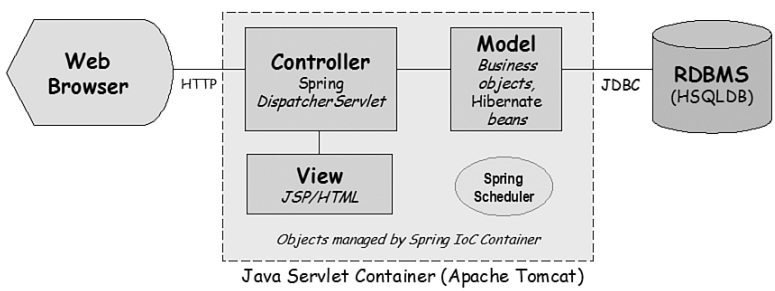


Figure 3.2 High-level architecture for Time Expression.

The architecture is fairly straightforward. We have our standard three-tier web architecture with the client tier (web browser), middle tier (application server), and our data tier (database).

Also standard is the use of the model-view-controller (MVC) design pattern, as you find in most Java-based web frameworks these days. The controller is the point of entry of the HTTP/web request; it controls the model and the view. The *model* deals with data, which is obtained by the controller and passed to the *view* for rendering in a presentable way. In our case, the view will be written using JavaServer Pages (JSP).

What makes our architecture interesting isn't that it uses a MVC pattern, but rather what's in the middle tier, namely the Spring Framework and Hibernate, two technologies we will cover in detail later in the book. Hibernate, as you will see later, makes database persistence very easy because you can reference database tables and records as plain old Java objects (POJOs). The Spring Framework (springframework.org) provides many benefits, as well, especially when you're working with POJOs. For example, we will use the Spring MVC for our web framework because it makes for cleaner code (when compared to something like Struts). Another notable feature of the Spring Framework is the support for scheduling jobs rather than depending on an external scheduling service such as CRON or the Windows Scheduler. Of course, the core feature provided by the Spring Framework is the inversion of control (IoC) functionality, which we will learn about in later chapters.

From User Stories to Design

We covered a variety of user stories in Chapter 2. For the sake of brevity, we will not develop every single user story in this book. However, the user stories I have chosen will given you complete end-to-end working examples of a form and a no-form screen. In addition, we will look at advanced topics, such as implementing application security using interceptors, sending emails, and scheduling jobs, which take care of a couple more user stories covered in Chapter 2.

In the rest of this chapter, I will provide examples based on at least the first two user stories, tag named *Enter Hours* and *Timesheet List*, in Chapter 2.

Exploring Classes Using CRC Cards

Figure 3.3 shows the domain model we established in Chapter 2. The domain model enables us to explore domain or business objects. The user stories will enable us to discover the web-based user interface controller classes. So, let's look at coming up with objects for the Timesheet List user story next, to see exactly how CRC cards work.

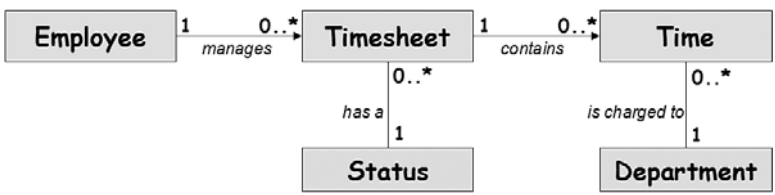


Figure 3.3 Domain model for Time Expression.

Figure 3.4 shows the Timesheet List UI prototype from Chapter 2. As I mentioned, we already know our user interface will be web based and will use the MVC paradigm. So, let's approach the discovery of our initial classes from the MVC perspective.

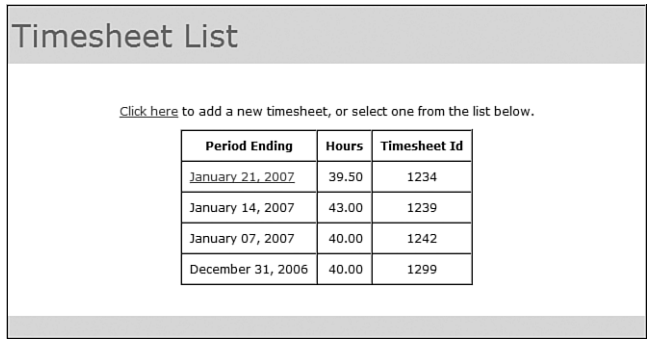


Figure 3.4 Timesheet List screen.

On the model part of the MVC, we already know some entity names for Time Expression from our domain model. For the controller part, we know the user story tag (*Timesheet List*, in this example) from Chapter 2. Given these, we can now proceed with our initial class design using CRC cards.

In case you have been wondering, CRC stands for class, responsibilities, and collaborators. Table 3.1 shows the layout of a sample CRC card along with some explanations for the three components you see there. Note that although I have shown an electronic version, CRC cards can actually be done on basic 3" x 5" index cards and later translated into a class diagram (if needed).

CRC cards provide an informal object-oriented technique for discovering interactions between classes. I like CRC cards because they can be used in an informal session with developers or users to discover objects without the need for a computer. Furthermore, CRC cards can be used to develop a formal class diagram, if needed (something we will do later in this chapter).

Tables 3.2 through 3.4 show some sample CRC cards for the actual classes we will develop later in this book, to meet the requirements for the Timesheet List screen.

Table 3.1 **A Simple CRC Card Layout**

Class Name (Noun)	
Responsibilities (obligations of this class, such as business methods, exception handling, security methods, attributes/variables)	**Collaborators** (other classes required to provide a complete solution to a high-level requirement)

Table 3.2 **Sample CRC Card for Timesheet Class**

Timesheet	
Knows of period ending date Knows of time Knows of department code	

Table 3.3 **Sample CRC Card for TimesheetManager Class**

TimesheetManager	
Fetches timesheet(s) from database Saves timesheet to database	Timesheet

Table 3.4 **Sample CRC Card for TimesheetListController Class**

TimesheetListController	
Controller (in MVC) for displaying a list of timesheets	TimesheetManager

We just covered some basics about CRC cards. For now, we have a good enough idea of what we need to move forward with the next step.

Application Flow Map (Homegrown Artifact)

In past projects, I have used a table similar to Table 3.5. This format is homegrown, in that it is something I came up with. I call it an *application flow map* because it shows me how a user interface will function (or flow) end to end. This technique also nicely maps the user stories to the *view* (the "V" in MVC), which maps to the *controller* and, finally, to the *model* objects.

Table 3.5 **Sample Application Flow Map**

Story Tag	**View**	**Controller Class**	**Collaborators**	**Tables Impacted**
Timesheet List	timesheetlist	TimeSheetListController	TimesheetManager	Timesheet
Enter Hours	enterhours	EnterHoursController	TimesheetManager	Timesheet Department

A Complementary Technique

In comparing this application flow map to techniques such as class diagrams or CRC cards, you will find that this map complements CRC cards and class diagrams. CRC cards list, among other things, responsibilities of each class, which is lacking in the application flow map. Class diagrams on the other hand, show relationships, cardinality, behavior (methods), attributes, and possibly more, which are more details than I like to have in this map.

By putting together classes in a textual and table format, we could also search for class names (in a large system, for example) and also sort these easily using a spreadsheet program or command-line utilities.

Extending the Application Flow Map with CRUD Columns

This table can also be altered for use with non-UI stories such as the Reminder Email: Employee user story. For example, the view and controller class columns can be replaced with a single column named Job, for instance.

Furthermore, you can extend this table by splitting the Tables Impacted column into four separate CRUD (create, read, update, delete) columns. This not only shows which tables are impacted, but *how* they are impacted, by the various collaborator classes. By adding CRUD columns, you essentially provide end-to-end flow of a user story (from the view to the database and back) in one row of our table.

UML Class Diagram

Next, we will look at a rudimentary class diagram. This is an optional step in my opinion (see sidebar on UML diagrams) because our CRC cards and application flow map provide us with enough information to move forward with coding. However, class diagrams can be a good thing when used appropriately.

Figure 3.5 shows a sample and minimal class diagram for the classes we have defined so far.

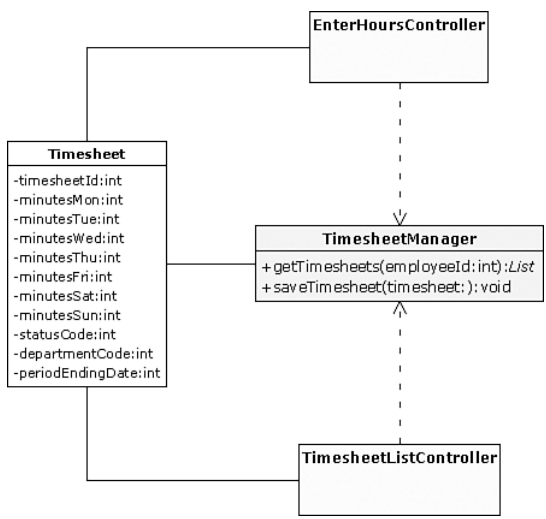


Figure 3.5 Sample class diagram for Time Expression.

Personal Opinion:

UML Diagrams

Over the years, I have used several types of UML diagrams, including the essential class diagram, package diagrams (my favorite), and the less-often seen deployment diagram.

Then there are ones that I am not a fan of, such as the popular sequence diagram. I don't like this diagram because I find it gets complex and cumbersome quickly. However, I'll be the first to tell you that I do not have better and alternative ways to do what some of these diagrams do (at least not yet anyway, but eventually I hope to because I'm currently conducting some research in better ways to model/diagram—check the visualpatterns.com website for updates periodically, if you are interested).

Meanwhile, I use UML diagrams when appropriate because I think they add value when used in the right place and at the right time. In fact, I think UML diagrams are most useful when generated using reverse-engineering tools to document the system already built (perhaps during a system handover).

I hope I don't come across as being dismissive about UML, because this isn't quite my intention, especially since it took a lot of work over a number of years from some intelligent people to make a standard such as UML even possible. (In fact, this is precisely why I'm basing all my research on work that has already been done instead of simply trying to reinvent the wheel.)

My main complaint about UML diagrams is that they get complex very quickly, specially for larger projects. Another issue I have with UML is that it requires special tools, which, because of software licensing costs, can be expensive for an organization. In addition, some of these tools can have a steep learning curve and hence require training for the people using these tools (one common example being Rational Rose), resulting in additional cost to the organization.

Furthermore, simpler tools such as OpenOffice.org, Microsoft PowerPoint, Microsoft Visio, and other similar tools provide the capability to connect a variety of shapes (rectangles, for example) using connectors, which are essentially straight or curved lines that connect two objects and stay tied to those objects when you move them around. This is a powerful feature because it enables you to create flowchart-like diagrams. I use connectors extensively, as you will see in many free-form diagrams in this book; in fact, almost all diagrams in this book were developed using OpenOffice.org!

Also, I tend to follow practices recommended by Agile Modeling, such as modeling with a purpose and producing good enough artifacts. Furthermore, I update these only when it hurts, because many artifacts can be thrown away after they have served their purpose. After implementing a design in code, you already have your documentation—yes, the code. (As I mentioned earlier, code can be reverse engineered to produce pretty class and other diagrams.)

What makes the idea of heavy documentation seem like sheer madness is the fact that I cannot recall one software development project where the documentation was maintained until the very end and matched the end product. This is the case because we live in a fast-paced world with sometimes unrealistic software delivery deadlines, and it becomes a difficult task to keep the documentation up-to-date.

In summary, use UML diagrams when appropriate, but don't be shy or hesitant about using simple, yet effective, free-form diagrams. Let me end by providing the same blurb from the agilemodeling.com website I provided in Chapter 2: "Your goal is to build a shared understanding, it isn't to write detailed documentation."

UML Package Diagram

For our sample application, Time Expression, we will use the prefix `com.visualpatterns.timex` for our package name.

If you have worked with Java already, you probably know that the first part of the package name is typically tied to an organization's domain name, just used backwards. For example, `com.visualpatterns` is the reverse for `visualpatterns.com`, which happens to be my website. The `timex` portion of our package name is derived from the name of our sample application. The remainder, the suffixes, for our package names are shown in Figure 3.6, a rudimentary UML package diagram.

> **Note**
>
> I have chosen very basic Java packages names to match our MVC pattern-based design. For example, we could have called our model package something like `domain`, but I prefer to match things up—for example, matching the package names with the architecture or application flow map. That way, someone new taking over my code can easily follow its organization. So, a fully qualified package name for the model package would be `com.visualpatterns.timex.model`.

As you might guess, the controller package will have controller-related classes in it. The job package will contain our email reminder job. The util package contains common and/or utility code.

Last but not least, the test package will contain our unit test code. Although I have chosen to place our test classes in a separate package, many developers prefer keeping the test classes in the same directory as the implementation code they are testing. This is a matter of preference, but in my opinion, having a separate package/directory for the test classes keeps things nice and clean in the actual implementation package directories.

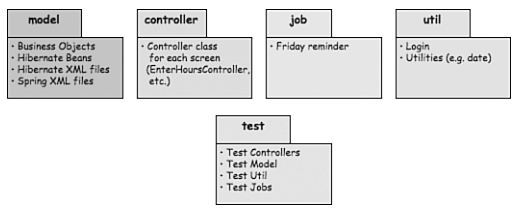


Figure 3.6 UML package diagram for Time Expression.

Directory Structure

Figure 3.7 shows the directory structure we will use for our sample application. This should look pretty straightforward and familiar; the most notable subdirectories here are src, build, lib, and dist. This figure will be referenced in later chapters (Chapters 4, 5, and 7, for example) and the directories relevant to each chapter will be discussed in a bit

more detail, when needed. Meanwhile, Figure 3.7 provides a brief description for all the key directories.

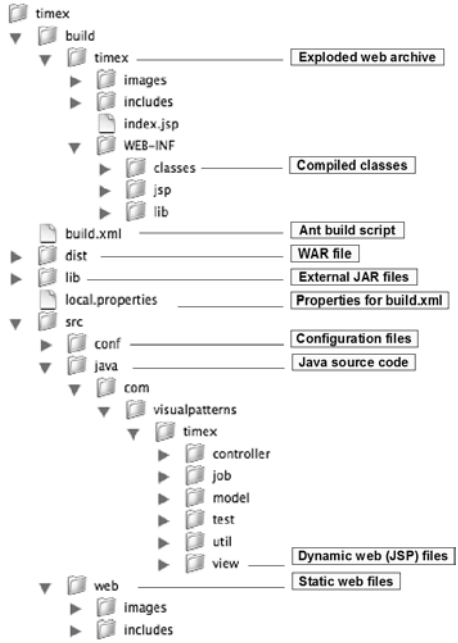


Figure 3.7 Development directory structure for Time Expression.

Sample File Names

Given our directory structure shown in Figure 3.7, we can now come up with some sample filenames for the classes we discussed in this chapter. For example, for the Timesheet List screen we discussed earlier in this chapter, we will most likely end up with the following files under the `timex/src/java/com/visualpatterns/timex/` directory:

- `controller/TimesheetListController.java`
- `model/Timesheet.java`
- `model/TimesheetManager.java`
- `test/TimesheetListControllerTest.java`
- `test/TimesheetManagerTest.java`
- `view/timesheetlist.jsp`

End-to-End Development Steps

Given what we have learned in this chapter so far, we can put together the steps that will be required to develop (code) for our first user story, from the web UI to the database, and back. Here are tasks that will most likely be required to complete the first user story:

- Set up our environment including the JDK, Ant, and JUnit (in Chapter 4)
- Write test and implementation classes for model package (using Hibernate in Chapter 5)
- Write test and implementation classes for controller package (using Spring Framework in Chapter 7)

Acceptance Tests

Acceptance tests can serve as our detailed requirements as they do in many Agile style projects. One example is a list of valid operations a user can perform on a given screen. The idea of using acceptance tests as requirements is feasible because these tests are something our customer expects our application to conform to. For our purposes, we will use them only for our unit tests; however, it is becoming more and more common in the real world to use acceptance tests as detailed requirements.

The following sections are our list of acceptance tests and something we will implement for the user stories we will develop. In the real world, these types of acceptance tests would be provided by the customer.

Sign In

- The employee id can be up to 6 characters. The password must be between 8 and 10 characters.
- Only valid users can sign in.

Timesheet List

- Only a user's personal timesheets can be accessed.

Enter Hours

- Hours must contain numeric data.
- Daily hours cannot exceed 16 hours. Weekly hours cannot exceed 96 hours.
- Hours must be billed to a department.
- Hours can be entered as two decimal places.
- Employees can view and edit only their own timesheets.

Other Considerations

As I mentioned earlier, we need to do just enough architecture and design to get us going. Although we did a reasonable amount of architecture and design in this chapter, there are a lot of things we haven't discussed yet but will in later chapters, such as the following:

- Application security—This will be covered in Chapters 7, "The Spring Web MVC Framerwork" and 10, "Beyond the Basics."
- Transaction management—This will be covered in Chapter 5. We will see how to programmatically implement transaction management using Hibernate.
- Exception handling—In Chapter 10 we will look at handled and unhandled exceptions and provide some guidance on when to use one versus the other.
- Other features—Features required for Time Expression such as scheduling jobs and sending emails will be covered in Chapter 10. Other topics also discussed in later chapters include logging, tag libraries, and more.

Big Design Up Front Versus Refactoring

According to Martin Fowler (refactoring.com), refactoring "is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior." Many developers have been refactoring code for years, but Martin Fowler gave it a formal name (and I'm glad he did).

As you begin to code an application, you will invariably find better ways to do things than you might have originally thought of (before coding began). For example, this could include removal of redundant code or cleaning up of code. Hence, I am a big believer that refactoring should always be an open option, not just for code but also for database design, architecture, documentation, build/integration scripts, and more. It also alleviates the burden of figuring out the entire design and process of an application up front.

For example, I recently came across a portion in an essay on the agiledata.org website, which helps summarize how I feel about this subject; this portion states that "Agile developers iterate back and forth between tasks such as data modeling, object modeling, refactoring, mapping, implementing, and performance tuning."

Take this book, for example. This is essentially a project for me as I'm developing a sample application from scratch and a book alongside it. Although I have done some upfront planning, I don't have 100% of the answers figured out, but I am not worried because I can refactor the architecture, design, code, or process used for Time Expression in later chapters because I want to make progress now instead of spending too much time trying to think of every possible scenario that could go wrong.

In short, you should definitely do some initial architecture and design, but keep in mind that if there is a way to improve something that adds value, such as simpler or cleaner code, and if it is not too late in the process (for example, the day of acceptance tests or deployment), you should go ahead and refactor!

Summary

In this chapter, we covered a lot of material and accomplished the following objectives we established at the beginning of the chapter:

- Develop a free-form architecture diagram
- Explore objects using CRC cards
- Assemble an artifact I like to call an application flow map
- Develop class and package diagrams for Time Expression
- Establish our development directory structure and look at some sample filenames (we will create these in later chapters)
- Look at the steps we will follow in the upcoming chapters for end-to-end development of our screens
- Review a list of advanced concepts we will need to consider as our sample application evolves

At the beginning of the chapter, I promised you a diagram to show you how we got here and some of the artifacts we produced along the way. (Did you cheat and take a peak already?) Figure 3.8 shows this diagram. Of course, this also clearly shows that XP has artifacts at various levels—conceptual, physical, and even in implementation. Note that the lines shown in Figure 3.8 are unidirectional because this is how we developed these artifacts in the previous and in this chapter. However, in the real world, these would be bidirectional because there is input coming from one direction and feedback going back in the opposite direction.

One parting thought on the subject of artifacts and documentation. Remember, that the database and code are the most important artifacts of all! I cannot emphasize this enough. The other artifacts we discussed in this book are merely ones you pick and choose, depending on your needs (these are not required). Furthermore, many of these optional artifacts could potentially be discarded after they have served their purpose, because most people don't update these anyway. However, code is always current because that is what the application for the customer is built with; the database can outlive all software programs written around it, so that should be considered the most important component of a system.

Speaking of database and code—now it is time to get our hands dirty and begin setting up our development environment using tools such as Ant and JUnit in the next chapters, so we can actually begin coding!

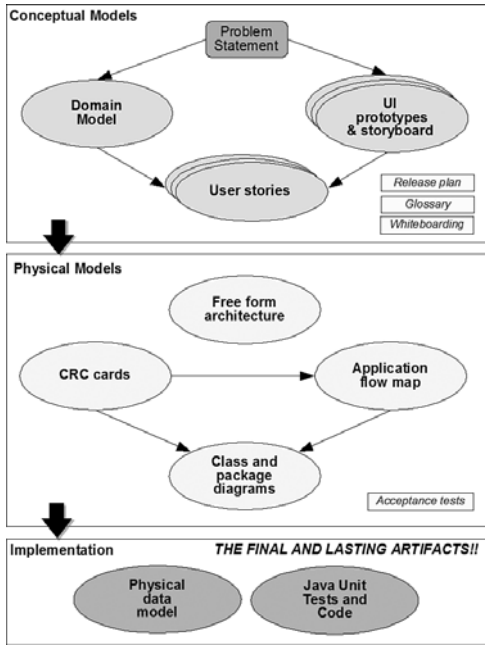


Figure 3.8 Conceptual, physical, and implementation artifacts for Time Expression.

Recommended Resources

The following websites are relevant to or provide additional information on the topics discussed in this chapter:

- Agile Model Driven Development http://www.agilemodeling.com
- Agile Data http://www.agiledata.org/
- Extreme Programming http://extremeprogramming.org
- CRC Cards http://c2.com/doc/oopsla89/paper.html

This page intentionally left blank