



Using Agile Processes and Modelling To Build Enterprise Applications

by [Anil Hemrajani](#)
05/09/2006

Abstract

Software development methodologies have been with us for a long time now. These methodologies typically tend to include a process, modeling techniques, or both. Some examples of the past and present methods include Structured Systems Analysis and Design Methodology (SSADM), Object Modeling Technique (OMT), the Rational Unified Process (RUP), and many others. Recent studies have shown us that the traditional approach of doing big requirements up front (BRUF) or big design up front (BDUF) using a waterfall approach such as SSADM not only can result in a significant waste of time and effort but also can cause many software developments projects to be challenged and/or fail entirely. This article looks at alternative approaches to BRUF and BDUF that can help minimize some of this wastage for enterprise software application development.

Most of material in this article has been taken directly from the book, [Agile Java Development With Spring, Hibernate and Eclipse](#).

The Problem

One organization that actively researches causes of failures in software development projects is The Standish Group. For example, its CHAOS report from year 2000 reflects that 23% of software projects in that year failed and 49% were challenged. This organization has been publishing similar reports since 1994. As another example, 1998 showed similar results, in that 28% projects failed and 46% were challenged. Another set of statistics published by The Standish Group is even more astonishing; these show the usage of features built into applications. According to The Standish Group, 45% of an application's features are never used, 19% are rarely used, 16% are sometimes used, 13% are often used, and finally, 7% are always used. These numbers simply boggle the mind because a big percentage of features built by developers are almost never used!

While this is just one source of information, it does match what I have seen in my 20-year career in information technology. Now that I have reviewed some problems with BRUF and BDUF, I'll look at the potential solution.

One Solution

The Standish group also provides ten factors for success. Called the CHAOS Ten, some of the top factors include executive support, user involvement, experienced project manager, clear business objectives, and minimized scope. My personal experience with cancelled, challenged, and/or failed projects matches very closely to these four success factors.

In 2001, seventeen methodologists came together to unify their methodologies under one umbrella; they jointly defined the term, *Agile*. The outcome of this was the *Manifesto for Agile Software Development*, a set of values and principles for these agile methods. The term *agile* incorporates a wide range of methods; these include Extreme Programming (XP), Scrum, Feature Driven Development, Agile Modeling, Crystal, and a few others. Many methodologies tend to include both process and modeling, since they often are complementary techniques.

Since agile methods tend to follow a common set of principles and values, one unpublished benefit of agile methods is that you have the option to pick and choose from various techniques and tailor them to your environment. In this article, I will do just that, as I look at using Extreme Programming (XP) and Agile Model Driven Development (AMDD) for enterprise application development. While XP provides a full software development lifecycle, AMDD provides guidelines for modeling. However, the focus of this article will be slightly more on AMDD than XP, since XP is a full lifecycle process with social aspects to it that are outside the scope of this article.

Using XP for a Software Development Process

XP is a disciplined and streamlined, full lifecycle development process created by Kent Beck, Ward Cunningham, and Ron Jeffries. There are entire books on XP, so I'll provide a summarized version of my interpretation of this method.

XP projects typically begin with the users providing requirements in the form of a few to several dozen *user stories* to make a good *release*. User stories are one- to three-sentence feature requests written by the customer (end user, business analyst, or another project stakeholder). The user stories are then prioritized by the customer and grouped into *iterations* (in a release plan) with the highest priority user stories developed first. The result of each iteration is a small release consisting of production-ready (that is, acceptance tested) code for the user stories implemented in that given iteration. Figure 1 shows a rather simplified visual representation of an XP lifecycle.

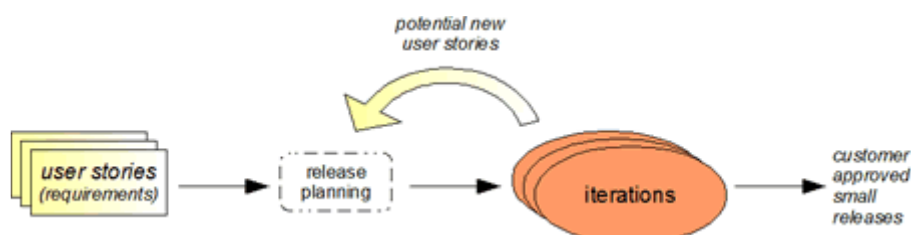


Figure 1. A simplified representation of the XP lifecycle

A common theme found in Agile methods is the notion of shorter and more frequent cycles for various aspects of software development projects. In the XP world, this includes the following:

- Shorter releases (for example, two to three months)
- Shorter iterations (for example, one to three weeks)
- Shorter build times (for example, ten minutes)
- More frequent integrations, anywhere between once to a few times per day, or even automatic continuous integration every time code is checked in

The other important aspects of XP include having the development team (developers, testers, project managers) in the same room with the customer or a representative, on-site. The idea is to have easy access to each other, since communication is a key factor for success on XP projects.

XP also emphasizes simplicity. For example, Kent Beck states in his book, *Extreme Programming Explained* (2nd Edition), "What is the simplest thing that could possibly work?" For designing software applications, this is reflected in the form of incremental design or as Scott W. Ambler (Agile Modeling) likes to call it, "just-in-time design." The idea behind this is to do some big-picture architecture and design up front but leave the detailed design for when it is needed, for example, during an iteration.

There are many other aspects to XP that I have not covered here in the interest of brevity; some of these include pair programming, sustained pace, collective code ownership, and so on. Also, it is important to note that XP isn't an all-or-nothing proposition. In other words, you don't have to adopt all of XP's practices in your project. For example, in a recent email conversation with Mike Cohn (author, *User Stories Applied*, Addison-Wesley), he wrote, "I encourage teams to start with Scrum but to then 'invent XP' meaning I want them to come up with the appropriate XP practices for their environment." In short, if XP is something you are interested in using, I encourage you to research XP either via related books or Web sites such as extremeprogramming.org, xprogramming.com, and xp123.com.

Next, I will look at some modeling techniques.

Using AMDD for Modeling Guidelines

According to thefreedictionary.com, a model is "A preliminary work or construction that serves as a plan from which a final product is to be made ... used in testing or perfecting a final product."

In this article, I use the word *model* loosely instead of using words such as *diagrams* and/or *artifacts*, since it also ties into the term, *model driven development (MDD)*, explained later in this article.

[Agile Modeling](#), created by Scott W. Ambler, is a set of values, principles, and practices for modeling in an agile manner. [AMDD](#) is a subset of Agile Modeling and provides effective (and agile) practices for Model Driven Development; one example of MDD is the Object Management Group's (OMG) Model Driven Architecture (MDA). Instead of creating extensive models, AMDD recommends creating "good enough" models. AMDD encourages modeling activity at two levels: initial modeling (for example, before an XP-style release begins), and model storming (for example, during impromptu design discussions in an iteration).

The initial modeling activity typically includes requirements and architecture-related modeling. For example, the requirements modeling may include usage models such as user stories, a domain model, and user interface (UI) models such as UI prototypes and a *storyboard* (also known as UI flow map). The architecture modeling can include a high-level architecture diagram accompanied with some initial discussions about scalability, reliability, security, and so on. I like to refer to the models produced from this exercise as *release-level models* since these tend to be higher level and conceptual in nature and impact the application as a whole, for that given release. For individual iterations, the models have more details and tend to be more physical in nature. Some examples of models in this category include CRC cards, UML class diagrams, and so on; these can be viewed as *iteration-level models*. Figure 2 shows some of the choices of models available to us using AMDD with an XP slant.

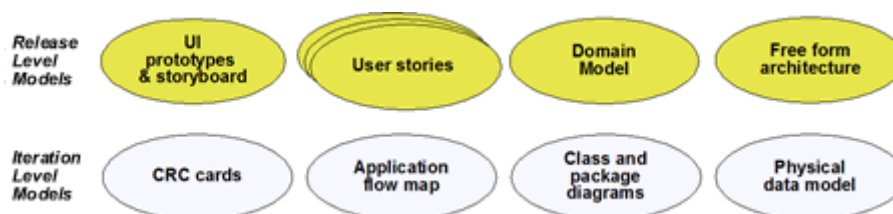


Figure 2. Choices of release- and iteration-level models

There really isn't much more to AMDD than this, so let's look at how you would use it in practice by reviewing some examples of release- and iteration-level models next. But first, let's define a sample application you can use to demonstrate your examples. Let's assume that a fictional organization wants to build a time sheet system to manage its hourly staff. To get things started, the customer may define a problem statement:

Problem Statement: Our employees currently submit their weekly hours worked using a paper-based time sheet system that is manually intensive and error-prone. We require an automated solution for submitting employee hours worked, in the form of an electronic time sheet, approving them, and paying for the time worked. In addition, we would like to have automatic notifications of time sheet status changes and a weekly reminder to submit and approve employee time sheets.

Next we may give our application a name. "Time Expression" seems like a catchy name, so let's go with that. Now, we are ready to move forward with some release-level modeling.

Release-level models

At the beginning of a given release, for example, a quarterly release, it is important to define the scope of the project so everyone is in agreement on what is within scope and what is not; this can easily be accomplished using a simple scope table, as shown in Table 1.

Scope	Functionality
Include	Time Expression will provide the capability to enter, approve, and pay for hours worked by employees.
Defer	Time Expression will calculate deductions from paychecks, such as federal/state taxes and medical expenses.
Defer	Time Expression will track vacation or sick leave.

Table 1. Sample Scope Table

Next, we could either start with a domain model or UI prototypes, depending on the nature and size of the application being built; for example, if the application does not have a UI, then having UI prototypes obviously makes no sense. In addition, which comes first also depends on what your organization gives priority to; for example, your organization may place more emphasis on domain models, so you may start there. However, other organizations find they can discover domain objects better and easier when working with users on UI prototypes.

A domain model helps to define major business concepts (entities) and the relationships among them, as demonstrated in Figure 3.

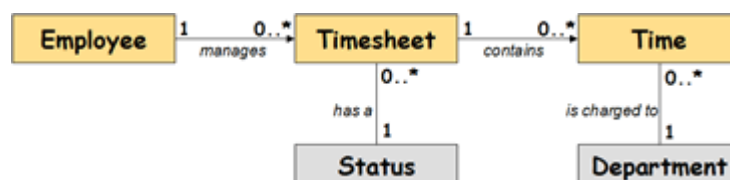
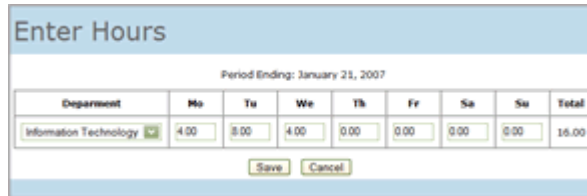


Figure 3. Sample domain model

User interface prototypes and storyboards are initial screen mockups to get a feel for how the customer visualizes the application; a sample prototype for a *Enter Hours* screen, is shown in Figure 4. A storyboard shows the flow of the screens, as demonstrated in Figure 5.



Enter Hours								
Period Ending: January 21, 2007								
Department	Mo	Tu	We	Th	Fr	Sa	Su	Total
Information Technology <input checked="" type="checkbox"/>	4.00	8.00	4.00	0.00	0.00	0.00	0.00	16.00

Save Cancel

Figure 4. *Enter Hours*— a sample UI prototype



Figure 5. Sample storyboard (also known as UI Flow Diagram)

Another model that we may wish to define at the release level is a high-level architecture diagram, similar to what's shown in Figure 6. This should have just enough details to get the project going with the design details to be determined at the time of iteration they are needed in. A high-level architecture diagram can help with discussions about security, scalability, transaction management, exception handling, and other important application-wide issues.

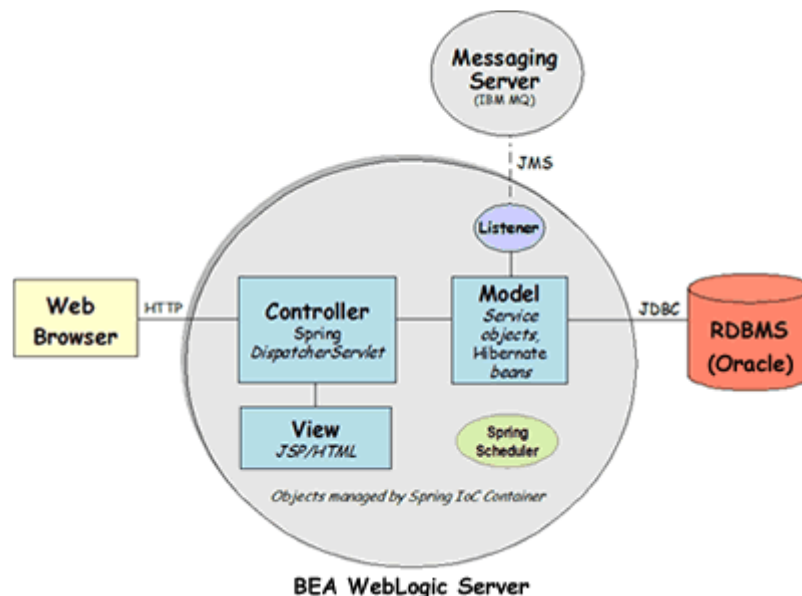


Figure 6. Sample high-level architecture

One other thing to define at the release level is a glossary, a list of common business and technical terms that may be used during the project. This is essential to a project for shared understanding of terminology. For example, in our sample application, these terms may include *period ending date*, *pending status*, and so on. In the XP world, this is also the point where an initial release plan (essentially a project plan) would be defined.

Iteration-level models

By now we have enough information about our application to move forward with our first iteration. Sometimes, projects may actually develop a proof-of-concept, for example, a demonstration of round-trip connectivity between the UI and the database for an application. (In the XP world, this would be considered a *spike solution*, a simple program to explore potential solutions.) These types of exercises (for example, proof-of-concept) can be viewed as being part of iteration 0 (zero), since no customer features are delivered during this iteration.

An obvious consideration of object-oriented development is to design and code objects. In the XP world, objects can be explored with one or more members of the team using CRC cards. This exercise of exploring objects can also be used to define class and method naming conventions, something known as a *system metaphor* in XP. Before we begin exploring objects, let's see what we know about our sample application so far. For example, we already know our domain objects (from our domain model), we know that our architecture will utilize a Model-View-Controller (MVC) design pattern, we know about some UI screens (from our prototypes), and we presumably have some user stories (perhaps, with short tags/names for each of them). Given some of these facts, we can begin naming our objects. So, if we begin developing our Enter Hours screen (Figure 4), we may have a class named *EnterHoursController*, which in turn may use a service (or model) object named *TimesheetManager*. The *TimesheetManager* class in turn would require a *Timesheet* domain object (as demonstrated in Figure 7).

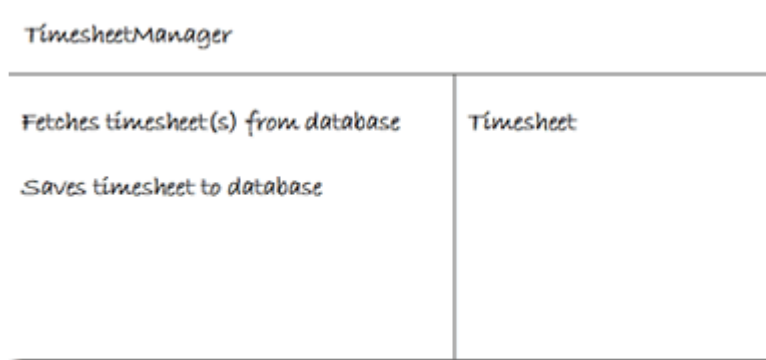


Figure 7. Sample CRC card

Once we have our basic objects explored using CRC cards, we could use something I like to call an *Application Flow Map* that shows an end-to-end flow of the user features. Figure 8 shows a sample application flow map; notice how we can see all the components of our MVC architecture. The other nice thing about this type of model is that it can be extended easily later in the project by adding columns for database tables affected by various parts of the application. For example, we could add four CRUD (create, read, update, delete) columns to show which database tables are affected and how.

Story Tag	View	Controller Class	Collaborators
Timesheet List	Timesheetlist	TimeSheetListController	TimesheetManager
Enter Hours	Enterhours	EnterHoursController	TimesheetManager

Figure 8. Application flow map

CRC cards and the home-grown application flow map may work well for some projects; for other projects, UML class diagrams may be the norm and/or you may feel more comfortable with those techniques as many developers do. In this case, class and possibly package diagrams can be used as a substitute or to complement CRC cards and/or an application flow map. Figures 9 and 10 show examples of class and package diagrams for our sample application.

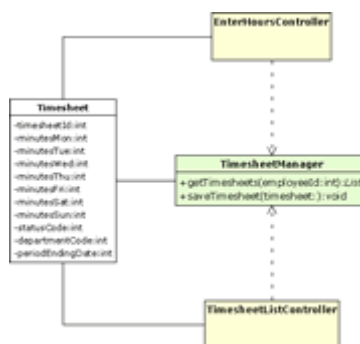


Figure 9. UML class diagram



Figure 10. UML package diagram

One other important thing to note about XP is the acceptance test. While user stories serve as high-level requirements at the release level, acceptance tests combined with ad hoc question-and-answer sessions with the user serve as detailed requirements at the iteration levels; acceptance tests are also used to write unit tests. For example, an acceptance test for our sample application may include something along the lines of "the maximum hours entered for any given day may not exceed 24."

Tying it all together

If you have come across the myth that XP programmers simply code without designing, you have surely noticed by reading this article that it couldn't be further from the truth. Furthermore, XPers consider refactoring a daily design since it constantly improves the code. To illustrate my point further, Figure 11 shows the various models we created at the release and iteration levels; these can also be viewed from another perspective, that is, as conceptual and physical models.

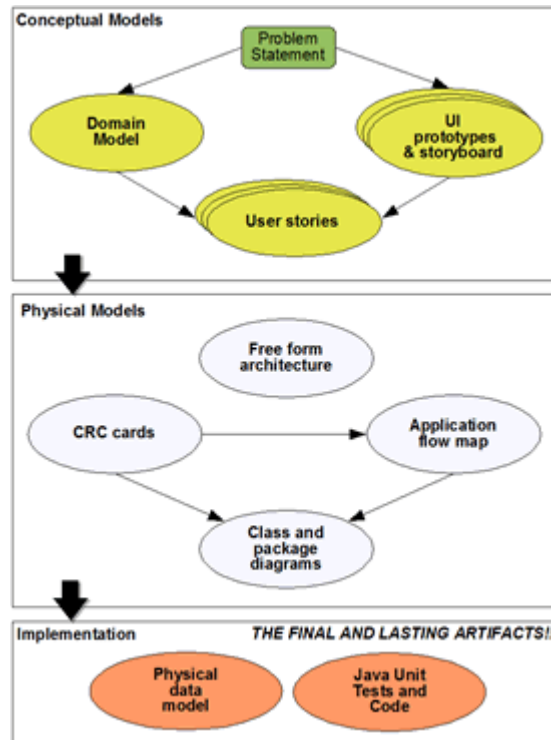


Figure 11. How the models fit together

The next steps most likely will include creating the actual database and, of course, writing code!

In my opinion, the database and code base are the lasting artifacts, and models can easily be reverse-engineered from both of these. All the other models we created up to this point could potentially be considered "throw aways" once the database and code have been implemented. The reason I mention this is because most projects I've been on fail to maintain the design and requirements documents anyway, so what's the point of keeping them around? Of course, some day OMG's vision of going directly from models to executable may come true (with or without an intermediary code-generation step); if it does, that would be absolutely wonderful because then the models would effectively replace the code we write these days and there would be no need to keep models and code synchronized.



Agile Versus Traditional Methods

On BRUF/BDUF style projects, more times than not, the requirements and/or design documentation becomes the goal toward the beginning of a project. This is typically not the case with agile projects. As a result, you may have noticed that the process and modeling techniques I discussed in this article did not require lots of formal and detailed requirements or design up front.

On the requirements side, using XP we start out with some general ideas of feature requests in the form of simple user stories, enough to do some release planning. The details are handled at the iteration level using acceptance tests as our detailed requirements during the applicable iteration. On BRUF-style projects, these may have all been handled up front in the form of detailed use cases with sections on pre-conditions, post-conditions, success paths, failure paths, business rules, and so on. As a result, BRUF-style projects often end up with a book of requirements that project stakeholders seldom read cover to cover.

On the technical (design) side of agile development, we can use an informal, high-level architecture to give us a sense of the enterprise application being built, enough to move forward with the first couple of iterations. In other words, we would not spend an enormous amount of time designing the physical database or creating detailed object relationship diagrams since they evolve over the course of a release. Furthermore, agile methods incorporate refactoring techniques to continually improve the design of an application. The rationale behind this is that many aspects of architecture and design can only get flushed out and/or proven once some coding has been done. In BDUF-style projects, the architecture design is typically finalized up front, before even a single line of code is written.

Conclusion

I just reviewed how to use a couple agile methods such as XP and AMDD to develop enterprise applications. These techniques aren't rocket science or entirely original, in that they build on years of what has been proven to work (or not). However, they do incorporate some of the techniques that enable you to work in a nimble fashion and that help you adapt quickly and *embrace change* in your organization, projects, and stakeholder needs. I would recommend using these merely as guidelines because when process, models, or documentation become your goal, you are losing sight of what really matters, the customer you are building the application for.

References

- Visit the homes of the [Agile Manifesto](#), [Agile Modeling](#), [Extreme Programming](#), [Refactoring](#), and [Agile Draw](#)
- [A Laboratory For Teaching Object-Oriented Thinking](#) provides an introduction to CRC Cards (from OOPSLA '89 conference)
- The material in this article is based on the book [Agile Java Development with Spring, Hibernate and Spring](#)
- [Comparison of Diagramming Methods](#)
- The [story](#) behind the Agile Manifesto
- [The Standish Group](#)
- [VisualPatterns.com XP and AMDD-based comics](#)



Appendix: A Note About Agile Draw

You may have noticed that many of the models used in this article use a free-form technique; this is no accident. The techniques used here are part of an effort called *Agile Draw*, a new technique backed by many published authors (including me) and many other experienced people. This technique is virtually notation-free and highly simplifies modeling, particularly at the conceptual level. The idea behind Agile Draw is to use the simplest tools possible and keep the notation to a bare minimum (for example, there are only three core shapes: circles, boxes, and lines). The emphasis is on "what" the models convey (that is, concepts and content) and not "how" it is conveyed (such as notations and tools). In addition, Agile Draw provides guidelines to take these models one step further, if needed, by adding "pizzazz" to them through colors, shadows, callouts, and more. These snazzier models can be used for presentations, proposals, Web sites, and so on. For details, visit the AgileDraw.org Web site.

*[Anil Hemrajani](#) is the author of *Agile Java Development With Spring, Hibernate and Eclipse*. He has 20 years of IT experience and has worked with Java Technology since late 1995 as a developer, entrepreneur, author, and trainer.*