ObjectiveView

A Magazine for the Professional Software Developer

Strata – Debra Stroud

Agile Development Special

Unit Testing

Turning Comments Into Code

Combining UML and TDD

Agile Retrospectives

Agile Modeling

published by

RATIO

Software Development, Training and Consultancy
www.ratio.co.uk/objectiveview.html for back copies

web distribution partner

ICONIX

www.iconixsw.com
Tel: +1 310 4580092
Fax: +1 310 3963454
Email: marketing@iconixsw.com

web distribution partner

ThoughtWorks®

www.thoughtworks.com

web distribution partner

Software Reality

www.softwarereality.com

web distribution partner

CPTTM

www.cpttm.org.mo

web distribution partner

Ambysoft

www.ambysoft.com

To subscribe for email delivery simply email:
objective.view@ratio.co.uk with subject: subscribe

ObjectiveView

CONTENTS

CONTACTS

Editor

Mark Collins-Cope
mcc@ratio.co.uk

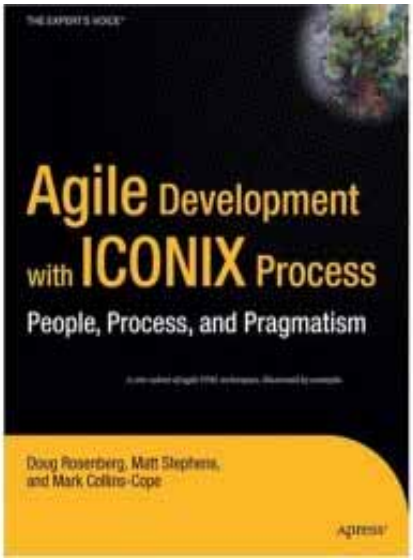
Free subscription
PDF by email – email to:
objective.view@ratio.co.uk
(with subject: subscribe)

Feedback/Comments/Article Submission
info@ratio.co.uk

Circulation/Sponsorship Enquiries
info@ratio.co.uk

Authors may be contacted through the editor. All questions or messages will be passed on.

advertisment



Agile Development with Iconix Process

A pragmatic collection of agile UML techniques, illustrated by example.

Available now from Amazon.com.

- Defines a core subset of agile practices and separates agile fact from fiction
- Describes how to get from use cases to source code using a minimal object modeling process
- Defines extensions to the core ICONIX Process, including: persona analysis/interaction design; and how to combine object modeling with Test Driven Development (TDD)

Effective Unit Testing

In an excerpt from her recent book – Effective Software Testing – Elfriede Dustin discusses the importance of unit testing…….

Unit testing is the process of exercising an individual portion of code, or a component, to determine if it functions properly. Almost all developers perform unit testing on some level prior to regarding a component or piece of code as complete. The process of unit and integration testing is instrumental to the delivery of a quality software product, and is often neglected, or implemented in a cursory manner. If unit testing is done properly, later testing phases will be more successful. There is a difference, however, between casual, ad-hoc unit testing based on knowledge of the problem, and structured, repeatable unit testing based on the requirements of the system.

To accomplish the goal of structured and repeatable unit testing, executable unit test software programs are developed, either prior to or in parallel with development of the software itself, that exercise the code in ways necessary to verify that it meets the functionality specified by the requirements, and to verify that it works as designed. These unit test programs are considered part of the development project and are updated along with the requirements and source code as the project evolves.

Executing unit tests ensures that the software meets at least a baseline level of functionality prior to integration and system testing. Discovering defects while the component is still in the development stage offers a significant savings in time and costs, since the defect will not have to be placed into the defect tracking system, recreated, and researched, but rather fixed in place by one developer prior to release.

The unit testing approach discussed here is based on a lightweight, pragmatic method of unit and integration testing that is applicable for most software projects. There are other, more complicated approaches to unit testing, such as path coverage analysis, which may be necessary for very high-risk systems. However, most projects do not have the time and resource budgets necessary to devote to unit testing at that level.

An important point to note about this section is that "pure" unit testing is not discussed. Pure unit testing is the practice of isolating a component from *all* external components that it may call to do its work, therefore allowing the unit to be tested completely on its own. This approach requires that all underlying components be "stubbed" to provide an isolated environment, which can be very time consuming and carries a high maintenance penalty. Since the approach discussed in this section does not result in the underlying components being isolated from the component under test, unit testing a component will actually result in some integration testing as well, since it will be calling lower level components to do its work. This approach to unit testing is acceptable, however, if those underlying components have also been unit tested and proven to be correct. Unit testing an upper level component without isolation is effective, since the components below it have been unit tested themselves and therefore should not contain problems. Any unit test failures are most likely to be in the component under test, not in the lower-level components.

Structure the development approach to support effective unit testing

Software engineers and programmers need to be accountable for the quality of their own work. Many view their job as producers of code who are not responsible for testing the code in any formal way, which, in their mind, is the job of the system testing team. In reality, programmers must be responsible for producing a high quality, initial product, which adheres to the stated requirements. Releasing code with a high number of defects to the testing team usually results in long correction cycles, most of which can be drastically reduced through the proper use of unit testing.

Although there is a slight concern that knowledge of the code could lead to less effective unit tests, this is generally not applicable if the component is performing specific functions that are related to documented requirements of the system. Although the unit test may exercise a component that only performs a small part of a functional requirement, it is usually straightforward to determine if the component fulfills its portion of the requirement properly. Aside from writing unit test programs, the developer will also need to examine code and components with other tools, such as memory checking software to find memory leaks. It may also be beneficial for multiple developers to examine the source code and unit test results, to increase the effectiveness of the unit testing process.

In addition to writing the initial unit test, the developer of the component is in a good position to update the unit test as necessary when modifications are made to the code. These modifications could be in response to

general improvements and restructuring, or in response to a defect or requirement change. Having the developer responsible for the code also responsible for the unit test is an efficient way to keep unit tests up-to-date and useful. In addition, depending on how unit tests are implemented, they could cause the build to halt, meaning it will not compile or produce a working executable, if the unit test program is part of the software build process. For example, suppose a developer removes a function, or "method", from a component's C++ interface. The unit test, which has not been updated, still requires the presence of this function to compile properly, and therefore fails to compile. This prevents the build from continuing on to other components of the system until the unit test is updated. To remedy the problem, the developer will need to adjust the unit test program's code to account for the fact that the method is no longer part of the interface. Here again, it is important for the developer to perform any necessary updates to the unit test program whenever the code is changed. Some software projects also opt to make successful unit test *execution*, not just compilation, necessary for the build to be considered successful. See later in this article for more information on this topic.

Unit tests will need to be written in an appropriate language that is capable of testing the code or component in question. For example, if the developer has written a set of pure C++ classes to solve a particular problem or need, the unit test will most likely also need to be written in C++ in order to exercise the classes. Other types of code, such as COM objects, could be tested from Visual Basic or possibly with scripts, such as VBScript, Jscript, or Perl.

In a large system, the code is usually developed in a modular fashion, by dividing functionality into several *layers*, each of which is responsible for a certain aspect of the system. For example, a system could make use of the following layers in its implementation:

- Database abstraction. An abstraction for database operations, this layer wraps up database interaction into a set of classes or components (depending on the language) that are called by code in other layers to interact with the database.
- Domain objects. A set of classes that represent entities in the system's problem domain, such as an "Account" or an "Order". Domain objects typically interact with the database layer. A domain object will contain a small amount of code logic, and may be represented by one or more database tables.
- Business processing. Components, or classes, that implement business functionality that makes use of one or more domain objects to accomplish a business goal, such as "Place Order" or "Create Customer Account".
- User interface. The user-visible components of the application that are used to interact with the system. This layer can be implemented in a variety of ways, but may manifest itself as a window with several controls, a web page, or a simple command line interface. This layer is typically at the "top" of the system's layers.

The above list is a somewhat simplified example of a layered implementation, but it demonstrates the separation of functionality across layers from a "bottom-up" perspective. Each layer will be made up of several code modules, which work together to perform the functionality of the layer. During the development of such a system, it is usually most effective to assign developers to work with a single layer, and communicate with developers, and components, in other layers through a documented and defined *interface*. So, in a typical interaction, the user chooses to perform some action in the user interface, and the user interface layer calls the business processing (BP) layer to carry out the action. Internally, the BP layer uses domain objects and other logic to process the request on behalf of the user interface (UI) layer. During the course of this processing, the domain objects will interact with the database abstraction layer to retrieve or update information in the database. There are many advantages to this approach, including the separation of labor across layers, a defined interface for performing work, and the increased potential for reusing layers and code.

There will typically be one or more unit test programs in each layer, depending on the size and organization of the layer's implementation. In the above example, a domain object unit test program would, when executed, attempt to manipulate each domain object just as if the BP layer were manipulating it. For example, the following pseudo-code outlines a unit test for a domain object layer in an Order Processing System that features three types: "Customer", "Order", and "Item". The unit test attempts to create a customer, an order, and an item and manipulate them.

```
// create a test customer, order, and item
try
{
        Customer.Create("Test Customer");
        Order.Create("Test Order 1");
        Item.Create("Test Item 1");

        // add the item to the order
        Order.Add(Item);

        // add the order to the customer
        Customer.Add(Order);

        // remove the order from the customer
        Customer.Remove(Order);

        // delete the customer, order, and item
        Item.Delete();
        Order.Delete();
```

```
        Customer.Delete();
}
catch(Error)
{
        // unit test has failed since one of
        // the operations threw an error – return it
        return Error;
}
```

Similarly, the BP layer will have a unit test that exercises its functionality in the same way that the user interface actually uses it, as in the following pseudo-code fragment:

```
// place an order
try
{
        OrderProcessingBP.PlaceOrder
        (“New Customer”, ItemList);
}
catch(Error)
{
        // unit test has failed since the operation
        // threw an error – return the error
        return Error;
}
```

As discussed in the introduction to this section, a natural result of unit testing a layered system without isolating underlying components is that there will be integration testing of the component under test and the associated underlying components, since a higher layer will need the services of a lower layer to do its work. In the above examples, the BP component will use the Customer, Order, and Item domain objects to implement the PlaceOrder functionality. Thus, when the unit test executes the PlaceOrder code, it is also indirectly testing the Customer, Order, and Item domain objects. This is a desirable effect of unit testing, since it will allow unit test failures to be isolated to the particular layer in which they occur. For example, suppose the domain objects unit test passes, but the BP unit test fails. This most likely indicates that there is either an error in the BP logic itself, or possibly a problem with the integration between the two layers. Without the domain object unit test, it would be more difficult to tell which layer in fact has a problem.

As mentioned earlier, unit tests should be based on the defined requirements of the system, using use cases or other documentation as guide. A functional requirement will typically have implementation support in many layers of the system, each layer adding some necessary piece to allow the system to satisfy the requirement, as determined in the design phase. Given this, each unit test for each affected layer will need to test the components to make sure they properly implement their piece of the requirement. For example, the order processing system described earlier might have a requirement entitled “Discontinue Item”.

To satisfy this requirement, the system will need to have a Business Process component that knows how to load an item and discontinue it, as well as check to see if any open orders contain this item. This in turn requires that the Domain Object and Database layers allow the Item object to be discontinued, perhaps through a Discontinue() method, and that the Order object supports searching for items in the order using an ID. As you can see, each layer participates in satisfying the requirement by providing methods or implementation.

Preferably, each requirement will have a representative test in the unit test program in each layer, where applicable, to demonstrate that the layer provides the functionality necessary to satisfy the requirement.

Using the previous example, the unit tests for each layer will then include a TestDiscontinueItem method that attempts to perform this requirement against the components in the layer that have functionality related to the requirement.

In addition to testing success cases of the requirement, error cases (also known as exceptions) should be tested as well to verify that the component gracefully handles input errors and other unexpected conditions. For example, a particular requirement states that the user must provide a full name, which, as defined in the data dictionary, should not exceed 30 characters. The unit test, along with attempting a name of acceptable length, should also attempt to specify a name of 31 characters to verify that the component restricts this input

Develop unit tests in parallel or before the implementation

Popularized by *Extreme Programming*, the concept of developing unit tests prior to the actual software itself is a useful one. Using this approach, it is necessary to have requirements defined prior to the development of unit tests, since they will be used as the guide for unit test development. Note that a single requirement will probably have implications on many unit tests in the system, and these unit tests will need to check the component for adherence to whichever part of the requirement it needs to fulfill. See earlier in this article for examples of requirement impact on a multi-layered system.

There are many benefits to developing unit tests prior to actually implementing a software component. First, and most obvious, is that the software will be considered complete when it provides the functionality required to successfully execute the unit test, and no less. In this way, the software is developed to meet the requirement, and that requirement is strictly enforced and checked by the unit test. The second benefit is the effect of focusing the developer’s efforts on satisfying the exact

problem, rather than developing a larger solution that also happens to satisfy the requirement. This allows the smallest possible solution to be developed, and will most likely result in less code and a more straightforward implementation. Another, more subtle benefit is that if there is any question as to the developer's interpretation of the requirements, it will be reflected in the unit test code. This provides a useful reference point for determining what the code was intended to do by the developer, versus what it is supposed to do according to the requirements.

To properly take advantage of this technique, the requirement documentation must be present and for the most part complete prior to development. This is usually regarded as the best approach, since developing prior to the completion of requirements for a particular function can be risky. Requirements should be specified at a somewhat detailed level, allowing for the required objects and functions to be easily determined[1]. From the requirement documentation, the developer can layout a general unit test strategy for the component, including success and failure tests.

To ease the development of unit tests, developers should consider an *interface-based* approach to implementing components. Good software engineering practice is to design software around interfaces, rather than around how the components function internally. Note that component or software interfaces are not the same as "user interfaces", which are intended to present and retrieve information from the user through a graphical or textual means. Component interfaces usually consist of functions that can be called from other components that will perform a specific task given a set of input values. If the function names, inputs, and outputs are specified and agreed upon, then the implementation of the component is a separate matter. Designing the interface first allows the developer to layout the component from a high level, focusing on its interaction with the outside world. It may also help development of the unit test, since the component's interface can be "stubbed", meaning the functions on the interface are written to simply return a hard-coded result, with no real logic. For example, consider the following interface:

```
class Order
{
        Create(orderName);
        Delete();
        AddItemToOrder(Item);
}
```

[1] The RSI approach to use case analysis is an effective way to document requirements from both the user and system perspective. For more information on requirements definition and RSI, see *Quality Web Systems*, chapter 2 or visit www.ratio.co.uk/rsi.html.

The functions present in this interface were determined by examining the requirements, which stated that the system must provide a way to create an order, delete an order, and add items to an order. For the purpose of writing the unit test, among other things, the interface can be stubbed, as in the following:

```
Create(orderName)
{
        return true;
}

Delete()
{
        return true;
}

AddItemToOrder(Item)
{
        return true;
}
```

As you can see, these interface "stubs" don't actually do anything useful, they simply return "true". Since the interface itself is valid, however, the benefit of stubbing a component is that a unit test can be written (and compiled, if necessary) against the interface, and will still work properly once the functions are actually implemented.

Unit tests can also assist in the development of the interface itself, since it is useful at times to see how a component will be actually used in code, rather than simply seeing it on a design document. Implementing the unit test may cause some refinements and other "ease of use" type enhancements to the component, since the process of implementing real code against an interface tends to highlight deficiencies in its design.

In practice, it may be difficult to always develop unit tests first, so in some situations, parallel unit test and implementation development is acceptable. The reasons for this are numerous – based on the requirements, it may not be immediately obvious how to design the best interface for the component, the requirements may not be 100% complete due to some outstanding questions, and other, non-requirement related, factors such as time constraints. In these cases, every attempt should still be made to define the component's interface as completely as possible up-front, and develop a unit test for the known parts of the interface. The remaining portions of the component and unit test can evolve as development continues on the component.

Updates to the requirements should be handled in a manner similar to that of the initial implementation. First, the unit test is modified with the new requirements, which may include additional functions on the component's interface, or additional values to be

taken and/or returned from the interface. In conjunction with unit test development, the interface is updated with the new parts necessary to allow the unit test to function, with stubbed implementations. Finally, the component itself is updated to support the new functionality, at which point the developer has an updated component that works with the new requirements, along with an updated unit test.

Make unit test execution part of the build process

Most software systems of significant size are comprised of source code that must be compiled[2], or "built", into an executable that can be used by the operating system. There are usually many executable files in a system, and those executables may use each other to accomplish their work. In a large system, the time it takes to build the entire code base can be quite significant, stretching into hours or days depending on the capabilities of the hardware doing the build. Many development environments are such that each developer must also build a "local" version of the software, on his or her own machine, and then proceed to make the necessary additions and modifications to implement new functionality. The more code there is to compile, the longer it will take to build, which is time that will be spent by each developer as they build their local versions of the system. In addition, if there is some defect in a lower layer of the system, it may not function properly, which could result in extensive debugging time by the developer to determine why the local version does not function properly.

As discussed earlier, unit test programs are a valuable way to ensure that the software functions as specified by the requirements. Unit test programs can also be used to verify that the latest version of a software component functions as expected, prior to compiling other components that depend on it. This will eliminate wasted build time, as well as allowing developers to pinpoint which component of the system has failed, and start immediately investigating that component.

In a layered software architecture, as described earlier, layers build upon each other, with higher layers calling down to lower layers to accomplish a goal, i.e., satisfying the requirement. Compiling a system such as this requires that a layer must be present, meaning compiled and ready for use, for the next layer up to successfully compile and run. This kind of bottom-up build process is common, and allows for reuse of layers, as well as the separation of responsibilities among developers.

[2] *Compiling* is a term used by most development environments to describe the act of producing an executable module from a set of source code, such as a collection of C++ files.

If unit tests have been written for the components in each layer, it is usually possible to have the build environment automatically execute the unit test program(s) after the build of a layer is finished. This could be done in a Makefile or a Post-build Step, for example. If the unit test executes successfully, meaning no errors or failures are detected in the layer's components, the build will continue to the next layer. If the unit test fails, however, the build will stop at the layer that failed. Tying a successful build to successful unit test execution can avoid a lot of wasted build and debugging time by developers, and it also ensures that the unit tests are actually executed.

It is quite common that unit tests are written initially, but are not updated, maintained, and executed on a regular basis. Forcing the build to also execute the unit test will ensure that these problems are avoided. This comes with a price, however. When project schedules are tight, especially during bug-fix and testing cycles, there can be considerable pressure to turn fixes around in very short time, sometimes in a period of minutes. Updating the unit test programs to allow the layer to build can seem like a nuisance, and possibly a waste of time at that particular moment. It is important to keep in mind, however, that the minimal time spent updating a unit test can prevent hours of debugging and searching for a defect. This is especially important if pressure is high and source code is being modified at a fast pace.

Many development projects use *automated builds* to produce regular releases of the system, sometimes on a nightly basis, that include the latest changes to the code. In an automated build situation, the failure of a component to compile properly will halt the build until the next day, until someone can rectify the issue with the source code. This is, of course, unavoidable, since a syntactical error in the source code must be examined and corrected by a developer in order for the build to proceed. Adding automated unit test execution to the build will add another dimension of build quality, above simply having a system that is syntactically correct and therefore compiles. It will ensure that the product of an automated build is in fact a successfully unit-tested system. This ensures that the software is always in a testable state, and does not contain major errors in the components that can be caught in the unit tests.

One of the major issues of unit testing is its inconsistency. Many software engineers do not follow a uniform, structured approach to unit testing. Streamlining and standardizing unit test programs is a good way to lower their development time and avoid differences in the way that they are used. This is especially important if they are part of the build process, since it is easier to manage unit test programs if they all behave the same. For example, when encountering errors, or processing command line arguments, the unit tests should be predictable. Using a standard for unit tests, it could be required that unit test programs all

return zero for success, and one for failure, a result that can be picked up by the build environment and used as a basis for deciding if the build should continue. If no standard is in place, different developers will probably use different return values, thus complicating the situation.

One way to achieve this goal is to create a unit test framework. This framework handles processing the command line arguments (if any), and reporting errors. Typically, the framework is configured at startup with a list of the necessary tests to run and calls them in sequence. For example:

```
Framework.AddTest(CreateOrderTest)
Framework.AddTest(CreateCustomerTest)
Framework.AddTest(CreateItemTest)
```

Each test (i.e., CreateOrderTest, CreateCustomerTest, and CreateItemTest) is a function that exists somewhere in the unit test program. The framework will then execute all of these tests by calling these functions, and handle any errors that they report, as well as return the result of the unit test as whole, usually pass or fail. Having a framework such as this can reduce unit test development time, since all that needs to be written and maintained in each layer are the individual tests, not all of the supporting error handling and other execution logic. These "common" unit test functions are written one time, in the framework itself. Each unit test program simply implements the test functions, and defers to the framework code for all other functionality, such as error handling and command-line processing.

Since unit test programs are directly related to the source code that they test, they should reside in the project or workspace of the related source code. This allows for effective configuration management of the unit test, along with the components themselves, which avoids "out-of-sync" problems. The unit tests are so dependant on the underlying components, that it is very difficult to manage them any other way but as part of the layer. Having them reside in the same workspace or project also makes it easier to automatically execute them at the end of each build. The reusable portions of a unit-testing framework, however, can exist elsewhere, and simply be used by each unit test program as they are built.

Summary

One of the most important ingredients of a high quality software release is effective unit testing, i.e. implemented by developers, ideally automated to be run after each nightly automated build, and then handed off to the testers can actually reuse some of the automated unit tests to expand upon. My experience has shown over and over again, that if unit testing is done properly, later testing phases will be more successful. Lack of developer unit or integration testing usually results in releasing code with a high number of defects and is often the cause for counterproductive, needlessly long correction cycles, and is known to result in missed release deadlines. Without unit testing in place, changing even the smallest module can be fraught with unknown implications and risks.

On the projects I have worked on where unit testing played a major role in the development lifecycle and was implemented efficiently and even automated, the system testing lifecycle was much more effective in the sense that System testing didn't have to get bogged down or stuck on unit testing issues that should have been solved in the earlier development phases. And by now almost everyone has seen the statistics that show the earlier in the development lifecycle a defect is found the cheaper it is to fix it.

Elfreide Dustin can be contacted via the Editor.

Subscribe for email delivery of

ObjectiveView

(in PDF format)

Email

objective.view@ratio.co.uk

with subject:

subscribe

Do check out back issues at:

www.ratio.co.uk/objectiveview.html

http://www.objectiveviewmagazine.com/

Retrospective agility – have you learned anything?

Tim Mackinnon, Agile Coach at ThoughtWorks, takes a looks at agile software development "retrospectives" …

Agile, Extreme, Adaptive… these words are cropping up everywhere, from the new car in a TV commercial to the software project down the hall. Everything is new and exciting, and everyone seems to be on the agile bandwagon. But are they all really in the same flexible world? In the keynote for XP2005, Jutta Eckstein pointed out to the audience, "If you are not holding retrospectives on your Agile software projects, you are not doing Agile projects!" [1]

What did she mean? For many this might seem a startling sentence, for others this might be "old hat" – but it boils down to observations made years ago by people we often quote but often fail to listen to. As Fred Brooks observed in the "Mythical Man-Month" (1975): "The techniques of communication and organisation demand from the manager as much thought and as much experienced competence as the software technology itself" [2].

Twelve years later, Tom DeMarco and Timothy Lister reminded us in "Peopleware" that "for the overwhelming majority of bankrupt projects we studied, there was not a single technological issue to explain the failure" [3]. They further expand with, "Whatever you name these people related problems, they're more likely to cause you trouble on your next assignment than all the design, implementation and methodology issues you'll have to deal with".

Just as Jutta points out, I believe that true agile projects are effectively using retrospectives to counter the effects of non-technical problems. This article describes some of the background behind retrospectives, as well as highlighting some common issues and solutions that I have learned from enabling agile projects over the past 6 years.

Agile respects people

With all of these warnings of doom and gloom, how are agile software projects faring? To consider this question it's worth briefly explaining the history of the term "agile". In early 2001, various originators and practitioners of different methodologies met to work out what it was they had in common. From this meeting they discovered that they "all emphasized close collaboration between the programmer team and business experts; face-to-face communication; frequent delivery of new deployable business value; tight, self-organizing teams; and ways to craft the code and the team such that the inevitable requirements churn was not a crisis"[4]. At this meeting they also coined the term "Agile", formed the Agile Alliance and documented what is now referred to as the Agile Manifesto [5].

Interestingly, in this manifesto, two of the four items prominently stand out as being people focused activities:

- **Individuals and interactions** *over* processes and tools
- **Customer collaboration** *over* contract negotiation

Of course these items seem like common sense, and I'm sure that many teams will quite happily quote the phrase "people over process" (item 1, in the list above). But referring back to the warnings of Brooks and DeMarco – how do you really cope with the people and the troubles they might be having with "the process"? How do you ensure that your customer understands the difficulties their demands might be imposing on the team (or in fact ensure that they are truly part of the team)?

Retrospectives help people

At ThoughtWorks, we have successfully used agile methods on many different sized projects throughout the UK, India, Australia and North America. We are big advocates of agile development, however we too have noticed that even with the best technical staff, the messages of Brooks, DeMarco and Lister constantly ring true. In a well oiled agile team, it is still possible to encounter "problems that seem to eat away at the moral fibre or the team" [6]. In fact many agile teams have been mindful of this kind of problem [7] as was Martin Fowler, our chief scientist at ThoughtWorks. In one of his articles from early 2001 he identified the usefulness of something called project retrospectives, adding: "Over time, the team will find what works for them, and alter the process to fit."[8]. This observation fits well with the idea of valuing collaboration, as well as individuals and interactions as mentioned in the Agile Manifesto.

So what exactly is a Project Retrospective? Norm Kerth originally described it as:

> *retrospective (rèt´re-spèk-tîv) -- a ritual held at the end of a project to*

learn from the experience and to plan changes for the next effort. [9]

Although originally Kerth described them "at the end" of a project, in Agile usage retrospectives are held anywhere from weekly to monthly to assess how well the team is working with regards to its process. In the aforementioned efficient teams, as per Brooks and Demarco, it's quite common for issues to build up that need a release. The act of taking Kerth's advice and taking the time to discuss "what has gone well", "what we should do differently" and "what puzzles us" in a structured manner is extremely helpful for the team to adjust its process or redistribute its resources more effectively. Normally, during a retrospective a team will build up a series of actions which they will prioritize and select from to implement in the weeks following it.

To achieve this kind of result, a certain amount of planning is required, something Kerth describes as a process akin to planning a menu. He suggests identifying a "starter", then moving on to a "main course" and finally finishing off with a "dessert". He classifies his exercises under these headings, as shown in the table below:

Starters	*Main Courses*	*Deserts*
I'm too busy	*Artefacts Contest*	*Making the Magic Happen*
Define Success	*Develop a Timeline*	*Change the Paper*
Create Safety	*Emotions Seismograph*	*Closing the Retrospective*
	Offer Appreciations	
	Passive Analogy	
	Session Without Managers	
	Repair Damage Through Play	

Selecting appropriate "dishes" for the three stages of the retrospective by understanding what the team needs, results in an activity that can help them learn what is most important.

Along with correct activities, I always find that it's important to make sure that everyone understands that this is not an opportunity to place blame. Many people shy away from retrospectives because of bad experiences in the past where the event simply degenerated into a "blame fest". A great retrospective is truly an opportunity to learn what things are going well (so that they can be repeated) as well as learning what things need to be improved. In this respect I always find it important to read out what Kerth calls the "prime directive":

"Regardless of what we discover, we understand and truly believe that everyone did the best job they could, given what they knew at the time, their skills and abilities, the resources available, and the situation at hand."

This sets the foundation for people to learn, and helps them understand that this is really an opportunity to learn.

While retrospectives are a useful practice for any team, in the case of agile teams using XP practices, retrospectives are particularly important. The tight relationship between the practices, how they support each other, and the principles and values that they are derived from leaves lots of room for mismatches to occur. If everyone is focusing on a particular practice then there is a high probability that something else is suffering in the development process and this is likely to be causing issues somewhere else. Being able to identify any other issues and actually talk about them in an open and honest manner is fundamentally important.

Learning isn't Easy

While there are many interesting and varied exercises [9] that can used in project retrospectives to encourage learning, there is one that stands out in importance, "The Create Safety Exercise". Ironically this exercise is probably the easiest to perform, but it's often skipped. Essentially it consists of the following steps:

1. **Make everything optional – stress that the process is not about finding fault but one of learning to improve for the next time.**
2. **Take a poll to find out how people feel. Ask people to privately vote on whether they feel safe enough to say what needs to be said.**
3. **Gather the ballots and tally them up for the group to see how safe they feel.**

For the second step, participants are asked to think of a number between 1 and 5 that indicates their level of safety, the scale we use is as follows:

5. **No Problem, I'll talk about anything**
4. **I'll talk about almost anything; a few things might be hard**
3. **I'll talk about some things, but others will be hard to say**
2. **I'm not going to say much, I'll let others bring up issues**
1. **I'll smile, claim everything is great and agree with managers**

The last item (smile and agree) is particularly important, as while it's a serious ranking, it usually breaks the tension of the introduction.

A participant of one of my retrospectives once came up to me and described a moment when he had been part of an offsite meeting for his company. In this meeting there was quite a serious agenda and many of the senior executives of the company were present to talk about "the issues". He described to me the tension in the air, and how he had longed for someone to step up and perform a safety exercise. He personally hadn't felt safe, and he also felt that not many others in the room felt very comfortable talking through the issues with those senior executives either. Sadly, the meeting had been a wasted opportunity with people playing lip service to those executives instead of talking honestly with them.

I always remember this story, and make it a point of performing this simple exercise and noting the results on the top of any records we take from the retrospective. Even when you think that it's not worth bothering to check safety, this simple exercise can give you some very useful insight.

For example, in a different situation I was helping facilitate a two day offsite meeting. Events at the end of the first day led to several attendees approaching the facilitators to explain their concerns about the direction of the meeting and how secure people felt. The following morning we worked with the organiser to help rebuild trust with the group. He was not convinced that we needed to repeat the safety check exercise of the previous day, he was positive that safety was low and he just wanted to address the group and explain the position he was in. However, we were keen to preserve the ritual of the safety exercise, the fact that it makes people consider how they feel, and that fact that the attendees looked to us to ensure that safety was maintained. We decided to go ahead with the exercise and were all surprised to find that safety wasn't as low as we had assumed. While there were a number of lower scores than the day before, the room wasn't full of 2's (I'm not going to say much). With this information in hand, the organiser was able to explain his reasoning much more effectively than if he had continued to assume that people weren't safe and weren't going to say much. This is an important lesson:

Always perform a safety exercise even if you think it's not necessary, the information you gain is often surprising.

As a final note, its worth mentioning that at times when I have experienced lower scores than expected it can sometimes be traced back to how you phrase the safety question. Kerth very carefully describes: "remark that there are managers in the room, and ask whether people feel safe enough to say what needs to be said". The "managers" reference in my experience can be misleading – there are many different reasons for people to feel unsafe. However the real emphasis should be on the "safe enough to say what needs to be said". This is particularly significant if you have new team members who might not feel that they can contribute to the content of the discussion. It is important to ensure that before you ask people to rate their safety, that they understand the agenda for the day, and that they also understand the optionality of the exercises (as Kerth describes) and finally, that they also understand that everyone can usefully contribute to the event even if they haven't been involved in the project from day one.

A new project member can equally describe how easy it is to pick up new things, or express confusion about why things are done in a certain way. Therefore I now ask people – "I want you to think about whether you feel safe enough to say what needs to be said about this project. We understand that different people have varied experience, but we are interested in knowing whether people feel able to share those different views with the people in this room, not whether everyone knows everything about the entire project."

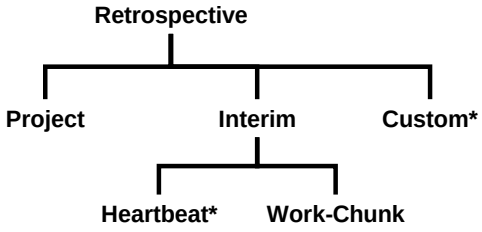
Dealing with the long haul

At ThoughtWorks we normally get called in to help larger software projects that have a life span of years rather than months. This is quite common, as software projects rarely just stop when they are delivered. There are often new features required by customers, who having understand what is possible, now feel more comfortable specifying what they would like for a second version. This is particularly true with agile development, where the iterative nature of delivering features that a user has identified as being most important, gives them more confidence in looking for those things that they would like next. Given this longer outlook, it becomes particularly important to consider the health of the team as it is developing and supporting the new features. Of course, "team" in this sense is not just the developers writing code, but it also includes the testers, any business analysts, and of course the users of the real system and any management sponsors.

In this mixed team it is important to ensure that everyone is able to communicate how they feel things

are progressing, whether they think there are improvements that would help make them more effective, and whether there are as yet unidentified opportunities that could improve productivity.

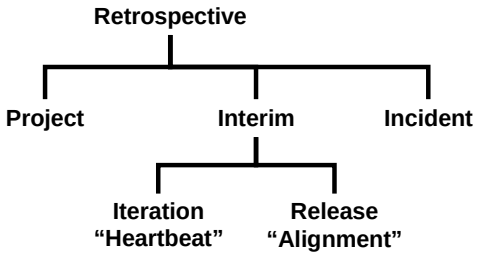
Interestingly, in the context of agile development, many teams have discovered that holding retrospectives more frequently is a key way to improving their ongoing process and preserving their long term effectiveness [6]. In 2003, I worked with fellow retrospective facilitators to categorize this difference. We came up with the following categorizations:



In the class of Interim retrospectives, the term "Work-chunk" never really became mainstream and this possibly reflects the difficulty we had in coming up with a generic term that people could agree on. I personally prefer the term "Milestone" or "Release" retrospective reflecting the idea that milestones or releases are ideal opportunities to hold retrospectives. However many in attendance at the meeting weren't happy with this usage as milestones often mean different things to different people. On the other hand, the term Heartbeat retrospective has become widely used, however not in the way that was initially proposed. The original intent was to reflect the idea that holding retrospectives in a steady monotonic fashion was important, like your heart beating. Your heart doesn't miss a beat, and my observation of 25+ monthly retrospectives (not tied to iterations, but to calendar), was that skipping one was equally as dangerous [6].

Over the past few years however, I have noticed that the term "heartbeat" retrospectives has come to take on the slightly different life of short, weekly retrospectives (typically held at the end of every iteration, lasting 30 minutes or less). The vision in people's minds is that you heart beats relatively quickly, with each beat being quite short. As this definition seems to have caught the hearts and minds of the community I think it makes sense to keep it.

Having tried lots of different styles of retrospective over the last few years, I have come to the conclusion that in the Agile world, there are still 4 styles of retrospective that each have their own particular use, however I would now draw the hierarchy with names as follows:



Project retrospectives

These are as defined by Kerth [9], and typically take 1 to 3 days depending on the project size, normally occurring at its end. As the duration is quite long, it is recommended that teams use an external facilitator.

Incident retrospectives

Rather than being pre-planned, these are differentiated by the fact that something unexpected has happened, and the team needs to learn from the experience with short notice. Ideally they are also externally facilitated.

In the category of Interim retrospectives, there are two types of retrospective that agile teams should definitely consider.

Iteration heartbeat retrospectives

Typically most agile teams use iteration lengths of 1 to 2 weeks (although scrum teams might use slightly longer), and following this short period of work the team is well suited to making suggestions about process changes. Typically these kinds of retrospectives are quite informal and last anywhere from 15 to 60 minutes. They can be self facilitated and the team ultimately derives a list of actions that they will implement in the next iteration.

It is this last point which raises a potential warning. I have encountered teams who have told me that they have experienced frustration in identifying actions which they haven't been able to deal with before the next heartbeat arrives. This made them feel overwhelmed and sometimes stressed, negating the positive effect of having the retrospective in the first place. My suggestion (and now current usage of these kinds of retrospectives) is to get the team members (before their iteration planning meeting) to each share 1 thing they thought went well that they want to repeat, and one thing they would do differently. We record these items on a flip chart and if appropriate quickly vote to determine which item they would like to see improved for the next iteration. During the following iteration planning meeting, I typically try to make sure that the team considers any of their observations when estimating or suggesting proposed implementations. Sometimes this might also suggest a card that can be played in the iteration.

Interestingly, I have also noticed that the product of Iteration heartbeats is typically only process related improvements, which aren't particularly profound or controversial. People are happy to make tweaks to their ongoing process but they need more time to consider larger issues.

Release alignment retrospectives

In the second edition of Extreme Programming explained [11], Kent Beck describes the idea of Quarterly releases and using this as a moment to reflect on product direction, team dynamics and goal alignment. This ties in perfectly with what I call Release "Alignment" retrospectives. Typically these retrospectives are planned well in advance and have a suggested duration of 3-4 hours (I've often been asked to reduce their duration, but teams just aren't able to cover the material and propose recommendations in less than 3 hours). These retrospectives are more in line with the menu of exercises described earlier. A typical agenda for one would look as follows:

- Create Safety Exercise
- Project Health through Pictures
- Project Timeline
- Review "what went well", "what didn't go so well"
- Actions
- Top Tips for Future Projects

The "Top Tips" exercise is of interest in that it stemmed from the observation that it was actually quite tricky and time consuming trying to share information between different project teams in larger organisations. Rather than trying to "mine" information from different teams, I decided to pose the question to each team as the challenge: "You have just won the lottery and are catching the plane to Hawaii in the morning. What tips would you like to leave for your replacement team so that they can continue your existing work, as well as be successful in any new projects?"

What differentiates this type of retrospective from its "heartbeat" counterpart is that team members have the time and opportunity to comment on more than just process related improvements. As teams become more accustomed to working with each other (and this make take time, requiring several retrospectives) they start to take this opportunity to talk about issues of team dynamics, personalities or failed approaches. For example one team raised the thorny issue of whether certain pairs were confusing refactoring with redesign, and using it as a mechanism to make un-agreed changes. These are much harder issues to confront however they open the door for real productivity improvements over and above the obvious process related issues that are often discussed weekly. Another team found that it didn't appreciate the skills of each of its members (and pre-work interviews showed people's misconceptions of each other). A Belbin team roles exercise [13] followed by an appreciation exercise [6][12] gave the team particularly profound insight into how each team member could positively influence the others.
As we noticed in [6], it is important to schedule these kinds of retrospectives even if you think that no-one has anything to say. It is often on these occasions that you "Make the Magic Happen" [9] and someone will reveal some profound insight. It is also during this time that the team can also chart a course for future work and how it will reorient itself to achieve that new direction.

Identifying opportunities for innovation

While retrospectives are good at helping teams adapt and cope, there is still a need to help them identify new opportunities for improvement. We have experimented with Gold Cards [10] as well as other innovation drivers that aim to deliver benefits to our teams and to ThoughtWorks. Our experience of using Gold Cards in some projects has indicated that this technique shouldn't be introduced too early in a team's development. When a project has first started, team members don't always exhibit the stresses indicated in the literature. Thus problems of "religious guilt" are not always prevalent until a team has reached a stable velocity and is efficiently creating business value above all else. When a team does reach this plateau however, Gold Cards are a very efficient and simple way of managing self improvement.

"Away Days" on the other hand are a more traditional way of ensuring that employees get an opportunity to learn and share new skills which they feed back into both their teams and company (in this case ThoughtWorks). For a consulting company like ThoughtWorks it's also important to get employees together to explore new ideas with their peers, as shared innovation in technical teams is also the secret to high morale [10]. Our CEO and founder is often known to attend most regional "away days" to search for the sessions that are crowded and bubbling with enthusiasm, as these are the ideas of the future that deserve more support.

In different offices (ThoughtWorks is a global company) we are also experimenting with other alternatives to foster our undeveloped ideas. "Innovation Half Days" are an opportunity for employees to sign up for and attend a mini "Open Space"[17] where they share and experiment with hot topics that need more development or exposure. Following the session we hold a "Heartbeat" retrospective to capture information that can be fed into both the topics as well as the process of the half day. We are optimistic that these experiments are paying off with time saving techniques and an energized work force.

Common things that went well

Over the course of working with many of our different Agile teams, there are many little gems that stand out. The following are some of the items that commonly crop up in successful agile teams.

Good team dynamics

- Everyone worked well together, and “gelled”, good teamwork
- From original team to new team - they picked it up very quickly.
- Not like other projects, always know who to talk to (and get good answers)
- Have a team that takes pride in its work

Handled change well

- Even with unexpected changes in the project, handled it well and “got on with it”
- Agile adapts well to changing requirements

Good agile attitude

- When changing priorities there were no complaints (patience)
- It encouraged users to focus on things in more detail

Standup meetings

- Shared good information efficiently
- Users involved in stand-ups and better understood the issues
- A positive way to start the day

Pairing

- The transfer of knowledge through sharing works well
- It results in better code
- Getting the best of two people’s designs
- The act of explaining helps you solve things more quickly

Good user relationship

- Responded to users needs and still worked well together
- We could step in and support users (they loved it)
- Users were often pleasantly surprised with the results
- Good conversation around different options & flexible solutions

Great continuous build environment

- Removed manual intervention – identical from dev to live
- Continuous deployment testing gave confidence
- Allowed rapid changes even at the last minute

Sitting with the team

- Prevented isolation and helped each other better understand
- Reinforced relationships between different team roles

Test driven development

- Writing automated acceptance tests for the website
- Separating concerns via Mock Object approach
- Made tests more readable (good names, clear intent)

Common things to improve

There are also many common gotchas that you need to look out for, however its important to encourage people to provide reccomendations along with those issues. I ask people to write on a red post-it something they "want to improve", along with a "reccomendation" stuck beneath it on a yellow post-it.

Time pressure

- Acute pressure, full on
- Learning new things is tiring too
- Not being able to complete full stories

Recommendations:

- Ensure you do release planning, and visibly track progress of releases and iterations

Test environments

- Too few environments available, shared servers causing grief
- Not enough access rights to servers (to automate things like data migration)

Recommendations:

- Make sure other departments are part of the team, and they attend standup meetings

Business involvement is more than you think

- Agile needs business involvement all the way through (it doesn’t tail off)
- Keep getting trivial feedback until users really have to accept the full system (This is a shame as agile is optimized for dealing with true feedback)
- Getting a formal commitment for user time (e.g. 1 or 2 days per week)

Recommendations:

- Invite users to the standup / hold standup convenient for them.

Refactoring

- Is there dead code still left? Have we really done enough?

Recommendations

- Rotate in new staff, fresh blood helps identify opportunities
- Adopt a can do/won't put up with it attitude

Business acceptance criteria

- Getting acceptance criteria that shows something is complete can be hard, temptation to plow on

Recommendations

- Don't rely solely on technology, requires business knowledge and involvement

Top tips for future projects

While these tips are often quite project specific, there are some that re-occur between projects. Here are some examples:

Nominate more than one user

- So you can ask questions from more than one person as they will invariably be too busy when you need them

Talk to other projects

- Ask for Retrospective Results from similar domains
- Speak to staff who have experienced real projects

Other tips

- Automate From Day 1
- Use source control for OS and Application configurations
- Have a single deployment script
- Do QA pairing

New techniques

While retrospectives are an important tool to guide the successful delivery of agile projects, we are also adopting, and in some cases pioneering, many new techniques. The following are particularly interesting ideas that we are using and experimenting with.

Giving an A

This technique was introduced to me at the "Retrospective Facilitators Gathering – 2005" and is a simple but effective idea that comes from the book "The Art of Possibility"[14]. In the chapter "Giving an A", the author (conductor of the Boston Philharmonic orchestra) explains that his music students were so terrified of their marks that they didn't experiment with their music and really learn how to play. Thus he came up with the idea that he would give every student an A. He announced this in his class, but there was a caveat. Each student had to write him a letter - dated next May, which described (in the past tense) how getting that A grade had changed their lives. They were also instructed to avoid using phrases such as "I hope", "I intend" or "I will".

Drawing from this idea, we have asked teams to write similar letters to their managers indicating why their team is a great place to work since he/she/they changed….

We have also used this idea in the ThoughtWorks "Quick Start" project inception workshops. In these workshops user groups were asked to write a letter to the project team, thanking them for the marvellous product which has changed how they work. In particular…

The letters that result from this exercise provide many potential solutions and are often very moving to read.

Futurespectives

This exercise is related to the "TimeLine" exercise described by Kerth [9], however I was helped to develop it by attendees of the "Retrospective Facilitators Gathering – 2005", with inspiration from "Giving an A" [14] and Luke Hohmann's "Remember the Future"[15]. Participants are asked to imagine that they have stepped into a time machine and have teleported to a time just after the completion of their project (which in reality is just starting). As the project was a success, its sponsors are keen to do a project retrospective (and so we are examining the future past).

As we know the project was a success, there are many successful events which should be recorded on the timeline (as green post-its); however there may also have been some things that potentially didn't go well (recorded as red post-its). Given that we know that the team always managed to overcome any difficulties – problematic events should always be followed by amazingly successful actions that overcame the difficulty (and the post-its on the timeline should demonstrate this with green ones following any red).

Once the future timeline has been created, participants are then asked to step back and mine it in a similar way to a normal timeline, "what went well", "what can we do differently" and finally what actions should we take for the upcoming project. Hohmann describes this as a mind trick that helps people overcome blockages that are limiting them from seeing potential solutions. Our experiments with this exercise have proved quite promising.

Appreciative Inquiry

This is an area that has piqued the interest of the retrospective community; however it is a topic that while deceptively simple, does require some training to get right. "The idea of the appreciative eye, assumes that in every piece of art there is beauty". And so, "Appreciative Inquiry suggests that we look for what works in an organisation. The tangible result of the

inquiry process is a series of statements that describe where the organisation wants to be, based on the high moments of where they have been. Because the statements are grounded in real experience and history, people know how to repeat their success" [16].

There is a lot of research that has gone into this technique and we are still investigating how to effectively combine it with our experience with retrospectives, however this is definitely an area to watch in the future.

Conclusion

There has been lots of literature written about the importance of team members collaborating and communicating with each other. While technical problems are rarely to blame for project failure, many approaches to software development choose to overlook this important people aspect of delivery.

Importantly, Agile approaches to software development have whole heartedly embraced this problem and made it part of their process for delivering running software. However, while it's very easy to quote the mantra "people over process" as outlined in the agile manifesto, it's another matter to actually implement the necessary steps and learn from them.

Learning cannot be rushed, and far from just running a quick project retrospective every week, you need to periodically and systematically take the time to select the appropriate exercises that will allow your team to reflect on the true problems that it might be harbouring. The process of discovering these little gems can prove extremely rewarding both in terms of increased productivity as well as long term sustainability for the teams you have assembled.

A learning environment is a happy and productive environment which is why company's like ThoughtWorks, and the clients we work with, are investing increasingly more resources into making sure that they learn from and incrementally improve the projects they partake in.

About the author

Tim Mackinnon is a senior developer at ThoughtWorks, where he coaches teams learning Agile Development as well as facilitating workshops and retrospectives.

References

[1] http://www.xp2005.org/GuestLectures (last visited, July 2005)

[2] F. Brooks, "The Mythical Man-Month", Addison Wesley, 1975-1999

[3] T. Demarco, T. Lister, "PeopleWare", Dorset House, 1987-1999

[4] Agile Alliance at : http://www.agilealliance.org/programs/roadmaps/Roadmap/index.htm (last visited, July 2005)

[5] Agile Manifesto at: http://www.agilealliance.org/programs/roadmaps/Roadmap/index.htm (last visited, July 2005)

[6] T. Mackinnon, "XP - call in the social workers", in *Extreme Programming and Agile Processes in Software Engineering*, Lecture Notes in Computer Science 2675, M. Marchesi and G. Succi, Eds. Berlin: Springer, 2003, pp. 288 - 297.

[7] C. Collins, R Miller, "Adaption XP Style", in Proceedings XP 2001

[8] M. Fowler, "The New Methodology" at, http://www.martinfowler.com/articles/newMethodology.html (last visited, July 2005)

[9] N. Kerth, "Project Retrospectives, a handbook for team reviews", Dorset House, 2001

[10] Higman, Mackinnon et al, "Innovation and Sustainability with Gold Cards", in Proceedings XP Universe, 2001

[11] K. Beck, "*eXtreme Programming Explained: embrace change*. 2nd Edition", San Francisco: Addison-Wesley, 2004.

[12] Nancy Kline. "*Time To Think*". Ward Lock, 1999

[13] R. Belbin, "Management Teams, why they succeed or fail", Elsevier Butterworth-Heinemann, 1981-2004

[14] B. Zander, R Zander, "The Art of Possiblity", Harvard Business School Press, 2000

[15] Luke Hohmann, interview at: http://www.enthiosys.com/wdocs/Interview-SoftLetter041015.pdf (last visited, July 2005)

[16] S. Hammond, "The Thin Book of Appreciative Inquiry", Thin Book Publishing Co. 1998

[17] H. Owen, "Open Space Technology", Berrett-Koehler Publishers, 1997

Tim Mackinnon can be contacted via the Editor.

recruitment advertisment

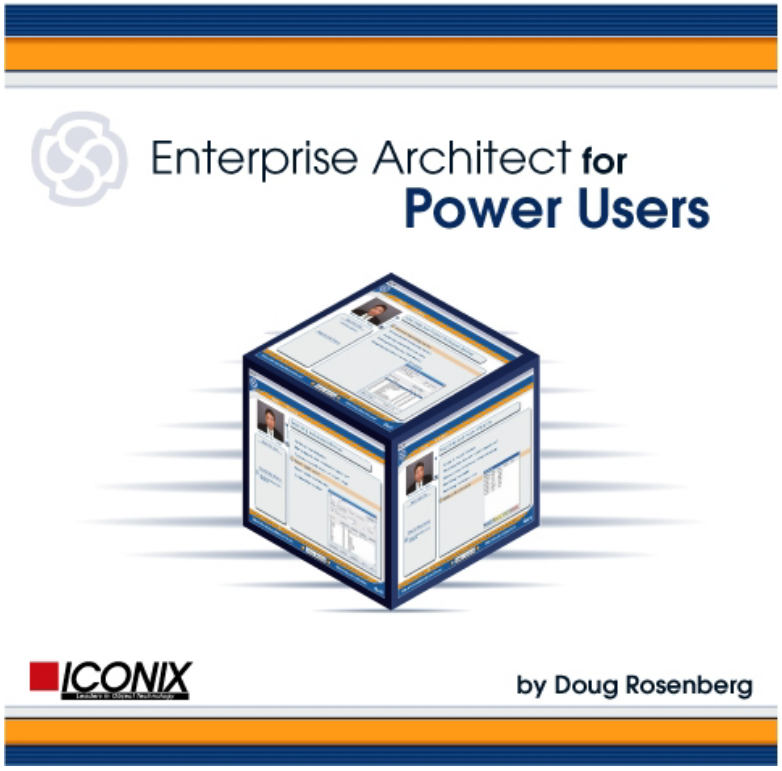
ThoughtWorks are Recruiting

ThoughtWorks is always interested in hearing from talented individuals interested in developing a career in technology consultancy. Our leadership in the practical application of Agile methods on enterprise-class projects allows our staff to deliver higher quality solutions more quickly and cost effectively, while giving client business leaders greater program and project control. We are currently hiring across all functions, from Java J2EE and C# developers through to Client Principals.

Be forewarned though: we're very selective. We only want the best and the brightest and our hiring process includes aptitude assessments and challenging interviews. To apply go to www.thoughtworks.com/uk/career/ onlineApplication.html.

To learn more about ThoughtWorks, visit our web sites at http://www.thoughtworks.co.uk or email work@thoughtworks.com.

Advertisment



Enterprise Architect for Power Users provides a comprehensive, yet easy to access tutorial on a wide range of advanced capbilities of the Enterprise Architect visual modeling software.

Topics include a discussion of how to set up EA in a variety of single and multi-user environments, and strategies to support projects of any size, how to set up EA to store models in a variety of popular version control systems.

There's a tutorial on UML 2 that explains what's new with this major new release of the UML standard, which is fully supported in Enterprise Architect. UML 2 topics include ports and interfaces on component diagrams, the addition of Timing diagrams to UML, and much more.

There's also information on EA's extensions beyond standard UML, for example, it's powerful built-in requirements tracking capability as well as EA's Code Engineering framework, which supports Forward and Reverse Engineering of source code, and Model/Code synchronization.

EA's Data Modeling capabilities are explained in detail, and the tutorial features over 30 narrated Step-By-Step demonstrations of specific features of the EA software, such as Importing DDL Schema

Order your copy today for just $149 from www.iconixsw.com or by phone 310-458-0092.
You can also order EA for Power Users together with Mastering UML with Enterprise Architect and the ICONIX Process for just $199. Or, try the ICONIX PowerPack which includes a both tutorials plus a copy of EA Corporate Edition for just $375.

Turning Comments into Code

Kent Tong (Tong Ka Iok) – Author of "Essential skills for Agile Development" – discusses how to turn comments into code.

Introduction

Consider a conference management application. In the conference, every participant will wear a badge. On the badge there is some information of the participant (e.g., name, etc.). In the application the Badge class below is used to store this information. Please read the code and comments below:

```
//It stores the information of a participant to be printed on his badge.
public class Badge {
	String pid;  //participant ID
	String engName; //participant's full name in English
	String chiName; //participant's full name in Chinese
	String engOrgName; //name of the participant's organization in English
	String chiOrgName; //name of the participant's organization in Chinese
	String engCountry; //the organization's country in English
	String chiCountry; //the organization's country in Chinese

	//************************
	//constructor.
	//The participant ID is provided. It then loads all the info from the DB.
	//************************
	Badge(String pid) {
		this.pid = pid;
		//************************
		//get the participant's full names.
		//************************
		ParticipantsInDB partsInDB = ParticipantsInDB.getInstance();
		Participant part = partsInDB.locateParticipant(pid);
		if (part != null) {
			//get the participant's full name in English.
			engName = part.getELastName() + ", " + part.getEFirstName();
			//get the participant's full name in Chinese.
			chiName = part.getCLastName()+part.getCFirstName();
			//************************
			//get the organization's name and country.
			//************************
			OrganizationsInDB orgsInDB = OrganizationsInDB.getInstance();
			//find the ID of the organization employing this participant.
			String oid = orgsInDB.getOrganization(pid);
			if (oid != null) {
				Organization org = orgsInDB.locateOrganization(oid);
				engOrgName = org.getEName();
				chiOrgName = org.getCName();
				engCountry = org.getEAddress().getCountry();
				chiCountry = org.getCAddress().getCountry();
			}
		}
	}
	...
}
```

Turn comments into code, making the code as clear as the comments

Consider the first comment:

```
//It stores the information of a participant to be
// printed on his badge.
public class Badge {
        ...
}
```

Why do we need this comment? Because the programmer thinks the name "Badge" is not clear enough, so he writes this comment to complement this insufficiency. However, if we can use this comment directly as the name of the class, the name will be almost as clear as the comment, then we will not need this separate comment anymore, e.g.:

```
public class ParticipantInfoOnBadge {
        ...
}
```

Why do that? Isn't writing comments a good programming style? Before the explanation, let's see how to turn the other comments in the above example into code.

Turn comments into variable names

Consider:

```
public class ParticipantInfoOnBadge {
        String pid;  //participant ID
        String engName; //participant's full name in English
        String chiName; //participant's full name in Chinese
        String engOrgName; //name of the participant's organization in English
        String chiOrgName; //name of the participant's organization in Chinese
        String engCountry; //the organization's country in English
        String chiCountry; //the organization's country in Chinese
        ...
}
```

We can turn the comments into variables, then delete the separate comments, e.g.:

```
public class ParticipantInfoOnBadge {
        String participantId;
        String participantEngFullName;
        String participantChiFullName;
        String engOrgName;
        String chiOrgName;
        String engOrgCountry;
        String chiOrgCountry;
        ...
}
```

Turn comments into parameter names

Consider:

```
public class ParticipantInfoOnBadge {
        ...
        //************************
        //constructor.
        //The participant ID is provided. It then
        // loads all the info from the DB.
        //************************
        ParticipantInfoOnBadge(String pid) {
                this.pid = pid;
                ...
        }
}
```

We can turn the comments into parameter names, then delete the separate comments, e.g.:

```
public class ParticipantInfoOnBadge {
        ...
        //************************
        //constructor.
        //It loads all the info from the DB.
        //************************
        ParticipantInfoOnBadge(
          String participantId)
        {
                this.participantId = participantId;
                ...
        }
}
```

Turn comments into a part of a method body

How do we get rid of the comment "It loads all the info from the DB" in the above example? It describes how the constructor of ParticipantInfoOnBadge is implemented (load the information from the database), therefore, we can turn it into a part of the body of the constructor, then delete it:

```
public class ParticipantInfoOnBadge {
  ...
  //************************
  //constructor.
  //************************
  ParticipantInfoOnBadge(String participantId) {
    loadInfoFromDB(participantId);
  }
  void loadInfoFromDB(String participantId) {
    this.participantId = participantId;
    ...
  }
}
```

Delete useless comments

Sometimes we may come across some comments that are obviously useless, e.g.:

```
public class ParticipantInfoOnBadge {
  ...
  //************************
  //constructor.
  //************************
  ParticipantInfoOnBadge(String participantId) {
                ...
  }
}
```

This comment is useless because even without it anyone can tell that this is a constructor. It not only is useless, but also takes up the precious visual space: A screen can display at most 20 and odds lines, but this useless comment already takes up 3 lines, squeezing out the useful information (e.g., code), making this code fragment hard to understand. Therefore, we should delete it as quickly as possible:

```
public class ParticipantInfoOnBadge {
        ...
        ParticipantInfoOnBadge(
          String participantId) {
                ...
        }
}
```

Extract some code to form a method and use the comment to name the method

Consider the first comment below:

```
void loadInfoFromDB(String participantId) {
                this.participantId = participantId;
                //************************
                //get the participant's full names.
                //************************
                ParticipantsInDB partsInDB = ParticipantsInDB.getInstance();
                Participant part = partsInDB.locateParticipant(participantId);
                if (part != null) {
                        //get the participant's full name in English.
                        engFullName = part.getELastName() + ", " + part.getEFirstName();
                        //get the participant's full name in Chinese.
                        chiFullName = part.getCLastName()+part.getCFirstName();
                        //************************
                        //get the organization's name and country.
                        //************************
                        OrganizationsInDB orgsInDB = OrganizationsInDB.getInstance();
                        //find the ID of the organization employing this participant.
                        String oid = orgsInDB.getOrganization(participantId);
                        if (oid != null) {
                                Organization org = orgsInDB.locateOrganization(oid);
                                engOrgName = org.getEName();
                                chiOrgName = org.getCName();
                                engOrgCountry = org.getEAddress().getCountry();
                                chiOrgCountry = org.getCAddress().getCountry();
                        }
                }
        }
```

This comment says that the code fragment following it will get the full name of the participant. In order to make the code fragment as clear as this comment, we can extract the code fragment into a method and use this comment to name the method, making this separate comment no longer necessary:

```
void loadInfoFromDB(String participantId) {
        this.participantId = participantId;
        getParticipantFullNames();
        //**********************
        //get the organization's name and country.
        //**********************
        //find the ID of the organization employing this participant.
        OrganizationsInDB orgsInDB = OrganizationsInDB.getInstance();
        String oid = orgsInDB.getOrganization(participantId);
        if (oid != null) {
                Organization org = orgsInDB.locateOrganization(oid);
                engOrgName = org.getEName();
                chiOrgName = org.getCName();
                engOrgCountry = org.getEAddress().getCountry();
                chiOrgCountry = org.getCAddress().getCountry();
        }
}
void getParticipantFullNames() {
        ParticipantsInDB partsInDB = ParticipantsInDB.getInstance();
        Participant part = partsInDB.locateParticipant(participantId);
        if (part != null) {
                //get the participant's full name in English.
                engFullName = part.getELastName() + ", " + part.getEFirstName();
                //get the participant's full name in Chinese.
                chiFullName = part.getCLastName()+part.getCFirstName();
        }
}
```

Likewise, the code fragment to get the information of the organization of the participant can be extracted into a method and be named by the comment, making the comment unnecessary:

```
void loadInfoFromDB(String participantId) {
        this.participantId = participantId;
        getParticipantFullNames();
        getOrgNameAndCountry();
}
void getParticipantFullNames() {
        ParticipantsInDB partsInDB = ParticipantsInDB.getInstance();
        Participant part = partsInDB.locateParticipant(participantId);
        if (part != null) {
                //get the participant's full name in English.
                engFullName = part.getELastName() + ", " + part.getEFirstName();
                //get the participant's full name in Chinese.
                chiFullName = part.getCLastName()+part.getCFirstName();
        }
}
void getOrgNameAndCountry() {
        OrganizationsInDB orgsInDB = OrganizationsInDB.getInstance();
        //find the ID of the organization employing this participant.
        String oid = orgsInDB.getOrganization(participantId);
        if (oid != null) {
```

```
                    Organization org = orgsInDB.locateOrganization(oid);
                    engOrgName = org.getEName();
                    chiOrgName = org.getCName();
                    engOrgCountry = org.getEAddress().getCountry();
                    chiOrgCountry = org.getCAddress().getCountry();
                }
        }
```

An extracted method can be put into another class

Consider the two comments below:

```
public class ParticipantInfoOnBadge {
        ...
        void getParticipantFullNames() {
                ParticipantsInDB partsInDB = ParticipantsInDB.getInstance();
                Participant part = partsInDB.locateParticipant(participantId);
                if (part != null) {
                        //get the participant's full name in English.
                        engFullName = part.getELastName() + ", " + part.getEFirstName();
                        //get the participant's full name in Chinese.
                        chiFullName = part.getCLastName()+part.getCFirstName();
                }
        }
}
```

As the programmer thinks these code fragments not clear enough, then he should extract them and use the comments to name them. But this time the extracted methods should be put into the Participant class instead of the ParticipantInfoOnBadge class:

```
public class ParticipantInfoOnBadge {
        ...
        void getParticipantFullNames() {
                ParticipantsInDB partsInDB = ParticipantsInDB.getInstance();
                Participant part = partsInDB.locateParticipant(participantId);
                if (part != null) {
                        engFullName = part.getEFullName();
                        chiFullName = part.getCFullName();
                }
        }
}
public class Participant {
        String getEFullName() {
                return getELastName() + ", " + getEFirstName();
        }
        String getCFullName() {
                return getCLastName() + getCFirstName();
        }
}
```

Use a comment to name an existing method

Consider the comment below:

```
public class ParticipantInfoOnBadge {
```

```
        ...
        void getOrgNameAndCountry() {
                OrganizationsInDB orgsInDB = OrganizationsInDB.getInstance();
                //find the ID of the organization employing this participant.
                String oid = orgsInDB.getOrganization(participantId);
                if (oid != null) {
                        Organization org = orgsInDB.locateOrganization(oid);
                        engOrgName = org.getEName();
                        chiOrgName = org.getCName();
                        engOrgCountry = org.getEAddress().getCountry();
                        chiOrgCountry = org.getCAddress().getCountry();
                }
        }
}
```

We need the comment of "find the ID of the organization employing..." only because the name "getOrganization" is not clear enough, so, we should directly use the comment as the name:

```
public class ParticipantInfoOnBadge {
        ...
        void getOrgNameAndCountry() {
                OrganizationsInDB orgsInDB = OrganizationsInDB.getInstance();
                String oid = orgsInDB.findOrganizationEmploying(participantId);
                if (oid != null) {
                        Organization org = orgsInDB.locateOrganization(oid);
                        engOrgName = org.getEName();
                        chiOrgName = org.getCName();
                        engOrgCountry = org.getEAddress().getCountry();
                        chiOrgCountry = org.getCAddress().getCountry();
                }
        }
}
public class OrganizationsInDB {
        ...
        void findOrganizationEmploying(String participantId) {
                ...
        }
}
```

The improved code

The improved code is shown below (all the comments have been turned into code and no longer exist separately):

```
public class ParticipantInfoOnBadge {
        String participantId;
        String participantEngFullName;
        String participantChiFullName;
        String engOrgName;
        String chiOrgName;
        String engOrgCountry;
        String chiOrgCountry;

        ParticipantInfoOnBadge(String participantId) {
                loadInfoFromDB(participantId);
        }
        void loadInfoFromDB(String participantId) {
                this.participantId = participantId;
```

```
                getParticipantFullNames();
                getOrgNameAndCountry();
        }
        void getParticipantFullNames() {
                ParticipantsInDB partsInDB = ParticipantsInDB.getInstance();
                Participant part = partsInDB.locateParticipant(participantId);
                if (part != null) {
                        participantEngFullName = part.getEFullName();
                        participantChiFullName = part.getCFullName();
                }
        }
        void getOrgNameAndCountry() {
                OrganizationsInDB orgsInDB = OrganizationsInDB.getInstance();
                String oid = orgsInDB.findOrganizationEmploying(participantId);
                if (oid != null) {
                        Organization org = orgsInDB.locateOrganization(oid);
                        engOrgName = org.getEName();
                        chiOrgName = org.getCName();
                        engOrgCountry = org.getEAddress().getCountry();
                        chiOrgCountry = org.getCAddress().getCountry();
                }
        }
}
```

Why delete the separate comments?

Why delete the separate comments? In fact, comments by themselves are not bad. The problem is that we often do not write clear code (because it is hard), so we take a shortcut (use comments) to hide the problem. The result is, nobody will try to make the code clearer. Later, as the code is updated, commonly nobody updates the comments accordingly. In time, opposed to making the code easier to read, these outdated comments will actually mislead the readers. At the end of the day, what we have is: Some code that is unclear by itself, mixed with some incorrect comments.

Therefore, whenever we see a comment or would like to write one, we should think twice: Can the comment be turned into code, making the code as clear as the comment? You will find that in most of the time the answer is yes. That is, every comment in the code is a good opportunity for us to improve our code. To say in another way, if the code includes a lot of comments, it probably means that the code quality is not that high (however, including a few or no comments does not necessarily mean the code quality is high).

Method name is too long

Consider the example below:

```
class StockItemsInDB {
        //find all the stock items from overseas whose inventory is smaller than 10.
        StockItem[] findStockItems() {
                ...
        }
}
```

In order to turn this comment into code, in principle we should change the code like this:

```
class StockItemsInDB {
        StockItem[] findStockItemsFromOverseasWithInventoryLessThan10() {
                ...
        }
}
```

However, this method name is too long, warning us that the code has problems. What should we do? We should determine: Is the customer of this system really only interested in those stock items from overseas and whose inventory are less than 10? Would he be interested in those stock items from overseas and whose inventory are less than 20? Would he be interested in those stock items from local and whose inventory are greater than 25? If yes, we can turn the comment into parameters:

```
class StockItemsInDB {
        StockItem[] findStockItemsWithFeatures(
                boolean isFromOverseas,
                InventoryRange inventoryRange) {
                ...
        }
}
class InventoryRange {
        int minimumInventory;
        int maximumInventory;
}
```

If the customer is really only interested in those stock items from overseas and whose inventory are less than 10, he must have some particular reason (why only those but not the others?). After further conversation he may say it is because he needs to replenish the stock, because the shipping from overseas takes longer.

Therefore, we find out that what he is really interested is the stock items that need replenishing, instead of those from overseas and whose inventory are less than 10. Therefore, we can turn the real purpose into the method name and turn the comment into the method body:

```
class StockItemsInDB {
        StockItem[] findStockItemsToReplenish() {
                StockItem stockItems[];
                stockItems = findStockItemsFromOverseas();
                stockItems = findStockItemsInventoryLessThan10(stockItems);
                return stockItems;
        }
}
```

References

- http://c2.com/cgi/wiki?TreatCommentsWithSuspicion.

Kent Tong can be contacted via the Editor.

adverstisment

Subscribe for email delivery of

ObjectiveView

(in PDF format)

Email: objective.view@ratio.co.uk with subject: subscribe

Do check out back issues at:

www.objectiveviewmagazine.com

Scott Ambler gives ObjectiveView his latest update on agile modeling…

Abstract

Agile Modeling (AM) defines a collection of values, principles, and practices which describe how to streamline your modeling and documentation efforts. These practices can be used to extend agile processes such as Extreme Programming (XP) and Scrum to make modeling and documentation explicit activities and Rational Unified Process (RUP) and Enterprise Unified Process (EUP) to make modeling and documentation less dysfunctional. The Agile Model Driven Development (AMDD) lifecycle describes an approach for applying AM in conjunction with agile implementation techniques such as Test Driven Development (TDD), code refactoring, and database refactoring.

Modeling is an important part of all software development projects because it enables you to think through complex issues before you attempt to address them via code. Unfortunately many modeling efforts prove to be dysfunctional. At one end of the spectrum are projects where no modeling is performed, either because the developers haven't any modeling skills or because they have abandoned modeling as a useless endeavor.

At the other end of the spectrum are projects which sink in a morass of documentation and overly detailed models, either because the project team suffers from "analysis paralysis" and finds itself unable to move forward or because the team has burdened itself with too many modeling specialists who don't have the skills to move forward even if they wanted to.

Somewhere in the middle are project teams that invest in modeling and documentation efforts only to discover that the programmers ignore the models anyway, often because the models are unrealistic or simply because the programmers think they know better than the modelers (and often they do).

We need to find a way to avoid these problems, to gain the benefits of modeling and documentation without suffering the drawbacks. This is what Agile Model Driven Development (AMDD) is all about.

Agile Models

A *model* is an abstraction that describes one or more aspects of a problem or a potential solution addressing a problem. Traditionally, models are thought of as zero or more diagrams plus any corresponding documentation. However non-visual artifacts such as use cases, a textual description of one or more business rules, or a collection of class responsibility collaborator (CRC) cards [1] are also models. An *agile model* [2] is a model that is just barely good enough.

Agile models are just barely good enough when they exhibit the following traits:

- Agile models fulfill their purpose.
- Agile models are understandable.
- Agile models are sufficiently accurate.
- Agile models are sufficiently consistent.
- Agile models are sufficiently detailed.
- Agile models provide positive value.
- Agile models are as simple as possible.

Figures 1 and 2 both depict agile models. Figure 1 shows a hand-drawn screen design sketch which was drawn in collaboration with users in order to identify what they felt a potential screen should look like.

Figure 2 depicts a physical data model (PDM) using the Unified Modeling Language (UML) notation [3] – it is possible to data model effectively using the UML. Both models are agile even though they're very different from each other:

- The data model is very likely a keeper whereas the screen sketch would be discarded once it's served it has purpose.
- The data model was created using a sophisticated modeling tool whereas the screen sketch was created using very simple tool.
- The data model was created using a complex notation, yet the screen sketch is clearly free-form.
- The data model depicts technical, detailed design whereas the screen sketch is more of an analysis-level diagram.

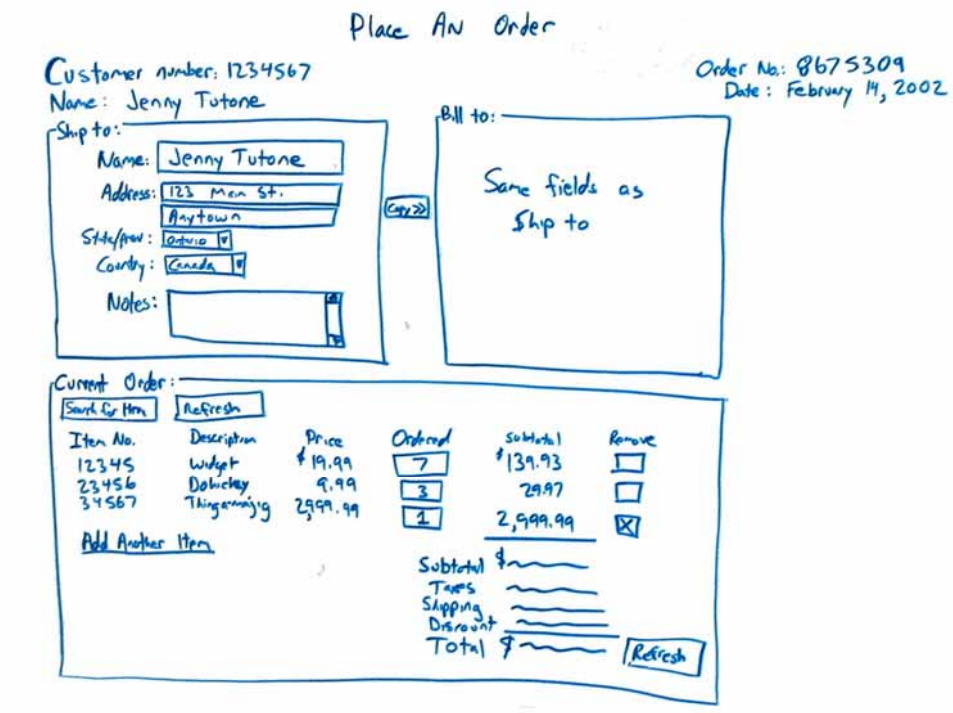


Figure 1: A hand-drawn screen sketch.

It is important to distinguish between the orthogonal concepts of models and documents: some models become documents, or parts of documents, although many models are discarded after they have been used. I suspect that 90% or more of all models are discarded – how many whiteboard sketches have you erased throughout your career? For the sake of definition a document is a permanent record of information, and an *agile document* [2] is a document that is just barely good enough. The principles and practices of Agile Modeling, described in the next section, are applicable to both modeling and documentation.

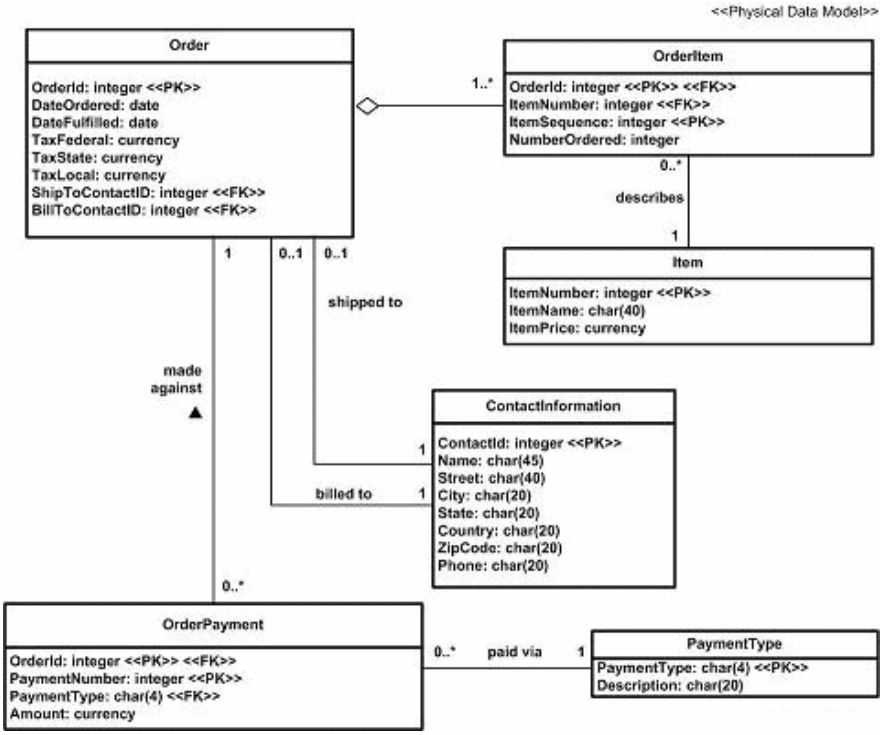


Figure 2: A physical data model (PDM).

Agile Modeling (AM)

The Agile Modeling (AM) process [2] is a chaordic collection of practices – guided by principles and values – that should be applied by software professionals on a day-to-day basis. The focus of AM is to make your modeling and documentation efforts lean and effective; AM does not address the complete system lifecycle and thus should be characterized as a partial process/process. The advantage of this approach is that organizations may benefit from the focused guidance of a partial process. The disadvantages are that organizations need the requisite knowledge and skills to know which processes exist and how to combine them effectively. The concept of partial processes seems strange at first, but when you reflect a bit you quickly realize that partial processes are the norm – development processes, such as Extreme Programming (XP) [4] and the Rational Unified Process (RUP) [6], address the system development lifecycle but do not address the full IT lifecycle. The Enterprise Unified Process (EUP) [7] – an extension to the RUP which addresses the production and retirement phases of a system, operations and support of a system, and cross-system issues such as enterprise architecture and strategic reuse – and ISO/IEC 12207-compliant processes [18] represent full IT lifecycles.

AM is practices-based, it is not prescriptive. In other words it does not define detailed procedures for how to create a given type of model, instead it provides advice for how to be effective as a modeler. The advantage of describing a process as a collection of practices is that it is easy for experienced professionals to learn and reflects (hopefully) what they actually do, the disadvantage is that it does not provide the detailed guidance for novices. Prescriptive processes, on the other hand, often provide the detailed guidance required by novices but are ignored by experienced professionals. Prescriptive processes are well suited as training material for new hires and perhaps as input into process audits to fulfill the requirement that you have a well documented process.

Think of AM as more of an art than a science. It is defined as a collection of values (www.agilemodeling.com/values.htm), principles (www.agilemodeling.com/principles.htm), and practices (www.agilemodeling.com/practices.htm). The values of AM include those of XP v1 – *communication*, *simplicity*, *feedback*, and *courage* – and extend it with *humility* (XP v2 adds the fifth value of *respect*, which I argue comes from *humility*). The principles of AM, many of which are adopted or modified from XP, provide guidance to agile developers who wish to be effective at modeling and documentation. They provide a philosophical foundation from which AM's practices are derived. The practices of AM are what people actually do. There is not a specific ordering to the practices, nor are there detailed steps to complete each one – you simply do the right thing at the right time.

Because every project team is different, and every environment is different, you should tailor your process to reflect your situation. AM reflects this philosophy – to claim that you are "doing AM" you merely need to adopt its values, its core principles and practices (see Table 1). The remaining principles and practices are optional, although they are very good ideas and should be adopted whenever possible. All of the values, principles, and practices are presented in detail at www.agilemodeling.com. This approach enables you to tailor AM to meet your exact needs. Table 2 lists the supplementary principles and practices although for brevity does not describe them in detail.

Why would you want to adopt AM? AM defines and shows how to take a light-weight approach to modeling and documentation. What makes AM a catalyst for improvement is not the modeling techniques themselves – such as use case models, class models, data models, or user interface models – but how to apply them productively. As depicted in Figure 3, AM can be tailored into other agile software development processes, such as XP or Feature Driven Development (FDD) [5], to enhance their modeling and documentation efforts. AM can also be tailored into "near-agile" processes, such as the RUP or EUP. Although you must be following an agile software process to truly be agile modeling, but you may still adopt and benefit from many of AM's practices on non-agile projects.

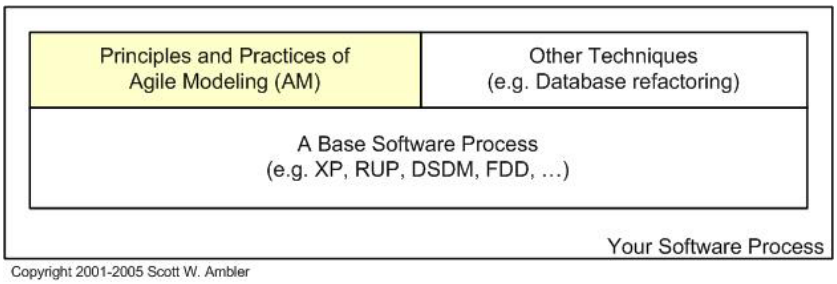


Figure 3: Tailoring AM into your software process.

Core Principles	Core Practices
• Assume Simplicity • Embrace Change • Enabling the Next Effort is Your Secondary Goal • Incremental Change • Maximize Stakeholder Investment • Model With a Purpose • Multiple Models • Quality Work • Rapid Feedback • Software is Your Primary Goal • Travel Light	• Active Stakeholder Participation • Apply the Right Artifact(s) • Collective Ownership • Create Several Models in Parallel • Create Simple Content • Depict Models Simply • Display Models Publicly • Iterate To Another Artifact • Model in Small Increments • Model With Others • Prove it With Code • Single Source Information • Use the Simplest Tools

Table 1. The core principles and practices of AM.

Supplementary Principles	Supplementary Practices
• Content is more important than representation • Open and honest communication • Work with people's instincts	• Apply modeling standards • Apply patterns gently • Discard temporary models • Formalize contract models • Update only when it hurts

Table 2. Supplementary principles and practices.

Agile Model Driven Development (AMDD)

As the name implies, AMDD is the agile version of Model Driven Development (MDD). MDD is an approach to software development where extensive models are created before source code is written. A primary driver of MDD is the Object Management Group (OMG)'s Model Driven Architecture (MDA) standard [11]. With MDD the goal is typically to create comprehensive models, and then ideally generate software from those models. This often requires complex computer aided system engineering (CASE) tools, so it is not surprising to discover that CASE tool vendors are often rabid proponents of MDD and MDA. I'm not convinced that the MDA is going to get much traction within the IT industry [15], if only for the simple reason that few IT professionals have the sophisticated modeling skills which MDA requires. AMDD applies the AM values, principles, and practices to an MDD-based approach.

AMDD takes a much more realistic approach: its goal is to describe how developers and stakeholders can work together cooperatively to create models which are just barely good enough. It assumes that each individual has some modeling skills, or at least some domain knowledge, that they will apply together in a team in order to get the job done.

It is reasonable to assume that developers will understand a handful of the modeling techniques indicated in Figure 4 but not all of them. It is also reasonable to assume that people are willing to learn new techniques over time, often by working with someone else that already has those skills. AMDD does not require everyone to be a modeling expert, it just requires them to be willing to try.

AMDD also allows people to use the most appropriate modeling tool for the job, often very simple tools such as whiteboards or paper, because you want to find ways to communicate effectively, not document comprehensively. There is nothing wrong with sophisticated CASE tools in the hands of people who know how to use them, but AMDD does not depend on such tools.

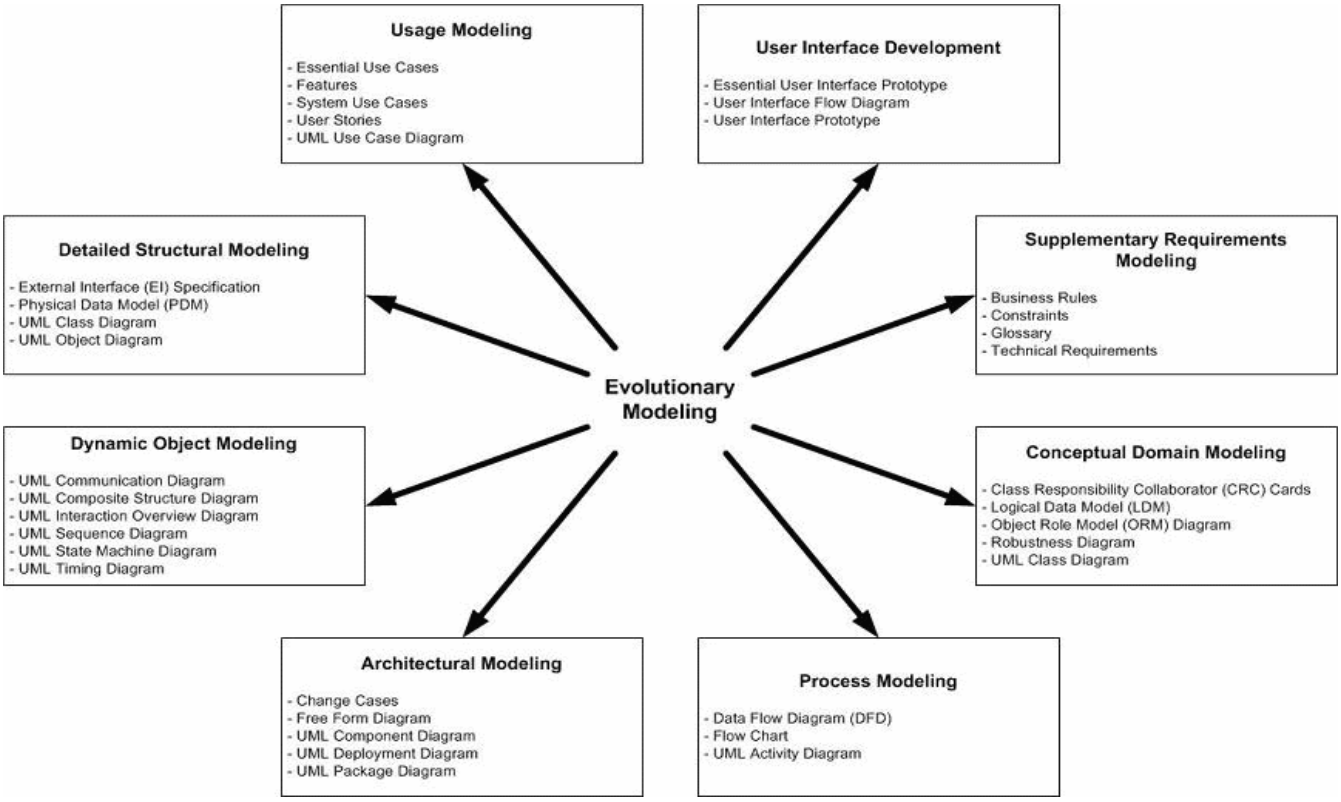


Figure 4: Categories of modeling and some suggested techniques.

Figure 5 depicts a high-level lifecycle for AMDD for the release of a system [9]. Each box represents a development activity. The initial up front modeling activity occurs during cycle/iteration 0 and includes two main sub-activities, initial requirements modeling and initial architecture modeling. The other activities – model storming, reviews, and implementation – potentially occur during any cycle, including cycle 0. The time indicated in each box represents the length of an average session: perhaps you will model for a few minutes then code for several hours.

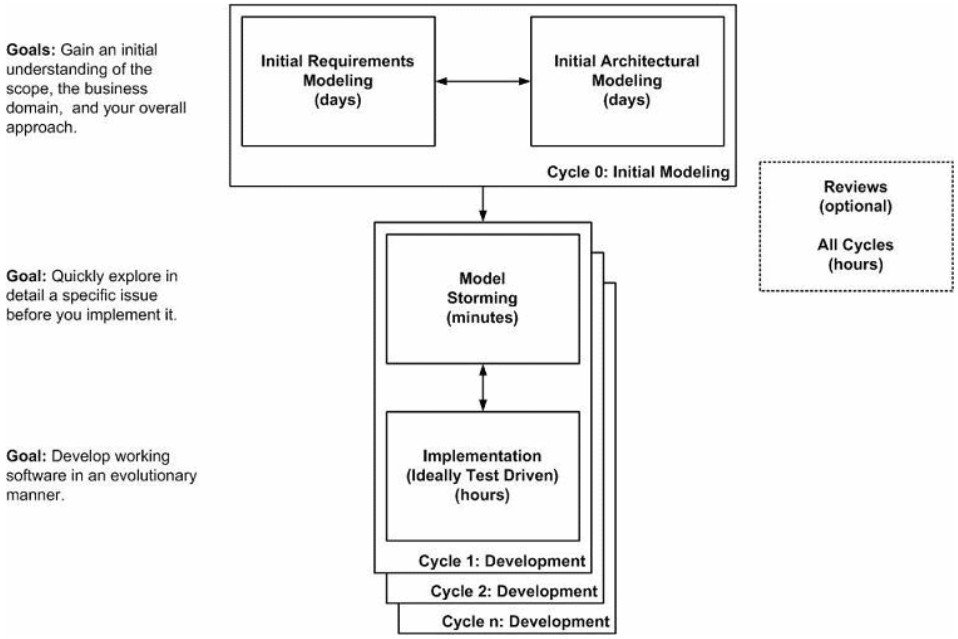


Figure 5. Taking an AMDD approach to development.

Initial Modeling

The initial modeling effort is typically performed during the first week of a long-term project. For short projects (perhaps several weeks in length) you may do this work in the first few hours and for longer projects (perhaps on the order of twelve or more months) you may decide to invest up to two weeks in this effort. You should not invest any more time than this as you run the danger of over modeling and of modeling something that contains too many problems (two weeks without the concrete feedback that implementation provides is a long time to go at risk).

Initial modeling occurs during cycle 0, the only time that an agile modeler will spend more than an hour or two at once modeling because they follow the practice *Model in Small Increments*. During cycle 0 you are likely to identify high-level usage requirements models such as a collection of use cases or user stories; identify high-priority technical requirements and constraints; create a high-level (sparse) domain model; and draw sketches representing critical architectural aspects of your system. In later cycles both your initial requirements and your initial architect models will need to evolve as you learn more, but for now the goal is to get something that is just barely good enough so that your team can get coding. In subsequent releases you may decide to shorten cycle 0 to several days, several hours, or even remove it completely as your situation dictates.

Model storming

During development cycles you explore the requirements or design in greater detail, and your "model storming" sessions are often on the order of minutes. Model storming is a just-in-time (JIT) approach to modeling with a twist – you model just in time and just enough to address the issue at hand. Perhaps you will get together with a stakeholder to analyze the requirement you're currently working on, create a sketch together at a whiteboard for a few minutes, and then go back to coding. Or perhaps you and several other developers will sketch out an approach to implement a requirement, once again spending several minutes doing so. Or perhaps you and your programming pair will use a modeling tool to model in detail and then generate the code for that requirement. Model storming sessions shouldn't take more than 15 or 20 minutes, otherwise you're likely not following the AM practice *Iterate to Another Artifact* properly, and often take a few minutes at most.

It's important to understand that your initial requirements and architecture models will evolve through your detailed modeling and implementation efforts. That's perfectly natural. Depending on how light you're travelling, you may not even update the models if you kept them at all.

You may optionally choose to hold model reviews and even code inspections, but these quality assurance (QA) techniques really do seem to be obsolete with agile software development. Although many traditionalists consider model reviews to be best practices they're really "compensatory practices" that compensate for common process-oriented mistakes such as:

- Distributing your team across several locations, thereby putting you at risk that the teams are not aware of what the others are doing.
- For allowing one person or a subset of people (often specialists) to "own" the model, thereby putting you at risk that the model is of poor quality or does not reflect the work of the others on the team.
- For long feedback loops, such as a (near) serial approach to development when it can be months or even years between modeling and coding activities.

When you follow AM's practices of Active Stakeholder Participation, *Collective Ownership*, *Model With Others*, and *Prove it With Code* you typically avoid these problems. The high-communication and open environment enjoyed by agile modelers ensures that many people, if not everyone on the team, works with all artifacts. This ensures that many "sets of eyes" see any given model, thereby increasing the chance that mistakes are found early. The focus on producing working software ensures that the ideas captured in models are quickly put to the test – very often something will be modeled and then implemented the very same day. In these environments the value of reviews quickly disappears.

Implementation

Implementation is where your team will spend the majority of its time. During development it is quite common to model storm for several minutes and then code, following common agile implementation practices for several hours or even days. These implementation practices are:

- **Code refactoring**. Refactoring [12] is a disciplined way to restructure code to improve its design. A code refactoring is a simple change to your code that improves its design but does not change its behavioral semantics. In other words a code refactoring does not add new functionality. Common code refactorings include Rename Process, Remove Control Flag, Change Value to Reference, and Move Process.

- **Database refactoring**. A database refactoring [10] is a simple change to a database schema that improves its design while retaining both its behavioral and informational semantics. There are

different types of database refactorings. Some focus on data quality (such applying a consistent format to the values stored in a column), some focus on structural changes (such as renaming or splitting a column), whereas others focus on performance enhancements (such as introducing an index). Structural database refactorings are the most challenging because a change to the structure of your database could cause your application (or others) to crash.

- **Test-Driven Design (TDD)**. Test-driven development (TDD) [13, 14], also known as test-first programming or test-first development, is an approach where you identify and write your tests before your write your code. There are four basic steps to TDD. First, you quickly add a test (just enough code to fail), the idea being that you should refuse to write new code unless there is a test that fails without it. The second step is to run your tests, either all or a portion of them, to see the new test fail. Third, you make a little change to your code, just barely enough to make your code pass the tests. Next you run the tests and hopefully see them all succeed – if not you need to repeat step 3. There are several advantages of TDD. First, it ensures that you always have a 100% unit regression test suite in place, showing that your software actually works. Second, TDD enables you to refactor your code safely because you know you can find anything that you "break" via a refactoring. Third, TDD provides a way to think through detailed design issues, reducing your need for detailed modeling.

These three techniques are effectively enablers of AMDD. Refactoring helps you to maintain a quality design within your object schema over time and supports detailed changes to your design that aren't captured within your design models. Similarly database refactoring helps you to maintain a quality design within your data schema, in many ways it could be thought of as normalization after the fact. Both techniques push evolutionary design decisions into the hands of the people most qualified to make them – the people actually building the system. AMDD and TDD go hand-in-hand because they are both "think before you code" techniques. AMMD provides a way to think through big issues whereas TDD provides a way to think through detailed issues.

Conclusion

Modeling is a skill that all developers must gain to be effective. Agile Modeling (AM) defines a collection of values, principles, and practices which describe how to streamline your modeling and documentation efforts. Modeling can easily become an effective and high-value activity if you choose to make it so; unfortunately many organizations choose to make it a bureaucratic and documentation-centric activity which most developers find intolerable.

The Agile Model Driven Development (AMDD) process describes a approach for applying AM in conjunction with agile implementation techniques such as Test Driven Development (TDD), code refactoring, and database refactoring. AMDD enables agile developers to think through larger issues before they dive down into the implementation details. AMDD is a valuable technique to have in your intellectual toolbox.

Resources

1. Beck, K., and Cunningham, W. (1989). A laboratory for teaching object-oriented thinking. In *Proceedings of OOPSLA'89*, pp. 1–6.
2. Ambler, S.W. *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*. New York: John Wiley & Sons Publishing. 2002
3. Ambler, S.W. *An Unofficial Profile for Data Modeling Using the UML*. www.agiledata.org/essays/umlDataModelingProfile.html
4. Beck, K. *Extreme Programming Explained – Embrace Change*. Reading, MA: Addison Wesley Longman, Inc. 2000
5. Palmer, S. R. & Felsing, J. M. *A Practical Guide to Feature-Driven Development*. Upper Saddle River, NJ: Prentice Hall PTR. 2002.
6. Kruchten, P. *The Rational Unified Process 2nd Edition: An Introduction*. Reading, MA: Addison Wesley Longman, Inc. 2000
7. Ambler, S.W., Nalbone, J, and Vizdos, M.J. *The Enterprise Unified Process: Extending the Rational Unified Process*. Upper Saddle River, NJ: Prentice Hall PTR. 2005.
8. Cockburn, A. *Agile Software Development.* Reading, MA: Addison Wesley Longman, Inc. 2002
9. Ambler, S.W. *The Object Primer 3rd Edition: Agile Model Driven Development with UML 2*. New York: Cambridge University Press, 2004.
10. Ambler, S. W. *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. New York: Wiley. 2003.
11. *Model Driven Architecture (MDA) Home Page*. www.omg.org/mda/
12. Fowler, M. *Refactoring: Improving the Design of Existing Code*. Menlo Park, CA: Addison Wesley Longman. 1999.
13. Astels, D.. *Test Driven Development: A Practical Guide*. Upper Saddle River, NJ: Prentice Hall. 2003.
14. Beck, K.. *Test Driven Development: By Example*. Boston, MA: Addison Wesley. 2003.
15. Ambler, S.W. *Agile Model Driven Development is Good Enough*. IEEE Software, September/October 2003, 20(5), pp. 70-73.

16. Ambler, S.W.. *Inclusive Modeling*. www.agilemodeling.com/essays/inclusiveModeling.htm. 2004.
17. Breen, P. *Software Craftsmanship*. Boston, MA: Addison Wesley. 2002.
18. *Guide for ISO/IEC 12207 (Software Life Cycle Processes)*. International Standards Organization (ISO), www.iso.org. 1998.
19. Ambler, S.W. *The Elements of UML 2.0 Style*. New York: Cambridge University Press. 2005.

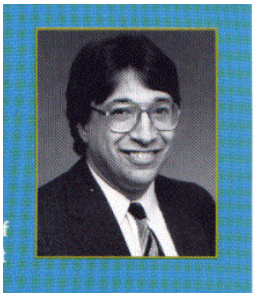
About the Author

Scott Ambler (www.ambysoft.com/scottAmbler.html): is a Senior Consultant with Canada-based Ambysoft Inc. a software services consulting firm that specializes in software process mentoring and improvement. He is founder and thought leader of the Agile Modeling (AM) (www.agilemodeling.com), Agile Data (AD) (www.agiledata.org), and Enterprise Unified Process (EUP) (www.enterpriseunifiedprocess.com) methodologies. He helps organizations adopt and tailor these processes to meet their exact needs as well as provides training and mentoring in these techniques.

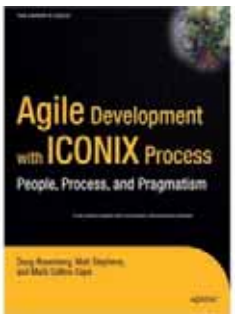
Advertisment

http://www.objectiveviewmagazine.com/

Combining Design Driven Testing with Test Driven Design



Doug Rosenberg and Matt Stephens discuss how to combine model driven design with test driven design…



ICONIX Process is a minimalist, use-case driven object modeling process that is well suited to agile Java development. It uses a core subset of UML diagrams, and provides a reliable method of getting from use cases to source code in as few steps as possible. It's described in the book *Agile Development with ICONIX Process* (more information can be found about the book here: www.softwarereality.com/AgileDevelopment.jsp).

Because the process uses a minimal set of steps, it's also well suited to agile development, and can be used in tandem with test-driven development (TDD) to help "plug the gaps" in the requirements.

The book describes the use case driven analysis and design process in detail, with lots of examples using UML, C# and Java. However, for this book excerpt, we focus on how to combine unit testing with up-front UML modeling, to produce a really rigorous software design. The process begins with the use cases and UML diagrams, then moves into Java source code via JUnit…

Test-Driven Development with ICONIX Process

In *Agile Development with ICONIX Process*, we put together an example system using "vanilla" test-driven development (TDD). We then repeat the example using a mixture of TDD and ICONIX modeling. In the excerpt below, we show this aspect of agile ICONIX development.

The premise behind TDD is that you write the unit tests first, then write the code to make the tests pass. The process of doing this in theory lets you design the code as you write it. However, we prefer a more rigorous, "higher-level" design approach, which we describe here.

How Agile ICONIX Modeling and TDD Fit Together

There's a prevailing opinion in the agile world that "formal" up-front design modeling and TDD are mutually exclusive. However, we're going to demonstrate that TDD can in fact be particularly effective with an up-front design method like ICONIX Process.

ICONIX Process takes the design to a low level of detail via sequence diagrams — one sequence diagram for each use case. These diagrams are used to allocate behaviors to the class diagrams. The code can then be written quickly without much need for refactoring. However, the coding stage is still not exactly a brainless activity. The programmer (who, incidentally, should also be actively involved in the design modeling stage) still needs to give careful thought to the low-level design of the code. This is an area to which TDD is perfectly suited.

The "Vanilla" Example Repeated Using ICONIX Modeling and TDD

The following serves as our list of requirements for this initial release:

• Create a new customer.

• Create a hotel booking for a customer.

• Retrieve a customer (so that we can place the booking).

• Place the booking.

As luck would have it, we can derive exactly one use case from each of these requirements (making a total of four use cases). For this example, we'll focus on the first use case, "Create a New Customer".

Let's start by creating a domain model that contains the various elements we need to work with, as shown in Figure 1. As you can see, it's pretty minimal at this stage. As we go through analysis, we discover new objects to add to the domain model, and we possibly also refine the objects currently there. Then, as the design process kicks in, the domain model swiftly evolves into one or more detailed class diagrams.

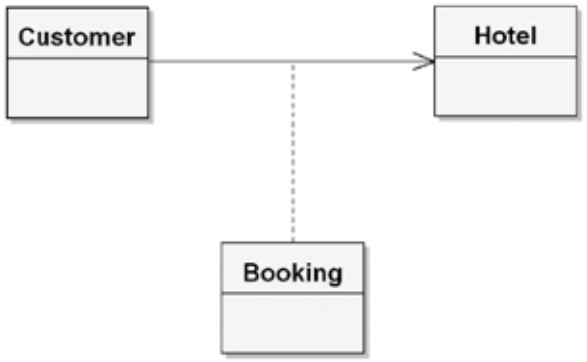


Figure 1. Domain model for the hotel booking example

The objects shown in Figure 12-1 are derived simply by reading through our four requirements and extracting all the nouns. The relationships are similarly derived from the requirements. “Create a hotel booking for a customer,” for example, strongly suggests that there needs to be a Customer object that contains Booking objects. (In a real project, it might not be that simple—defining the domain model can be a highly iterative process involving discovery of objects through various means, including in-depth conversations with the customer, users, and other domain experts. Defining and refining the domain model is also a continuous process throughout the project’s life cycle.)

If some aspect of the domain model turns out to be wrong, we change it as soon as we find out, but for now, it gives us a solid enough foundation upon which to write our use cases.

Here’s the use case for “Create a New Customer”:

- **Basic Course:** The system shows the Customer Details page, with a few default parameters filled in. The user enters the details and clicks the Create button; the system validates that all the required fields have been filled in; and the system validates that the customer name is unique and then adds the new Customer to the database. The system then returns the user to the Customer List page.
- **Alternative Course:** Not all the required fields were filled in. The system informs the user of this and redisplays the Customer Details form with the missing fields highlighted in red, so that the user can fill them in.
- **Alternative Course:** A customer with the same name already exists. The system informs the user and gives them the option to edit their customer details or cancel.

This use case probably has more user interface details than you’re used to seeing in a use case. This is a characteristic of “ICONIX-style” use cases: they’re quite terse, but they are very closely tied to the domain model, and to the classes that you’ll be designing from.

Next, we draw a robustness diagram – i.e. a picture version of the use case (see Figure 2).

A robustness diagram shows conceptual relationships between objects. Because it’s an “object drawing” of the use case text, it occupies a curious space halfway between analysis and design. Nevertheless, mastering **robustness analysis** is the key to creating rigorous designs from clear, unambiguous use cases.

The robustness diagram shows three types of object:

- Boundary objects (a circle with a vertical line at the left) – these represent screens, JSP pages and so forth
- Entities (a circle with a horizontal line at the bottom) – these are the data objects (e.g. Customer, Hotel Booking)
- Controllers (a circle with an arrow-head at the top) – these represent actions that take place between other objects (i.e. Controllers are the verbs)

(Note that in the book, we take the “Create a New Customer” use case and robustness diagram through several iterations, using the robustness diagram to polish up and “disambiguate” the use case text. For brevity we just show the finished version here).

Sequence Diagram for “Create a New Customer”

Now that we’ve disambiguated our robustness diagram (and therefore also our use case text), let’s move on to the sequence diagram (see Figure 3).

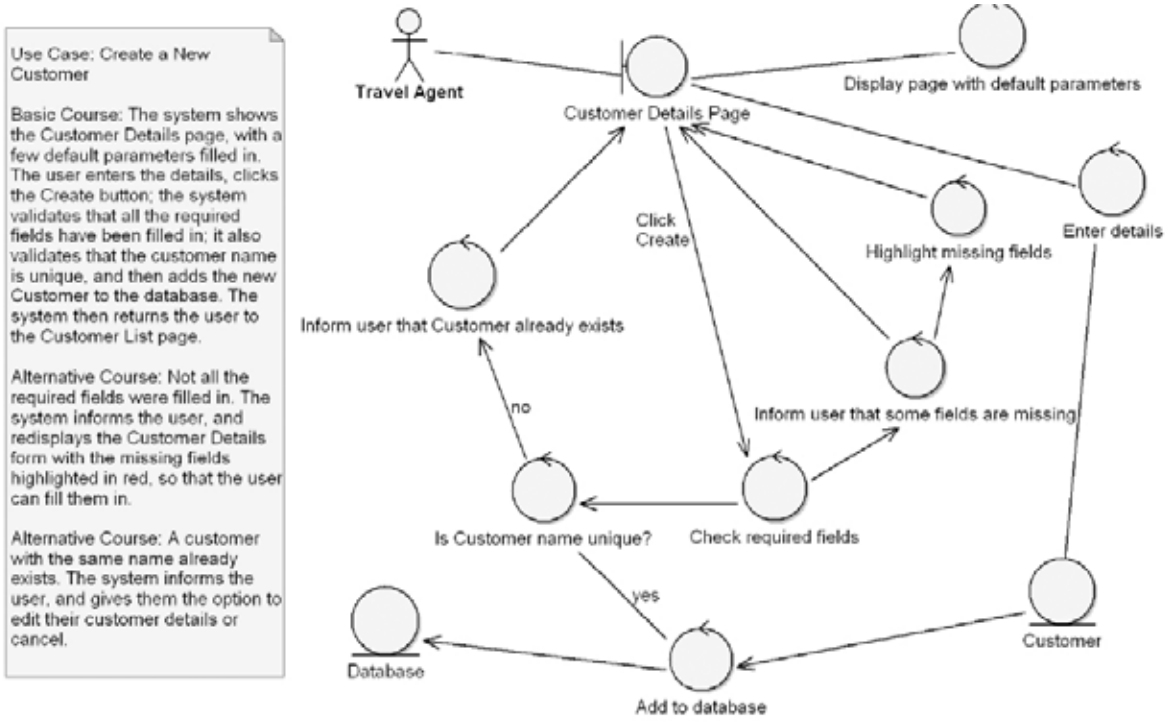


Figure 2. Robustness diagram for the Create a New Customer use case

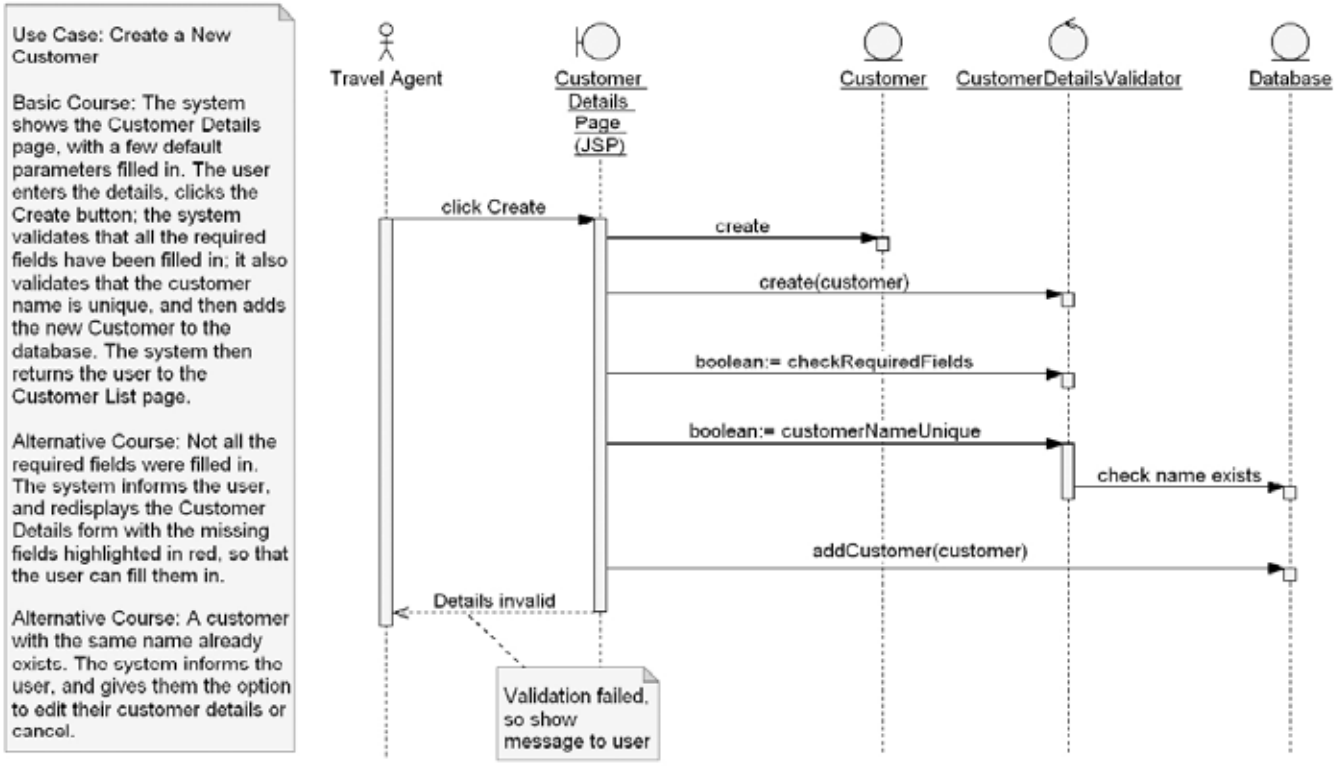


Figure 3. Sequence diagram for the Create a New Customer use case

More Design Feedback: Mixing It with TDD

The next stage is where the ICONIX+TDD process differs slightly from vanilla ICONIX Process. Normally, we would now move on to the class diagram, and add in the newly discovered classes and operations. We could probably get a tool to do this part for us, but sometimes the act of manually drawing the class diagram from the sequence diagrams helps to identify further design errors or ways to improve the design; it's implicitly yet another form of review.

We don't want to lose the benefits of this part of the process, so to incorporate TDD into the mix, we'll write the test skeletons as we're drawing the class diagram. In effect, TDD becomes another design review stage, validating the design that we've modeled so far. We can think of it as the last checkpoint before writing the code (with the added benefit that we end up with an automated test suite).

So, if you're using a CASE tool, start by creating a new class diagram (by far the best way to do this is to copy the existing domain model into a new diagram). Then, as you flesh out the diagram with attributes and operations, simultaneously write test skeletons for the same operations.

Here's the important part: the tests are driven by the controllers and written from the perspective of the Boundary objects.

If there's one thing that you should walk away from this article with, then that's definitely it! The controllers are doing the processing — the grunt work — so they're the parts that most need to be tested (i.e., validated that they are processing correctly). Restated: the controllers represent the software behavior that takes place within the use case, so they need to be tested. However, the unit tests we're writing are black-box tests (aka closed-box tests)—that is, each test passes an input into a controller and asserts that the output from the controller is what was expected. We also want to be able to keep a lid on the number of tests that get written; there's little point in writing hundreds of undirected, aimless tests, hoping that we're covering all of the failure modes that the software will enter when it goes live. The Boundary objects give a very good indication of the various states that the software will enter, because the controllers are only ever accessed by the Boundary objects. Therefore, writing tests from the perspective of the Boundary objects is a very good way of testing for all reasonable permutations that the software may enter (including all the alternative courses). Additionally, a good source of individual test cases is the alternative courses in the use cases. (In fact, we regard testing the alternative courses as an essential way of making sure all the "rainy-day" code is implemented.)

Okay, with that out of the way, let's write a unit test. To drive the tests from the Control objects and write them from the perspective of the Boundary objects, simply walk through each sequence diagram step by step, and systematically write a test for each controller. Create a test class for each controller and one or more test methods for each operation being passed into the controller from the Boundary object.

Looking at the sequence diagram in Figure 3, we should start by creating a test class called CustomerDetailsValidatorTest, with two test methods, testCheckRequiredFields() and testCustomerNameUnique():

```
package iconix;
import junit.framework.*;

public class CustomerDetailsValidatorTest extends
TestCase {

   public CustomerDetailsValidatorTest(String
      testName) {
      super(testName);
   }

   public static Test suite() {
      TestSuite suite = new  TestSuite
        (CustomerDetailsValidatorTest.class);
      return suite;
   }

   public void testCheckRequiredFields() throws
   Exception {
   }

   public void testCustomerNameUnique() throws
   Exception {
   }
}
```

At this stage, we can also draw our new class diagram (starting with the domain model as a base) and begin to add in the details from the sequence diagram/unit test (see Figure 4).

As you can see in Figure 4, we've filled in only the details that we've identified so far using the diagrams and unit tests. We'll add more details as we identify them, but we need to make sure that we don't guess at any details or make intuitive leaps and add details just because it seems like a good idea to do so at the time.

TIP: Be ruthlessly systematic about the details you add (and don't add) to the design.

In the class diagram in Figure 4, we've indicated that CustomerDetailsValidator is a <<control>> stereotype. This isn't essential for a class diagram, but it does help to tag the control classes so that we can tell at a glance which ones have (or require) unit tests.

Next, we want to write the actual test methods. Remember, these are being driven by the controllers, but they are written from the perspective of the Boundary objects and in a sense are directly validating the design we've created using the sequence diagram, before we get to the "real" coding stage. In the course of writing the test methods, we may identify further operations that might have been missed during sequence diagramming.

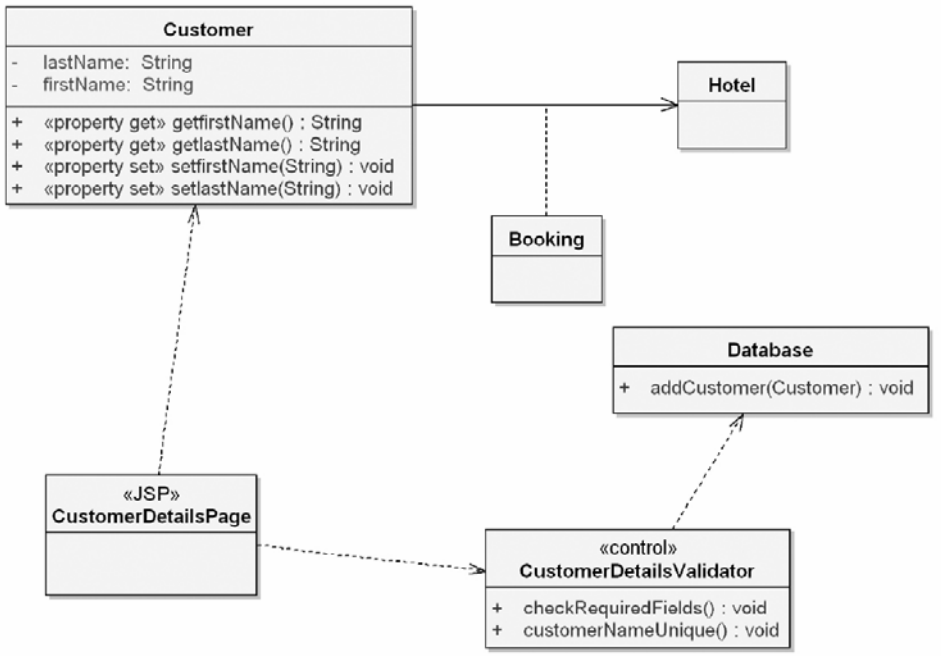


Figure 4. Beginnings of the detailed class diagram

Our first stab at the testCheckRequiredFields() method looks like this:

```
public void testCheckRequiredFields() throws
Exception {
   List fields = new ArrayList();
   Customer customer = new Customer (fields);
   boolean allFieldsPresent =
      customer.checkRequiredFields();
   assertTrue("All required fields should be present",
            allFieldsPresent);
}
```

Naturally enough, trying to compile this initially fails, because we don't yet have a CustomerDetailsValidator class (let alone a checkRequiredFields() method). These are easy enough to add, though:

```
public class CustomerDetailsValidator {
   public CustomerDetailsValidator (List fields) {
   }
   public boolean checkRequiredFields() {
      return false; / / make the test fail initially.
   }
}
```

Let's now compile and run the test. Understandably, we get a failure, because checkRequiredFields() is returning false (indicating that the fields didn't contain all the required fields):

```
CustomerDetailsValidatorTest
.F.
Time: 0.016
There was 1 failure:
1)
testCheckRequiredFields(CustomerDetailsValidator
Test)
junit.framework.AssertionFailedError:
All required fields should be present at
CustomerDetailsValidatorTest.testCheckRequiredFi
elds(
CustomerDetailsValidatorTest.java:21)
FAILURES!!!
Tests run: 2, Failures: 1, Errors: 0
```

However, where did this ArrayList of fields come from, and what should it contain? In the testCheckRequiredFields() method, we've created it as a blank ArrayList, but it has spontaneously sprung into existence—an instant warning sign that we must have

skipped a design step. Checking back, this happened because we didn't properly address the question of what the Customer fields are (and how they're created) in the sequence diagram (see Figure 3). Let's hit the brakes and sort that out right now (see Figure 5).

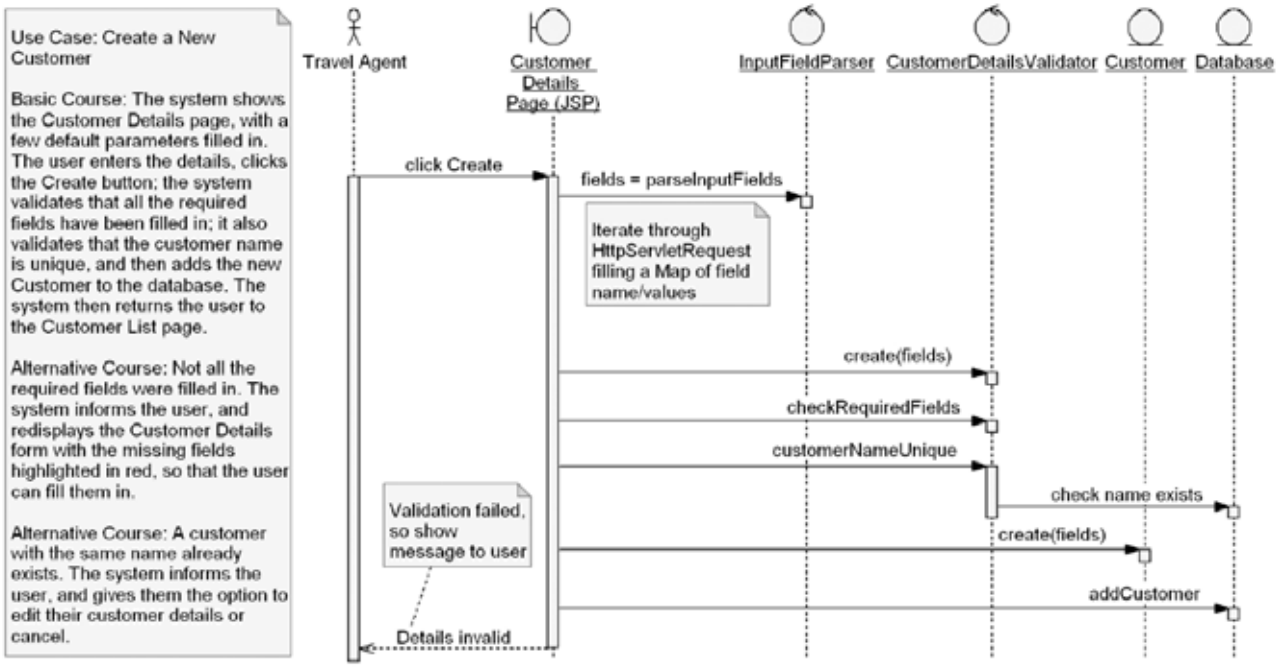


Figure 5. Revisiting the sequence diagram to add more detail

Revisiting the sequence diagram identified that we really need a Map (a list of name/value pairs that can be looked up individually by name) and not a sequential List.

Now that we've averted that potential design mishap, let's get back to the CustomerDetailsValidator test. As you may recall, the test was failing, so let's add some code to test for our required fields:

```
public void testCheckRequiredFields() throws
Exception {
   Map fields = new HashMap();
   fields.put("userName", "bob");
   fields.put("firstName", "Robert");
   fields.put("lastName", "Smith");
   Customer customer = new Customer(fields);
   boolean allFieldsPresent =
customer.checkRequiredFields();
   assertTrue("All required fields should be present",
         allFieldsPresent);
}
```

A quick run-through of this test shows that it's still failing (as we'd expect). So now let's add something to CustomerDetailsValidator to make the test pass:

```
public class CustomerDetailsValidator {
   private Map fields;

   public CustomerDetailsValidator (Map fields) {
      this.fields = fields;
   }

   public boolean checkRequiredFields() {
      return fields.containsKey("userName") &&
         fields.containsKey("firstName") &&
         fields.containsKey("lastName");
   }
}
```

Let's now feed this through our voracious unit tester:

```
CustomerDetailsValidatorTest
..
Time: 0.016
OK (2 tests)
```

The tests passed!

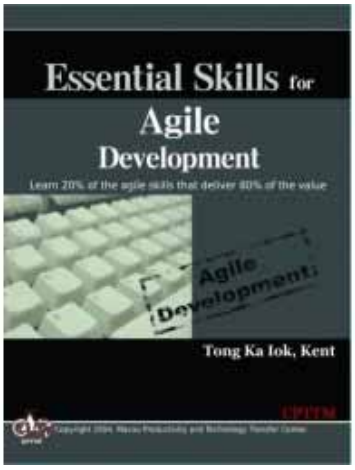
Summing Up

Hopefully this article gave you a taster of what's involved in combining a code-centric, unit test-driven design methodology (TDD) with a UML-based, use case-driven methodology (ICONIX Process). In *Agile Development with ICONIX Process*, we take this example further, showing how to strengthen the tests and the use cases by adding controllers for form validation, and by writing unit tests for each of the alternative courses ("rainy day scenarios") in the use cases.

Links:

Agile ICONIX Process:
http://www.softwarereality.com/AgileDevelopment.jsp
ICONIX Software Engineering (training and consulting):
http://www.iconixsw.com
Test Driven Development: **http://ww.testdriven.com**

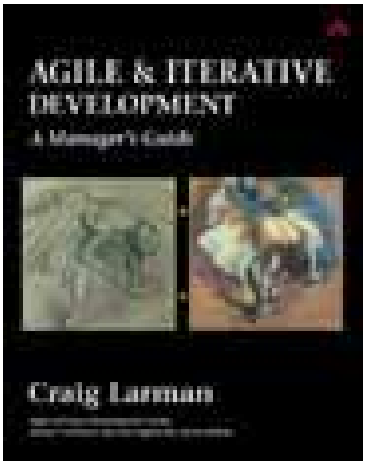
Books to look out for …



Essential Skills for Agile Development

This book has an elegant yet highly effective minimalist style. Rather than long theoretical discussion the book does what it does by example - and there's plenty of example code given. See the article earlier in this magazine. Overall the book covers many topics and issues related to agile software development, including: keeping code fit; handling inappropriate references; seperating database, UI and domain logic; unit testing and acceptance testing amongst others.

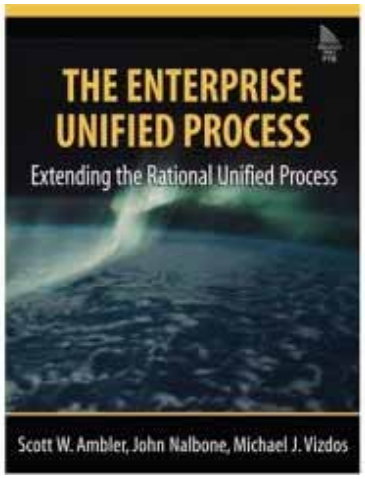
The reason this book is to be recommended to developers, is that even if you're not doing full on "agile" development, there's still plenty of useful material in it. The lack of hype is also refreshing - the book focuses on examples and shows good solutions. You should get it!



Agile and Iterative Development, A Manager's Guide

Using statistically significant research and large-scale case studies, noted methods expert Craig Larman presents the most convincing case ever made for iterative development. Larman offers a concise, information-packed summary of the key ideas that drive all agile and iterative processes, with the details of four noteworthy iterative methods: Scrum, XP, RUP, and Evo.

This book is a must if you need to get a grip on the spectrum of agile development techniques out there.



The Enterprise Unified Process

The Rational Unified Process is a powerful tool for improving software development -- but it doesn't go nearly far enough. Today's development organizations need to extend RUP to cover the entire IT lifecycle, including the cross-project and enterprise issues it largely ignores.

The Enterprise Unified Process systematically identifies the business and technical problems that RUP fails to address, and shows how EUP fills those gaps. Using actual examples and case studies, the authors introduce processes and disciplines for producing new software, implementing strategic reuse, "sunsetting" obsolete code and systems, managing software portfolios, and much more.

Books included in this section are selected on merit by the editor.

advertisment

An Architectural Reference Model for Large Scale Applications

a one day workshop

Introduction

This one day workshop explores an architectural reference model (ARM) applicable for large scale object-oriented applications. As object-oriented application become larger, with ever more classes and interfaces, the complexity of inter-class/interface dependencies increases – potentially exponentially. This typically manifests itself in applications becoming increasingly brittle, making change difficult and quality uncertain.

The ARM assists in managing complexity through the time honoured principle of of divide and conquer. By using the ARM and its associated - and fairly easy to apply, set of rules – your application will have a far greater coherency of structure, leading to:

- improved flexibility and increased ability to respond to customer requested change,
- greater clarity of responsibility – as to which code does what,
- improved code factoring – reducing duplication within the code,
- increased consistency in packaging rules,
- more manageable and well understood dependencies between packages,
- improved test coverage,
- increased likelyhood of achieving re-use,
- and, in general, greater overall application stability and quality

Contents

For further information email: info@ratio.co.uk

An online article based on the course material can be found at:
http://www.ftponline.com/ea/magazine/summer2005/features/mcollinscope