METHODS & TOOLS

Global knowledge source for software development professionals **ISSN 1661-402X**

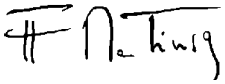
Summer 2005 (Volume 13 - number 2)

Is Collaboration the Synonym of Success?

In this issue, you will find an article about "Using Customer Tests to Drive Development". Lisa Crispin has produced interesting material, sharing her experience from the trenches. I saw yesterday a proverb that could summarise what I felt after reading this article: "Those who succeed find solutions, those who fail find excuses". There is no silver bullet described in this article. It is just the story of people trying to work together to improve the results of their efforts. And I suppose that Lisa Crispin is not stranger to this situation, even if most of the time she will use the "we" to describe what happens.

In the past century, I was evaluating quality of software development processes using the first questionnaire developed by the Software Engineering Institute. I had a customer whose project management activities were not formalised and this was a serious weakness to achieve CMM level 2. When the CIO asked me what to do, I replied that this could not necessarily be a problem, if the users and the developers were cooperating. Today this company has a very formalised software projects controlling (the term "management" seems to me inappropriate...) system where there is user-side project managers, development-side project managers and a lot of various tracking systems and reports. But this has not improved the rate of success of its projects.

Part of what Lisa Crispin shows in her article is that you can achieve better results when all project participants cooperate. Users, developers and testers are too often placed in situations when they try only to achieve their personal objectives, instead of working together to get the best results for the project. Formalisation of relationships often leads to confrontations. Users think that the developers do not want to accept their improved requirements. Developers think that users wants more than originally planned, often when development had already promised more than it could deliver. In real life projects, it is not common to find situations where both parties accept change... and its consequences. Beyond agility, for me the main lesson from Lisa Crispin's article is that a better project is a project where everybody takes in consideration the needs of all participants, or as Barry Boehm once stated it, looks for win-win situations.



Inside

Agile Development with ICONIX Process

Doug Rosenberg, Matt Stephens and Mark Collins-Cope
http://www.softwarereality.com/AgileDevelopment.jsp

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 on http://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 this book, 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

Let's pause and rewind, then, back to the start of the TDD example that we covered in the previous chapter. To match the example, we're going to need a system for travel agent operatives to place hotel bookings on behalf of customers.

To recap, 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.

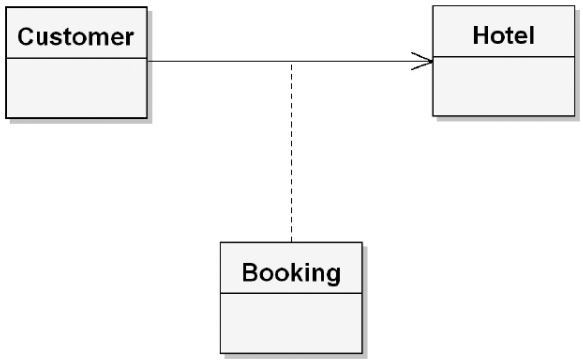


Figure 1. Domain model for the hotel booking example

The objects shown in Figure 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).

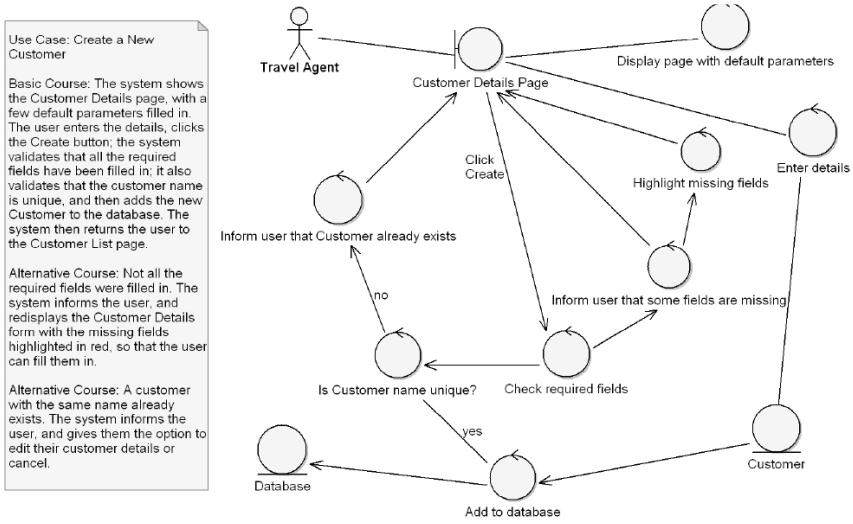


Figure 2. Robustness diagram for the Create a New Customer use case

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).

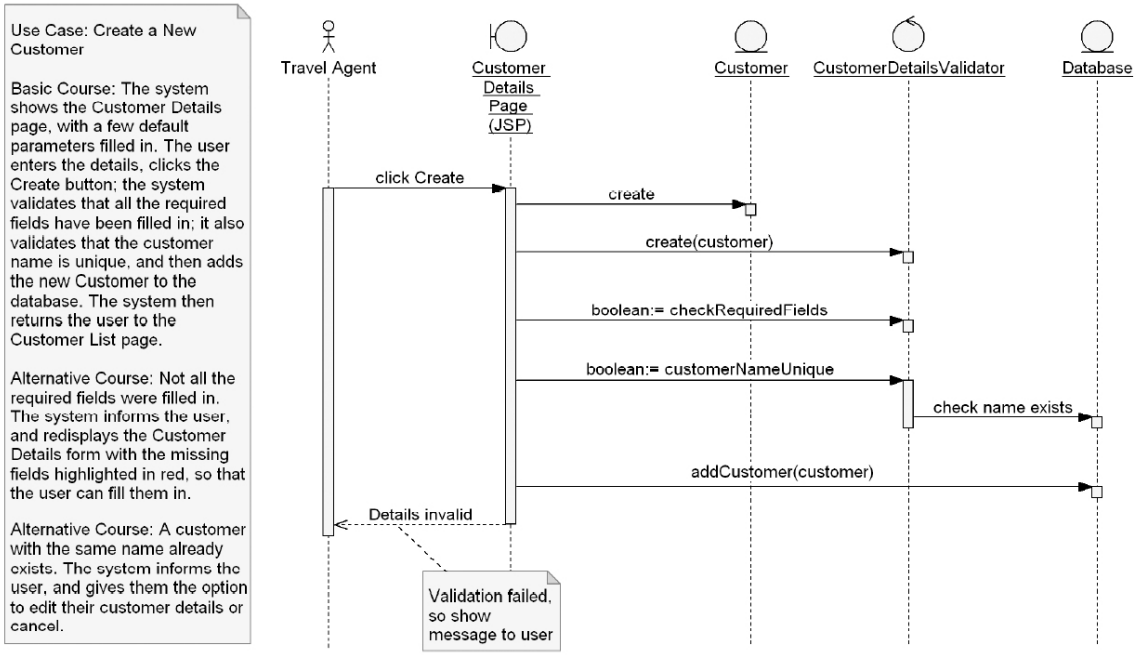


Figure 3. Sequence diagram for the Create a New Customer use case

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).

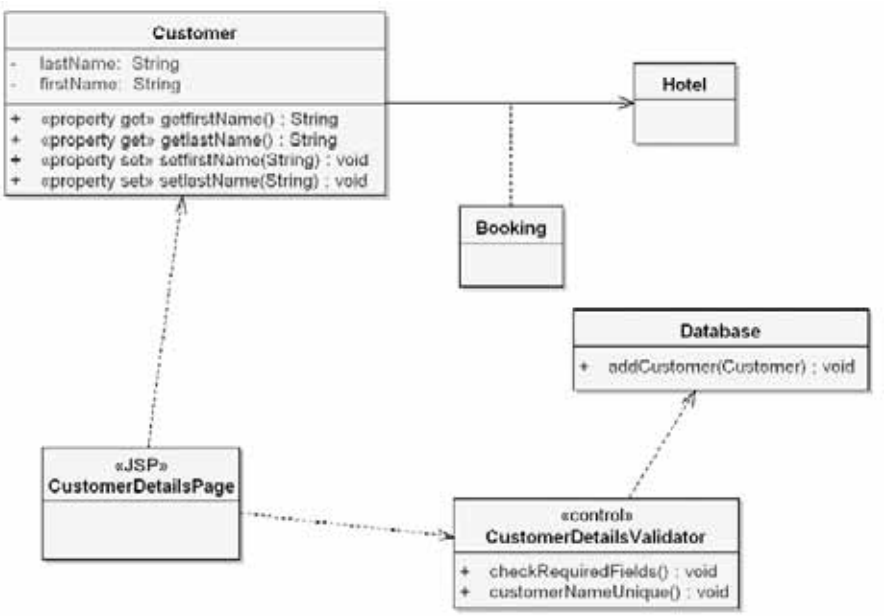


Figure 4. Beginnings of the detailed class diagram

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.

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(CustomerDetailsValidatorTest)
    junit.framework.AssertionFailedError:
        All required fields should be present
        at CustomerDetailsValidatorTest.testCheckRequiredFields(
        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).

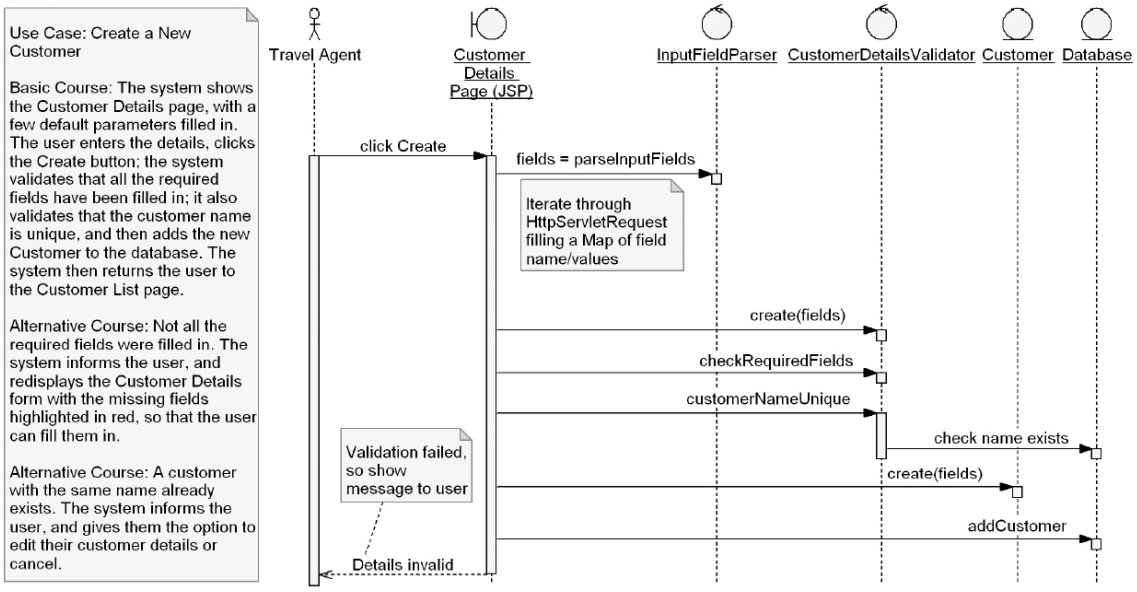


Figure 5. Revisiting the sequence diagram to add more detail

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 an 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.

References

Agile Development with ICONIX Process: People, Process, and Pragmatism
by Doug Rosenberg, Mark Collins-Cope, Matt Stephens
Publisher: Apress, ISBN: 1590594649

Agile ICONIX Process: http://www.softwarereality.com/AgileDevelopment.jsp

Test Driven Development: http://www.testdriven.com

Using Customer Tests to Drive Development

Lisa Crispin, lisa.crispin@gmail.com
http://lisa.crispin.home.att.net/

Like many agile software development teams, our team writes tests for each feature before the feature is actually developed. We've found many advantages to using tests to drive development, not only at the unit test level but at the functional, system and acceptance test levels. Not only do we have tests which show whether we've delivered the correct functionality, but we benefit from increased communication and collaboration, increasing the chances that we will deliver exactly what our customers want. Writing just the right amount of tests and level of detail has proved difficult at times, as has the automation and timing of the automation effort. The effort to overcome those problems has paid off and led us to devote even more resources to driving development with customer tests.

Test-driven development or TDD is a widely accepted practice used by agile software development teams of many flavors – not only Extreme Programming teams. For each small bit of functionality they code, programmers first write unit tests, then they write the code that makes those unit tests pass. TDD is seen as a design tool, since it forces the programmer to think about many aspects of each feature before coding. It results in a 'safety net' of tests that can be run with each build, ensuring that new code doesn't 'break' any existing code, or that refactored code maintains its functionality.

Why Customer Test-Driven Development?

I've been fortunate to work as a tester on three development teams practicing TDD as they produce J2EE-based web applications. Code produced via TDD far outshines code produced in a 'traditional' manner in quality and robustness. Testing code developed in this manner, I don't find the bugs I was used to finding in 'typical' projects, such as bugs produced by boundary conditions or 'invalid' input. Don't get me wrong, as a tester, this makes me happy. So what more do I want?

What I still saw, even with TDD, are misunderstandings between the project's customers (also known as business experts or product owners) and the software developers. Even if the deployed code was almost bug-free, it didn't do everything the customer had expected. We can enlarge on the concept of TDD and reduce the risk of delivering the 'wrong' code with what I call customer test-driven development.

Customer test-driven development (CTDD), also known as story test-driven development (SDD), consists of driving projects with tests and examples that illustrate the requirements and business rules. What do I mean when I say 'customer tests'? Basically, anything beyond unit and integration (or contract) testing, which are done by and for the programmers, testing small units of code and their interaction. I use the term 'customer' in the XP sense, meaning product owners and people on the business side who specify features to be delivered. Customer tests may include functional, system, end-to-end, performance, load, stress, security, and usability testing, among others. These tests show the customers whether the delivered code meets their expectations.

How CTDD Works

How do we know everything that the customer expects ahead of time? On agile projects, we make this easier by splitting features into small, manageable chunks, known as 'stories'. Often written on a small index card, stories take a form such as this:

> *As a retail website customer, I would like to be able to delete items out of my shopping cart so that I can check out with only the items I want.*

For each story, we ask the customer to tell us how she will know when this story is complete, in the form of tests and examples. Just as with TDD, before writing any code, we write tests that, when passing, prove the code meets the minimum requirements. These tests are ideally in a form that can be used with an automated tool, but they may also be higher-level tests, or guidelines for later exploratory testing.

Who are these cooperative customers? They're the people with the domain expertise, who understand business priorities. They may be in sales or marketing, they may be business managers or analysts, they may be end users, they may even be a development manager or customer support. They will most likely need help in writing effective tests that the programmers can use.

Testers provide this help, combining their understanding of the technical requirements of the system with the big picture of what the business needs. Testers ask questions to help elicit requirements and flush out hidden assumptions.

For example, with the story above to delete items out of the shopping cart, testers may ask:

Should there be a dialog for the user to confirm the delete?

Is there a need to save the deleted items somewhere for later retrieval?

Can you draw a picture of how the delete interface should look?

What should happen if all items in the cart are deleted?

What if the user has two browser sessions open on the same cart, and deletes an item in one of the sessions?

By writing customer tests ahead of coding the features, we can bring hidden assumptions to the surface. Frequent conversations with our customers, going over examples, enable our product owners to get the best possible results.

Teams doing TDD, especially those trying it for the first time, may tend to do only the more obvious, "happy path" tests. Misunderstood requirements and hard-to-find defects may go undetected. Writing customer tests first provides navigation for our project 'road trip'. The tests help the team identify milestones and landmarks to know if they're on track. When the tests all pass, we know we've reached our destination.

CTDD in Real Life

The basic process of CTDD sounds simple. Once stories are identified for an iteration, we can specify customer tests, write the fixtures which automate the tests, then write the code that makes the tests pass.

First, you need some way to specify the tests and then automate them. Our team uses a tool called FitNesse (http://www.fitnesse.org). FitNesse is an open source software development collaboration tool which enables customers, testers and programmers to specify test cases as simple tables of inputs and expected outputs. It is a wiki, which means it is easy to create and edit pages in a web browser without having to know HTML. Programmers find it easy to write test fixtures which operate the code and return the results, using the same language as the application under test (for example, Java).

Even with an appropriate lightweight tool such as FitNesse, specifying and then automating tests can be a huge challenge, as my team found when we decided to try CTDD. First of all, no matter how easy it is to specify tests in the tool, it's hard to make time to write tests in advance of, or even at the very beginning of an iteration. We were already struggling with TDD, and felt short of resources. When we faced a big upcoming theme to develop, our customer and I made writing tests in advance a priority. The customer wrote detailed examples in spreadsheet form, and I turned these into FitNesse test tables. Not surprisingly, we made some big mistakes.

When the programmers started writing the automated fixtures for the first story, they found the large number of highly detailed level of the tests overwhelming. It was hard for them to get a big picture of what how the highly complex business rules should work. Also, once they started writing the fixtures to automate the tests, the programmers needed a major redesign to the test cases. Since I had written so many tests in advance, it was a big and tedious job to go back and refactor all the tests. We also had disagreement in where business logic should be located. The programmers originally wanted to put much of the logic into the SQL that retrieved records for processing, but this made test automation more difficult. This turned into a bit of a crisis, and we wondered if CTDD was worth the investment. In spite of this, the FitNesse tests which finally emerged proved to be valuable.

In hindsight, even this small crisis and ensuing discussion proved a valuable learning experience. We finally agreed that putting the complex business logic into the Java code was the safest course of action. It could be tested more easily and thoroughly. We also agreed that, at least for our project, writing detailed tests well in advance of coding didn't serve the purpose of helping the programmers know what to code. It also led to wasted time refactoring tests later.

We decided to write only high level tests in advance of each development iteration, or during the first couple days of each iteration (we use two week iterations). These test cases, written by the business experts and/or testers, usually consist of bullet points, narrative and/or examples on a wiki page. Often, our business experts will explain the story to the team and write examples on a whiteboard. Customer tests don't replace direct communication between programmers and customers, they enhance it and document the story requirements. The programmers use these high-level tests to understand each story and help with the design. The high-level test wiki page might look something like this:

Story: As a retail website customer, I would like to be able to delete items out of my shopping cart so that I can check out with only the items I want.

Notes:
A delete box is displayed next to each item in the shopcart. An 'update cart' button is at the bottom of the list. If the user marks the delete box for one or more items and clicks the 'update cart' button, a window pops up giving the user the option to delete or move the item(s) to his wishlist. This popup window also has a cancel button.

Test cases:

- *Click the update cart button without checking any items. Should refresh the page.*
- *Delete one item. Verify it is gone from the cart and not in the wishlist.*
- *Move one item to the wishlist. Verify it is there and no longer displayed in the cart.*
- *Delete all the items. The "keep shopping" button should appear.*
- *(and so on – you get the idea).*

As the iteration gets underway, we write FitNesse tests for each story. The other tester on the team and I do this, collaborating closely with our business experts, who often prepare examples and test cases in spreadsheets. We use the build-operate-check pattern for our tests. Each test page starts off with tables that build inputs to the story. These inputs may be global parameters, data stored in memory to be used by the test, or if necessary, actual data in the database. Then we invoke a method (not written yet) which will operate on the inputs with the actual code. Last, the test contains tables which verify the results, again either reading data from memory or from the database itself. Each test also has a setup and teardown method.

For the sample story above, about deleting items from a shopcart, the test might be organized this way:

1. Build a shopcart into memory, adding all the fields for the item such as item number, description, price, quantity.
2. Operate on the shopcart, specifying one item to delete.
3. Check to see that the shopcart contains the correct remaining items.

Unless we're fairly certain of the test design, we only write one or two test cases, so that we don't lose a lot of work if we need a big refactoring later. We show the test to a programmer and discuss whether it's a good approach, making changes if needed. In true CTDD, the programmers would write the test fixture to automate these tests before writing the production code, the same as they do with unit tests. In our project, the programmers usually look over the high-level test cases and write their first draft of the code. Then they take on the task to automate the FitNesse test. Often, the test case shows a requirement that they neglected or misunderstood, and they go back and change the code accordingly. Once the methods for the FitNesse tests are working, we can go back and add test cases, often uncovering more design flaws or just plain bugs.

Once our FitNesse tests are passing, they become part of our daily regression test suite, catching any flaws introduced later. But their most important function has already been served: Writing the tests has forced communication between customers and testers, customers and programmers, testers and programmers. The team had a good understanding of the story's requirements before starting to write any code, and the resulting code has a good chance of meeting all the customer's expectations.

Building on CTDD

Despite the success whenever we write customer tests first, we can become complacent or get in too much of a rush and neglect this practice. Recently we had a fairly simple story (we thought) that only a couple of internal customers would use. We did write test cases in advance of coding, but as the internal customers were always busy and we thought we understood it well, we neglected to go over the test cases with the customers. After deploying to production, one of the customers tried to use the new functionality, and found it was not at all what he wanted. Now we have an unhappy customer, who has to write new stories to get what he wanted in the first place.

When our business was at its peak time of year and we were all busy, sometimes the tasks to automate tests got pushed towards the end of the sprint, making a time crunch for testing. We made 'writing FitNesse fixtures early' a team goal for a few iterations, until this became a habit. We use a task board to monitor progress for each story, and if the 'write FitNesse fixtures' task doesn't move into the Work In Progress column by a certain point, someone will ask about it during the standup meeting.

Our company has experienced the benefits of CTDD, and has made its use a company-wide goal. Although our company is quite small (only around 20 employees), we have a product owner (who is also our ScrumMaster) who works with our team full time. In addition, a senior vice-president who is the most knowledgeable about the domain has been given the time and directive to work closely with our engineering team. He writes test cases for at least one story in advance of each iteration. We have a new goal to finish high-level test cases for all stories of an iteration by the fourth day of the two-week iteration. This new commitment to CTDD, along with our commitment to early test automation, makes us talk to each other when we need to – before and during coding.

Other Success Stories

Our team has drawn inspiration from other teams' use of CTDD. Richard Watt and David Leigh-Fellows presented their techniques in "Acceptance Test Driven Planning" at XP/Agile Universe 2004. They use a practice they call "Getting our stories straight". QA works with customers at end of each iteration to write tests for next. They use these tests as a planning tool to guide estimation and task breakdown. It's not Big Up-Front Design (BUFD), but a "just right" amount of process.

Another source of ideas was an article in *Better Software Magazine* (July/August 2004) by Tracy Reppert called "Don't Just Break Software, Make Software". She details the story test-driven development used by Nielsen Media Research. Joshua Kerievsky, a pioneer of STDD, says it "helps teams obtain the necessary amount of story details before writing code." He also notes, "Following story test driven development clearly enabled me to produce designs I simply would not have anticipated."

What if I'm not On an Agile Team?

"It all sounds great", you say, "but I'm stuck in a traditional waterfall process. How can CTDD benefit my project?" After being on XP teams for a couple years, I had to go back to the "dark side" and work on a less agile team for a time. They wanted to implement agile development, but couldn't quite make the commitment. I asked the managers where they felt the most pain on projects. They immediately responded that requirements were their biggest problem. On some projects, too much time was spent gathering requirements, which were changed and out of date right away and thus useless. On the rest, the programmers were forced to start coding with no requirements at all.

"What if we write customer tests ahead of development?" I proposed. They agreed to try it, and our results were good. We were able to get people on the business side working more closely with us, and the programmers appreciated having some direction before they started writing code. Even though the programmers didn't assist directly with test automation (all automation was done by my test team), writing customer tests first did help guide development. The resulting software was closer to the customer's requirements. This success led to trying more agile practices on ensuing projects, again with good results. Writing tests ahead of development is something a test team can implement without a major effort, it just requires a bit of cooperation from the business side. It can change a team's culture just a bit, so that it may be open to more practices that will improve development, and improve life for the developers!

The Enterprise Implementation Framework (EIF): Beyond the IDEAL Model

By Sinan Si Alhir, salhir@comcast.net
http://home.comcast.net/~salhir

Introduction

Reality involves two dimensions, time and space, and two natural forces, change and complexity. As organizations progress toward their goals and objectives, organizational improvement and change is a necessity that involves the evolution of people, practices (process), and infrastructure (automation).

This paper focuses on introducing the IDEAL model and the Enterprise Implementation Framework (EIF). The IDEAL model is the Software Engineering Institute's (SEI) [1] "organizational improvement model that serves as a roadmap for initiating, planning, and implementing improvement actions." [2] The EIF is a holistic approach for achieving organizational improvement and change at the enterprise, community, and individual levels. I derived the EIF as an extension of the IDEAL model; it has proven most valuable in practice. While both the IDEAL model and the EIF may be more generally applied to organizational improvement and change as well as more specifically applied to software process improvement, the emphasis in this article is on software process improvement.

The IDEAL model offers a very pragmatic approach for software process improvement; however, it emphasizes the management perceptive and activities more than the human perspective, which focuses on the community and individual people. The EIF offers a broader framework for software process improvement wherein IDEAL model activities are balanced with the human perspective.

The IDEAL Model

The IDEAL model is a continuous improvement lifecycle model. It provides a roadmap for executing improvement programs using a lifecycle composed of sequential phases containing activities. It is named for its five phases. It originally focused on software process improvement; however as the SEI recognized that the model could be applied beyond software process improvement, it revised the model (designated as version 1.1) so that it may be more broadly applied. Again, while the IDEAL model may be broadly applied to organizational improvement, the emphasis herein is on software process improvement. Figure 1 shows the IDEAL model's phases and activities.

The ***Initiating phase*** focuses on initiating the software process improvement effort. The Set Context activity focuses on establishing the rationale (stimulus for change) for the software process improvement effort and the business goals and objectives that will be supported by the changes. The Build Sponsorship activity focuses on establishing management support for the software process improvement effort. The Charter Infrastructure activity focuses on establishing the infrastructure for managing the software process improvement effort.

For example, consider an organization whose customers are complaining that product features are not meeting their needs. Perhaps the product is a Global Positioning System (GPS) with both software and hardware components, a Business Process Management (BPM) system with software components only, an Accounting system, an Operating System, or other type of software (and possibly hardware) system.

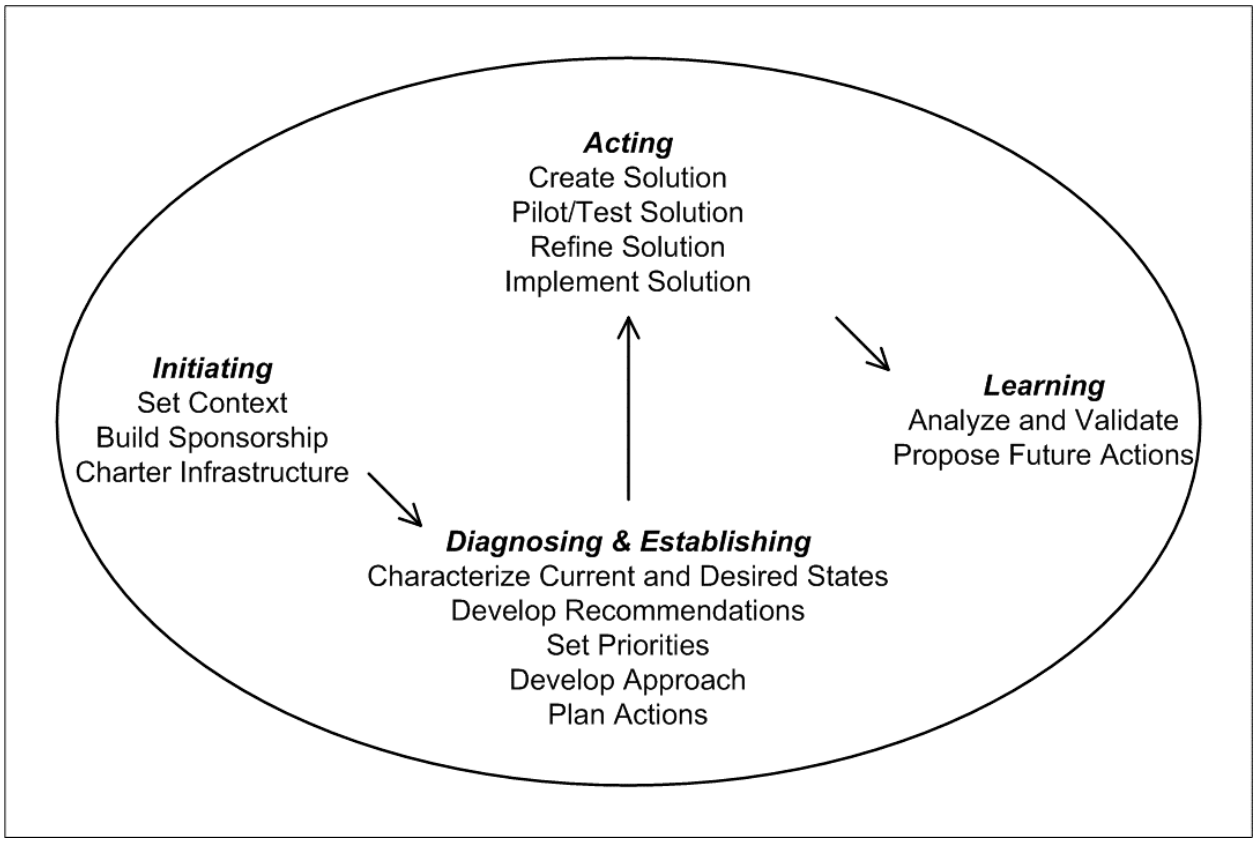


Figure 1: The IDEAL Model.

This situation establishes the rationale for the software process improvement effort with the overall business goal of increasing customer satisfaction by improving the software development process. The various organizational departments, including Marketing, Sales, Fulfillment (including Software Product Development), and Operations (including Support) who are stakeholders to the software development process must sponsor the software process improvement effort and an infrastructure must be established to support the software improvement effort.

The ***Diagnosing phase*** focuses on determining the current as-is and desired to-be state of the organization relative to the software development process. The Characterize Current & Desired States activity focuses on characterizing the current as-is and desired to-be state relative to the rationale established in the Initiating phase (rather than focusing on every aspect of the organization). The Develop Recommendations activity focuses on suggesting recommendations for progressing from the current as-is to the desired to-be state of the software process.

For the organization mentioned above, the current as-is state and the desired to-be state of the software development process relative to customer satisfaction must be explored, and recommendations for improving the software development process must be derived for achieving the overall goal of increasing customer satisfaction. Fundamentally, what is customer satisfaction, how is it measured, what are the measures, what are the measurements, what are the root causes of customer dissatisfaction, and how might customer satisfaction be improved relative to the software development process? Recommendations may include changes in the various organizational departments and how they interact with one another, partners, and customers relative to the software process.

The ***Establishing phase*** focuses on planning the transformation between the current as-is and desired to-be state of the organization relative to the software development process. The Set Priorities activity focuses on establishing the constraints for the software process improvement effort. The Develop Approach activity focuses on developing an overall approach relative to the constraints. The Plan Actions activity focuses on developing a detailed implementation plan relative to the constraints.

For the organization mentioned above, the constraints for the improvement effort must be explored, an overall approach must be derived, and a detailed plan must be derived. Fundamentally, what constraints bind the overall approach and detailed implementation plan, and what approach and plan satisfy those constraints yet allow for the transformation from the current as-is to the desired to-be state of the software development process? There may be specific budgetary constraints for the software process improvement effort or a specific window of time wherein the effort must complete so that the various organizational departments are not negatively impacted and delivery of the product to the customer is not negatively impacted.

The ***Acting phase*** focuses on executing the transformation between the current as-is and desired to-be state of the organization relative to the software development process. The Create Solution activity focuses on creating a multifaceted “best guess” potential software process improvement solution. The Pilot/Test Solution activity focuses on testing (verifying) the software process improvement solution. Verification focuses on “building the solution right”. The Refine Solution activity focuses on refining the software process improvement solution. The Pilot/Test Solution activity and Refine Solution activity are iterated as a test-refine process until the multifaceted “best guess” potential software process improvement solution evolves into a multifaceted “satisfactory” actual software process improvement solution. The Implement Solution activity focuses on implementing the software process improvement solution throughout the organization.

A multifaceted software process improvement solution focuses on people’s knowledge and skills, the organization’s software development processes, and the automation (tool) that enables people to efficiently perform those processes. A “best guess” potential software process improvement solution is usually created by a work group, and a “satisfactory” actual software process improvement solution is reached through iterations of the test-refine process. The work group should work toward an actual software process improvement solution that is “satisfactory” in meeting the goals and objectives of the software process improvement effort rather than simply seeking a “perfect” software process improvement solution.

For the organization mentioned above, perhaps the initial software process improvement solution involves increasing customer follow-up by the Sales department to better understand future product (software and hardware) requirements and increasing the customer support staff in the Operations department to better understand the effectiveness of the software testing and quality activities. However, after piloting this software process improvement solution for a few products, perhaps it is determined that the software process improvement solution is not having a drastic positive impact on customer satisfaction. The software process improvement solution is then refined to include increasing customer interactions in the overall product delivery process performed by the Fulfillment department, including more intense user involvement in requirements gathering and user acceptance testing. After piloting the software process improvement solution and a few more adjustments, it may be determined that the software process improvement solution is satisfactory and may be implemented throughout the organization for various products. Fundamentally, what software process improvement solution works in the context of the specific organization is the satisfactory actual software process improvement solution.

The ***Learning phase*** focuses on learning from the execution of the software process improvement effort. The Analyze and Validate activity focuses on evaluating the execution of the software process improvement effort, including what worked well and not so well, and validation of the software process improvement solution. Validation focuses on "building the right solution". The Propose Future Actions activity focuses on suggesting recommendations for future software process improvement efforts.

For the organization mentioned above, lessons learned are captured and analyzed, and recommendations for future software process improvement efforts are suggested.

The Enterprise Implementation Framework (EIF)

The Enterprise Implementation Framework (EIF) is an organizational transformation (improvement and change) process framework. The EIF provides an infrastructure for executing organizational transformation initiatives using a framework composed of phases wherein iterations contain activities. An organization (or enterprise) resides in an industry and is composed of teams (known as the community) which in turn are composed of team members (individual people). Relative to these three levels (enterprise, community, and individual), the EIF is focused on "value" (return-on-investment), envisioning a to-be, considering the as-is, leveraging a roadmap and realizing improvements and change while establishing an environment (best-practices and automation), fostering a culture (values and practices), and transferring knowledge and skills (self-sufficiency) to the community and individual people. Again, while the EIF may be broadly applied to organizational improvement and change, the emphasis herein is on software process improvement and change.

Figure 2 shows the EIF's phases, an iteration, and activities.

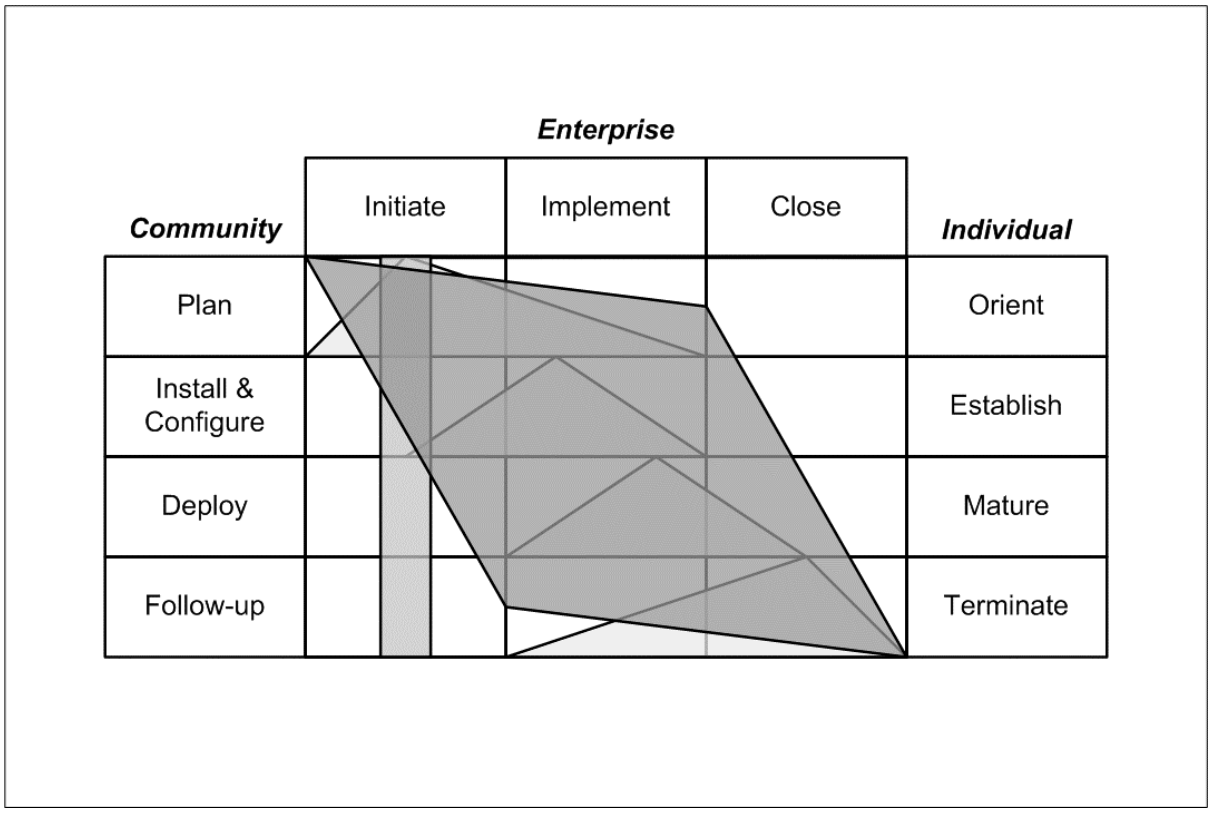


Figure 2: The Enterprise Implementation Framework (EIF).

Cycles and Phases

A cycle is composed of sequential phases. Each cycle results in a software process improvement generation. A phase is a major milestone where stakeholders can determine if the software process improvement initiative should continue or be terminated. Cycles establish an approach for implementing software process improvements within an enterprise. Figure 1 shows the EIF's phases (horizontal).

The ***Initiate phase*** focuses on initiating the software process improvement effort, including establishing a vision and identifying the community involved in the software development process. This phase generally coincides with the IDEAL model's Initiating phase for the software process improvement effort. The goal of this phase is to establish a vision and identify the community involved in the transformation. This is usually accomplished during one iteration, but may require more iterations to gain consensus.

For example, consider the organization discussed above whose customers are complaining that product features are not meeting their needs. During the Initiate phase, a vision is established with the various organizational departments and teams who are stakeholders to the software development process.

The ***Implement phase*** focuses on realizing software process improvements, including diagnosing and establishing software process improvements (best practices and automation) and acting to implement software process improvements (processes and automation usage models). This phase generally coincides with the IDEAL model's Diagnosing, Establishing, and Acting phases for the overall software process improvement effort. The goal of this phase is to evolve the vision and realize the transformation, including implementing software process improvements within the enterprise and software process changes within the community. This is usually accomplished across multiple iterations where each iteration involves some set of software process changes being deployed in the organization. For the organization mentioned above, during the Implement phase, the enterprise and the various organizational departments and teams who are involved in the software development process are iteratively transformed as further discussed in the Iterations and Activities section.

The ***Close phase*** focuses on learning form the software process improvement effort, including capturing lessons learned and recommendations. This phase generally coincides with the IDEAL model's Learning phase for the overall software process improvement effort. The goal of this phase is to close the transformation cycle, which may start another transformation cycle. This is usually accomplished during one iteration, but may require more iterations to reach closure. For the organization mentioned above, during the Close phase, the software process improvement effort is brought to closure.

The EIF's cycles and phases offer a very pragmatic general approach which emphasizes a management perceptive of software process improvement.

Iterations and Activities

A phase is composed of sequential time-boxed iterations. Each iteration results in a software process change increment. An iteration is a minor milestone where community members can influence and steer the software process improvement initiative. An activity is a primitive unit of work. Iterations establish an approach for deploying software process change to an enterprise, projects, departments, or other organizational unit using a human-oriented process for achieving sustainable software process change. Figure 2 shows the EIF's activities (vertical) which

address the community (left) and individual people (right). Each iteration involves some set of software process changes being deployed in the organization.

The ***Plan activity*** focuses on the community and establishing a delivery schedule. The ***Orient activity*** focuses on people and introducing mentoring/coaching. These activities generally coincide with the IDEAL model's Initiating phase for specific software process changes. This activity is generally driven by information pertaining to the overall software process improvement effort and previous iterations while explicitly being driven by the previous iteration.

The ***Install & Configure activity*** focuses on the community and installing and configuring the automation (tool). These activities include mentors/coaches delivering automation administration training, establishing the software process and usage model (information for configuring the tool to support the software process), implementing the usage model in the automation, and integrating the automation into the software development environment.

The ***Establish activity*** focuses on people and establishing an action plan. These activities include focusing on the current as-is and desired to-be state of the software process considering people, the software development process, and automation; and also focusing on an action plan (localized success plan) to transition capabilities. The plan is composed of activities involving mentors/coaches leading workshops and delivering software process and automation training, supporting project managers and teams (addressing questions, facilitating discussions, and steering), reviewing results and providing feedback, delivering software development process and automation expertise, and migrating software development content (requirements including features and user stories, tests including acceptance tests and unit tests, and so forth). These activities generally coincide with the IDEAL model's Diagnosing and Establishing phases for specific software process changes.

The ***Deploy activity*** focuses on the community and introducing the software process and usage model. The ***Mature activity*** focuses on people and collaborating against the action plan. These activities include mentors/coaches delivering software process and automation training and mentoring/coaching teams and individuals. These activities generally coincide with the IDEAL model's Acting phase for specific software process changes.

The ***Follow-up activity*** focuses on the community and evaluating progress. The ***Terminate activity*** focuses on people and concluding mentoring/coaching. These activities generally coincide with the IDEAL model's Learning phase for specific software process changes. This information generally drives the overall software process improvement effort and future iterations while explicitly driving the next iteration.

For the organization mentioned above, one iteration may involve working with the Sales and Operations (including Support) departments to understanding their involvement in the software development process; determining what software development process changes may be able to increase their involvement and positively impact customer satisfaction; and working to implement those software development process changes by modifying the existing software development process and perhaps introducing some automation. Another iteration may involve working with the Fulfillment department to modify their software development process such that customers are more involved, including more intense user involvement in requirements gathering and user acceptance testing. And, another iteration may involve working with the Marketing, Sales, Fulfillment, and Operations departments in order to modify their automated processes such that partners and customers can more readily be involved in the software development process.

The triangles in Figure 2 show the general distribution of activities across phases. The triangles generally show in which phase activities start, peek, and end, but they are not meant to be interpreted as strictly linearly increasing and decreasing. The vertical bar in the Initiate phase in Figure 2 shows an iteration, but iterations will span all phases. The diamond in Figure 2 shows the general distribution of effort across the lifecycle. The diamond generally shows that a transformation ramps up at the start of a cycle, reaches an optimum where all activities are being performed in parallel and as appropriate, and then ramps down at the end of the cycle.

The EIF's iterations and activities offer a very pragmatic general approach which emphasizes a human-oriented perceptive of software process changes.

The IDEAL Model and the Enterprise Implementation Framework (EIF)

The IDEAL model focuses on the management perceptive of software process improvement: how does the organization manage a software process improvement effort? However, the EIF focuses on the management perspective as well as the community's and individual people's perspective of software process improvement and change:

- How does the organization manage a software process improvement effort?
- How do the software process and automation changes impact the community and individual people?
- What are the necessary behavioral changes to ensure the software process improvement effort's success?

Generally, the IDEAL model's phases correspond to the EIF's cycle phases for the overall software process improvement effort as well as the EIF's iteration activities for specific software process changes. For an overall software process improvement effort, the EIF's Initiate phase corresponds to the IDEAL model's Initiating phase; the EIF's Implement phase corresponds to the IDEAL model's Diagnosing, Establishing, and Acting phases; and the EIF's Close phase corresponds to the IDEAL model's Learning phase. For specific software process changes within the overall software process improvement effort, the EIF's Plan and Orient activities correspond to the IDEAL model's Initiating phase; the EIF's Install & Configure and Establish activities correspond to the IDEAL model's Diagnosing and Establishing phases; the EIF's Deploy and Mature activities correspond to the IDEAL model's Acting phase; and the EIF's Follow-up and Terminate activities correspond to the IDEAL model's Learning phase.

The IDEAL model does not make a granular distinction between software process improvements versus software process changes. The EIF emphasizes that software process improvements are made at the organizational level while software process changes are made at the organizational unit level (projects, departments, and ultimately people). This distinction emphasizes the criticality of people as collaborating and contributing stakeholders in any software process improvement and quality effort.

The IDEAL model's Initiating, Diagnosing, Establishing, Acting, and Learning phases generally emphasize the management perceptive of software process improvement. However, the EIF uses the distinction between software process improvements and software process changes to further emphasize the impact on the community and individual people. The EIF's Initiate, Implement, and Close phases generally emphasize the management perceptive of software process improvement while the EIF's Plan, Install & Configure, Deploy, and Follow-up activities emphasize the community perceptive of software process changes and the EIF's Orient, Establish, Mature, and Terminate activities emphasize the individual people's perspective of software process changes. The community perceptive of software process changes involves

software process and automation changes while the individual people's perspective of software process changes involves adopting and leveraging the software process and automation changes. This emphasizes is critical since most software process improvement and quality efforts are less than successful due to insufficiently considering the impact on the community and individual people and people's role as collaborating and contributing stakeholders.

The IDEAL model's Acting phase, with its Pilot/Test Solution activity and Refine Solution activity, emphasizes iterative and incremental software process improvement. However, because the EIF's phases are grouped as cycles and activities are grouped as iterations, the EIF ultimately provides a more complete and scaleable framework for iterative and incremental software process improvement.

Ultimately, the EIF offers a more pragmatic framework for software process improvement than the IDEAL model due to its emphasis on a management perspective and human perspective, which focuses on the community and individual people, as well as its emphasis on iterative and incremental software process improvement.

Conclusion

Unequivocally, organizational improvement and change is a necessity where people are and will remain the "original ingredient" necessary for success. However, with a better understanding of the IDEAL model, individuals, teams, and organizations have a roadmap which may be further contextualized to increase the probability of success. The EIF as described in this paper provides a broader framework through which to leverage the IDEAL model. Furthermore, it is experience, experimentation, and application of the IDEAL model and EIF that will enable us to realize their benefits.

References

[1] Software Engineering Institute (http://www.sei.cmu.edu)

[2] The IDEAL Model (http://www.sei.cmu.edu/ideal)

Analyst Pro is an affordable, simple, powerful, and easy-to-use requirements tracking and management tool. It provides effective requirements tracking, importing, exporting, diagramming, and baselining. It also provides a traceability matrix to easily and efficiently show the impact of changing requirements on the system and process as a whole.

www.analysttool.com

Sparx Systems release Enterprise Architect v5.0. New features include MDA Style Model Transformations, advanced Version Control, RTF Report Generation and Floating License Management (Corporate Edition). Value and power for the whole design team, EA v5.0 is the latest generation UML 2.0 CASE tool. For details and to download the 30 day trial visit:

www.sparxsystems.com

Advanced Reuse and Component-Based Development With MKS. With Web Services and component-based development on the rise, code re-use initiatives are presenting some major challenges for organizations today: How do you identify the components and versions of components for a given release? Who owns a software component? Who owns the testing of that component? How do you apply a change across all instances of a component? FREE white paper from MKS with answers to your component-based development challenges.

www.mks.com

The Software Test & Performance Conference focuses on testing and performance issues for software developers, , development managers, test/QA managers and senior test professionals. This year's conference, scheduled for November 1-3, 2005 at the Roosevelt Hotel in New York City. For more information, and to register, go to

www.stpcon.com

METHODS & TOOLS is published by **Martinig & Associates,** Rue des Marronniers 25,
CH-1800 Vevey, Switzerland Tel. +41 21 922 13 00 Fax +41 21 921 23 53 www.martinig.ch
Editor: Franco Martinig ISSN 1661-402X
Free subscription on : http://www.methodsandtools.com/forms/submt.php