Extreme Programming and

Embedded Software Development

By James Grenning

Every time I do a project, it seems we don't get the hardware until late in the project. This limits the progress the team can make. About all we can do is create speculative design documents, not exactly demonstrable hard evidence of progress. Writing software is hard. Not writing it until late in the development cycle is suicide. A team using Extreme Programming[BECK] will make concrete progress in embedded software development early in the development cycle. Even prior to hardware availability. We can provide regular deliveries of executable software early in the project. We can invest a lot less in written documentation and more in the executable software. Even without the target platform, we will have a rich suite of tests demonstrating our progress.

This article explores one major advantage XP could provide to embedded software developers: the ability to achieve meaningful progress prior to hardware availability. The article also looks at some of the difficulties of applying XP to embedded software development.

Embedded Software Development

You're an embedded systems software developer. Maybe you have deadlines given to you that have very little to do with what it would actually take to deliver the product. Then an incomplete requirements document is tossed over the wall to you. Do your requirements or priorities change unexpectedly? Every now and then, you get to do a little programming. You throw the software over the wall to a testing team and they toss back a database of bugs. The end user of your product changes their mind on the needed features after you deliver the product. You probably work long hours to try to meet your deadlines, but you don't always meet them. You don't have the final hardware if you have any hardware at all. Maybe not all these have happened to you, but I'll bet most of them sound familiar.

Over the last 20 years, embedded software developers have used processes to prevent some of the problems mentioned above. We practiced defect prevention and used documentation and reviews as the main tools. We have prevented many defects, but is there a better way? Processes were supposed to keep project from being late, but late projects still plague the industry. Processes were supposed to help us deliver the right functionality the first time, but often the desired functionality is not easily determined up front.

There are differences between embedded software and non-embedded software. The development machine architecture is often different from the target machine. The hardware for the target machine is often developed concurrently with the software, and therefore not available until late in the project. There may be real-time constraints. There are likely to be threading issues. Typical human-computer interfaces are not used. The fact that a computer is operating the machine is often hidden from the user. There may be resource constraints

grenning@objectmentor.com

such as limited memory space or processing power. There may be safety issues. There certainly are additional complexities in developing embedded software.

What is Extreme Programming?

Kent Beck, author of Extreme Programming Explained says, "XP is a light-weight methodology for small-to-medium-sized teams developing software in the face of vague or rapidly changing requirements." Simply stated, XP is a set of values, rights and best practices that support each other in incrementally developing software.

XP values Communication, Simplicity, Feedback and Courage.

Programmers and Customers have the right to do a quality job all the time and to have a real life outside of work.

XP is a collection of best practices. Some may sound familiar. Some may sound foreign.

Customer Team Member – Teams have someone (or a group of people) representing the interests of the customer. They decide what is in the product and what is not in the product.

Planning Game - XP is an iterative development process. In the planning game, the customer and the programmers determine the scope of the next release. Programmers estimating the feature costs. Customers select features and package the development of those features into small iterations (typically 2 weeks). Iterations are combined into meaningful end user releases.

User Story – A User Story represents a feature of the system. The customer writes the story on a note card. Stories are small. The estimate to complete a story is limited to no greater than what one person could complete within a single iteration.

Small Releases – Programmers build the system in small releases defined. An iteration is typically two weeks. A release is a group of iterations that provide valuable features to the users of the system.

Acceptance Testing – The customer writes acceptance tests. The tests demonstrate that the story is complete. The programmers and the customer automate acceptance tests. Programmers run the tests multiple times per day.

Open Workspace – To facilitate communications the team works in an open workspace with all the people and equipment easily accessible.

Test Driven Design – Programmers write software in very small verifiable steps. First, we write a small test. Then we write enough code to satisfy the test. Then another test is written, and so on.

Metaphor – The system metaphor provides an idea or a model for the system. It provides a context for naming things in the software, making the software communicate to the programmers.

grenning@objectmentor.com

Simple Design – The design in XP is kept as simple as possible for the current set of implemented stories. Programmers don't build frameworks and infrastructure for the features that might be coming.

Refactoring – As programmers add new features to the project, the design may start to get messy. If this continues, the design will deteriorate. Refactoring is the process of keeping the design clean incrementally.

Continuous Integration – Programmers integrate and test the software many times a day. Big code branches and merges are avoided.

Collective Ownership – The team owns the code. Programmer pairs modify any piece of code they need to. Extensive unit tests help protect the team from coding mistakes.

Coding Standards – The code needs to have a common style to facilitate communication between programmers. The team owns the code; the team owns the coding style.

Pair Programming – Two programmers collaborate to solve one problem. Programming is not a spectator sport.

Sustainable P***ace*** –The team needs to stay fresh to effectively produce software. One way to make sure the team makes many mistakes is to have them work a lot of overtime.

Implications

If we are keeping our design simple, working in small iterations, and developing features in some order dictated by the customer the programming team will have to be skilled at evolutionary design. We have to start with a simple feature and a simple implementation and evolve that into a viable product. Let's see how that works.

Getting Started

Early in an embedded project, the programmer often does not have any real target hardware on which to run the application. When using a traditional design up front process the programmer spends time writing and reviewing design documents. When using XP we start on the customer's most valuable story, implement it, and then implement the next one. This process continues until there is a system with enough functionality to ship.

We don't have many documents to write, we don't have any hardware to play with, but we have to get started. What do we do? We start exploration. We do Exploration to make a first guess at the scale of the project, set a direction, and develop the first release plan. The goal is to quickly move to development.

In exploration, we set the boundaries of the system and start to write as many stories for the system as we can. The programmers then estimate the effort to build each story. Stories should be no bigger than the effort needed for one person the implement the story in two weeks. Now wait a minute! How can we develop anything of value in two weeks? The

grenning@objectmentor.com

value may not be releasable value, but is demonstrable value to the person or persons playing the role of the customer.

The customer will have to look inside the system and write stories about the behavior of the system. For example, let's say we are developing a telephone switch for small businesses, also known as a Private Business Exchange (PBX). The big story:

> "Telephone extension can place a local phone call but going off hook and dialing a 9, the cheapest line available is chosen from the trunk lines, the call duration is recorded in the PBX database for departmental charge back…"

OK, there could be big stories like this one. To deliver this PBX story we'll need working hardware and software. It may be months or years from being able to deliver that. The key to steering an XP team is to break the big stories up into small, architecture spanning, verifiable bits of functionality.

Lets try some smaller stories:

- Off-hook generates dial tone
- On-hook
- Extension calls extension
- Extension calls busy extension
- Flash, call transfer
- Flash, three way call
- Extension goes off-hook dials "9" and a line is available
- Extension goes off hook and dials "9" no lines are available
- Call records are filed by extension
- Round robin trunk group
- Priority trunk group
- Trunks groups are chosen based on number called

These stories are parts of the bigger story. They provide definite behavior. The customer can choose stories that must be included in the system, stories that are the most important and best understood. Each story demonstrates a small part of the system functionality except for one thing…

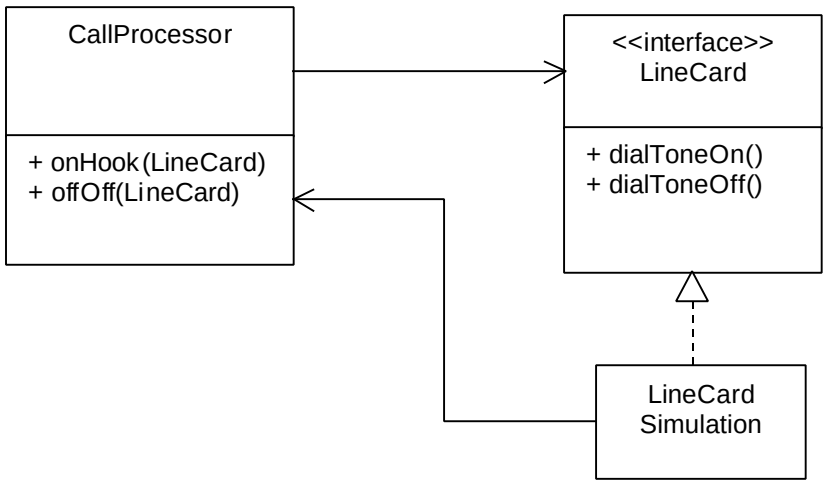
Hardware is Not Ready

XP's focus on automated testing can really help get the embedded software effort off the ground early in the product time line. I believe this is a key benefit to embedded software developers. So, how can we start developing the stories without the target platform? Good design techniques and simulation provide the key to solving this problem. This application can start to be grown and when a feature bumps into some non-existent hardware, we have to create an interface and simulate its behavior. We need to use abstraction. Bruce Douglas defines abstraction as: "the process of identifying the key aspects of an entity and ignoring the rest."[DOUGLASS] In our case, this key aspect is the interface. If our application could have any interface to the hardware we want, let's design in a really convenient interface, an interface that shields the application from the implementation details. This should be easy right now, considering we don't have any implementation details. The details will likely

grenning@objectmentor.com

change as hardware evolves, but the abstract view of the hardware interface might not have to.

In a way, not having hardware at the beginning of the project is an advantage. It forces the embedded software developer to look at the problem more abstractly. It keeps the API of the hardware interface safely isolated from the application. If we want to start developing the call processing part of the application, and develop the application logic for the "Off-Hook" story we could start with a design like this:



One simple call processing system can get onHook and offHook events from a line card, then tell the line card to turn on dial tone. We don't have a real line card yet, or know everything it needs to do. That does not stop us from starting a design with what we know about the line card. We bumped into some edge of the system that does not exist yet, so we create an interface. We also know that call processing is going to get very complex and its implementation will grow. Nevertheless, that does not stop us from an initial simple implementation that we can use to start testing and demonstrating progress.

How do we test this? We need a unit test tool[1]. There are many free unit test tools available. For this example, we'll use Java, and JUnit[2].

[1] There is a good catalog of unit test tools at www.xprogramming.com. For C++ development, look at CppUnit.
[2] www.junit.org

grenning@objectmentor.com

This test says that dial tone is initially off. The CallProcessor is told that an off-hook was detected on the given line card. We then expect that the line card would be generating dial tone.

```
//CallProcessorTest.java
import junit.framework.TestCase;

public class CallProcessorTest extends TestCase {
    public CallProcessorTest(String name) {
        super(name);
    }

    public void testDialTone() {
        CallProcessor cp = new CallProcessor();
        SimulatedLineCard slc =
                          new SimulatedLineCard();

        assertTrue(slc.isDialToneOn() == false);
        cp.offHook(slc);
        assertTrue(slc.isDialToneOn());
    }
}
```

Here is our first cut at the LineCard interface. We can expect that this will evolve into something more complex with more features.

```
//LineCard.java
public interface LineCard {
    public void dialToneOn();
    public void dialToneOff();
}
```

The SimulatedLineCard implements the LineCard interface and adds a method to check the state of the simulation.

```
//SimulatedLineCard.java
public class SimulatedLineCard implements LineCard
{
    boolean dialToneOn = false;
    public SimulatedLineCard() {
    }

    public void dialToneOn() {
        dialToneOn = true;
    }
    public void dialToneOff() {
        dialToneOn = false;
    }
    public boolean isDialToneOn() {
            return dialToneOn;
    }
}
```


grenning@objectmentor.com

The CallProcessor is told of on-hook and off-hook events and given a reference to the LineCard that detected the event.

```
//CallProcessor.java
public class CallProcessor {
    public CallProcessor() {
    }

    public void onHook(LineCard lc) {
        lc.dialToneOn();

    }
    public void offHook(LineCard lc) {
        lc.dialToneOff();
    }
}
```

We have started our design and have a couple of interesting architectural elements already: the LineCard interface and the CallProcessor. We also should have a working test! Time to celebrate! The test that simulates an off-hook, then checks that there is dial tone. Right now, it gives us firm footing and a green bar in JUnit, indicating the test passed. Oops, the test failed. We get a red bar! What is wrong with this code? The test shows that when an off-hook is simulated, the simulated line card is not generating dial tone as expected. The error is easy to find[3]. We find that these minor errors happen all the time. This bug would have been a lot harder to find after a week of coding with no testing. This test cost us very little. The test becomes more valuable with time. Here is the corrected code.

```
//CallProcessor.java - Version 2
public class CallProcessor {
    public CallProcessor() {
    }

    public void onHook(LineCard lc) {
        lc.dialToneOff();

    }
    public void offHook(LineCard lc) {
        lc.dialToneOn();
    }
}
```

Well, how should the system detect the off-hook and on-hook events? That sounds like a job for the line card. We know we don't have a line card, so let's evolve our design to look more like we expect it later. When a LineCard detects an off-hook, it has to let the CallProcessor know, which in turn will tell the LineCard to generate dial tone. First let's evolve the tests to match the evolved architectural vision. Notice we are evolving the LineCard to know about its CallProcessor. The test looks like this:

[3] The logic was backwards in CallProcessor.

grenning@objectmentor.com

```
//CallProcessorTest.java - version 2
import junit.framework.TestCase;

public class CallProcessorTest extends TestCase
{
    public CallProcessorTest(String name) {
        super(name);
    }

    public void testDialTone() {
        CallProcessor cp = new CallProcessor();
        SimulatedLineCard slc = new SimulatedLineCard(cp);

        assertTrue(slc.isDialToneOn() == false);
        slc.simulateOffHook();
        assertTrue(slc.isDialToneOn());
    }
}
```

The revised simulation shows how an off-hook results in the SimulatedLineCard telling the CallProcessor about the event.

```
//SimulatedLineCard.java - version 3
public class SimulatedLineCard implements LineCard
{
    boolean dialToneOn = false;
    CallProcessor callProcessor;
    public SimulatedLineCard(CallProcessor callProcessor) {
        this.callProcessor = callProcessor;
    }

    public void dialToneOn() {
        dialToneOn = true;
    }
    public void dialToneOff() {
        dialToneOn = false;
    }
    public boolean isDialToneOn() {
            return dialToneOn;
    }
    public void simulateOffHook(){
        callProcessor.offHook(this);
    }
}
```

This test and simulation is very simple. It is one of its strengths. It demonstrates our newborn architecture. CallProcessing is very simple right now. The interaction between a LineCard and CallProcessing is simple. As the design grows in complexity, the tests that cover it will also grow. As the design and the tests grow, we will continue to test that an off-hook results in generating dial tone. Some future change may break this test. We'll be ready to catch that side-effect defect. This test and ones like it will catch defects that would otherwise make it into production, or cost expensive review time to discover.

grenning@objectmentor.com

This process can continue to evolve the relationship between the LineCard and the CallProcessor. Later when we get a real line card, we can adapt its API to meet the architecture we have started to develop. This focus on testing leads to decoupled software.

We have completely ignored the complexities of concurrency, scalability and performance. We're limiting the scope in the iteration so we can stay focused. Some future stories will drive the need for concurrency. We won't try to solve the whole problem at once. Modularity will help us reshape the software, as we need.

Acceptance Tests

The customer can start to develop acceptance tests. Acceptance tests must be automated. A common way to do that is to provide the customer team with a scripting language to feed various traffic scenarios into the system. For example

```
LINECARD 1 OFFHOOK
VERIFY LINECARD 1 DIALTONE
LINECARD 1 ONHOOK
VERIFY LINECARD 1 NODIALTONE
```

At some point, we will want the customer to really pick up phones and listen for dial tone, etc. For the most part, we want to automate our tests. That means that we will have to slice away a bit of the outer layer and replace it with a testing interface. We would create a piece of code that reads and interprets the script, then calls the appropriate method on the application. The acceptance test captures and verifies the responses. Our architecture allows us to substitute different implementations of LineCard. We can create another implementation of LineCard for acceptance testing. It simulates the needed behavior and writes out its actions to a log file. The script's VERIFY command checks for the desired result.

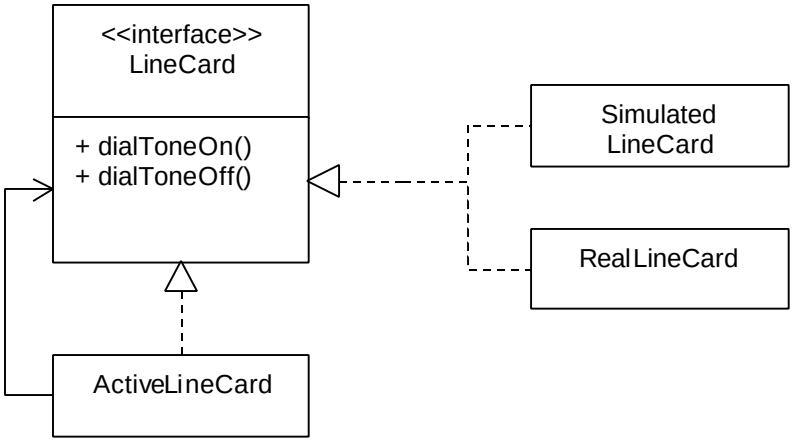
Much of the embedded application logic can be developed and tested, then verified by the customer team through techniques like this. We don't even need real hardware to make progress. Even when we have real hardware, we will want to continue to test the system through our automated scripts. We'll want to run them multiple times per day. They will run faster and will not require target hardware or special external test equipment. We'll try to test everything we can without the hardware and without manual intervention. Of course, there will be some testing of the fully integrated system once we have one.

Concurrency

As this application grows, there will undoubtedly be stories that lead us to need concurrent processing, and multiple threads of execution. We will have all this code without threads, and our customer plays the story that breaks our single threaded model. We could go and start adding threads to our application classes. If we mix threads up with the application logic, it will get hard to test. Also the threading structure of a design often needs to evolve over the life of the application. If the threads are intertwined with the application, it is much harder to untangle when concurrency needs change.

grenning@objectmentor.com

This leads us to use separation of concerns to keep our application logic separate from the concurrency model. For starters, lets say that each physical line card will have its own LineCard instance, whose methods will run in the LineCard's thread. We could employ the idea of an Active Object[SCHMIDT] in our design. An Active Object runs in its own thread.

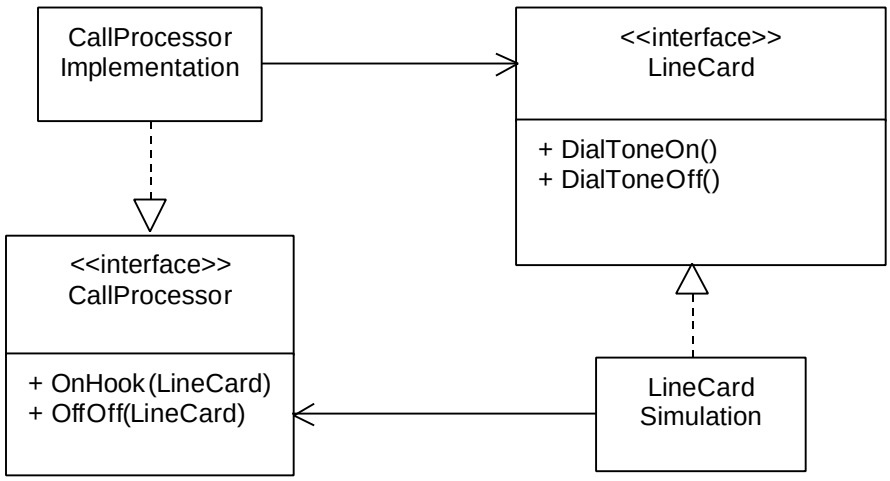


The ActiveLineCard would have its own thread. Calls to it will cause the specific line card request to be executed in the ActiveLineCard's thread. The ActiveLineCard would delegate the calls to a RealLineCard or a SimulatedLineCard through the LineCard interface. The threading logic would be contained in the ActiveLineCard. By keeping the tests for ActiveLineCard and RealLineCard separate, we can simplify testing.

grenning@objectmentor.com

Evolutionary Designs

It's tempting to add an interface to the CallProcessor to allow more decoupling. In XP we wait until the additional architectural elements have a clear need, thus keeping excess complexity out of the system. We probably will come across some test, or new user story that will force the need for the decoupling. If we're wrong and the need never arises, then we don't pay for the added complexity. Changing the design to incorporate the interface is a small change, but we'll wait until we need it. We would use the "Extract Interface"[FOWLER] refactoring once we determined it is needed.



A good design is important. In XP, designs evolve. As a mater of fact, all designs evolve. If we neglect refactoring (hacking happens), the design evolution will quickly destroy the design. The design-centric XP practices (Test Driven Design, Simple Design, Refactoring, Metaphor) provide support for changing the software design. A simple design with a consistent metaphor is easier for us to understand. When the design starts to deteriorate, we can use refactoring techniques to clean up the messy parts of the design. We can refactor mercilessly because we have the safety net the automated tests provide. These techniques give us confidence that we can evolve the design. We are going to make design decisions incrementally. The modular, loosely coupled system we have started will be easier to evolve. We'll resist speculative design changes.

Except for the smallest system, no amount of design up front will be correct. The concept of evolutionary design faces the fact that the design will change over time. Many systems start out with a good clean design. New requirements pummel the system. The existing design cannot cleanly support the new requirements. The design degrades. Hacking happens. The good design is lost forever. Therefore, XP teams refactor to keep the design alive.

Real Time

Embedded systems often have real time performance constraints. When the customer plays specific performance stories, we can devise unit and acceptance tests. These tests will not be easy to write, but neither is it easy to test the real time performance in a live system.

grenning@objectmentor.com

Let's say we determine that a specific interrupt service routine must complete within a specific time period. We can write unit tests to check the execution time of a specific critical code section. This test sounds like it would only run on the target platform. We may have to ask our hardware designer for a timer we can use for these kinds of tests.

We can create acceptance tests that simulate loads and event sequences that may be difficult to generate on a live system. Our test could pump in a known traffic load such as N calls per second with M LineCards. We then monitor the amount of CPU used during various loads, and count the number of lost calls. In addition, we could sample the behavior of specific LineCards during the load test and verify that they behave properly during load.

What do we do if we discover a performance problem? "You optimize because you have a problem in performance… Without performance data, you won't know what to optimize, and you'll probably optimize the wrong thing."[NEWCOMER]. Resist doing speculative optimizations. Keep the design clean and modular. If the performance tests fail, collect data to point you to the specific performance problem and solve it. We can fix performance problems more readily in a modular system.

Development Tools

One of the biggest obstacles to using XP on embedded systems development is the availability of tools. Some specific development system problems you might encounter are:

- No OO programming language (Java, C++)
- Inability to run code on the build system
- Compiler incompatibility
- There is no unit test tool like JUnit or CppUnit for your development or execution environment

No OO programming language

An OO programming language is not mandatory to do XP. OO facilitates module decoupling. Without OO it is very difficult to test pieces of the system independently. There is no easy runtime substitutability. If you do not have OO, you may have to rely more on acceptance testing than unit testing as it will be difficult to automate unit tests. Some link-time substitutability may be helpful for some tests. Don't be discouraged. XP values and many of the practices are still open to you.

Inability to run code on the build system

If the development system and the target system are different execution environments, it will help greatly to be able to build for both environments. Often target systems are scarce and extremely expensive. We have to sign up for time on the target hardware. Therefore, running simulations and unit tests on the development system is very important.

Compiler incompatibility

If it is impossible to run the code on the development system, then all execution must be done on the target platform. This will slow down development. XP relies on a fast change, compile, and test cycle. Add to that a download step. The cycle needs to run every few minutes. A long download will sap productivity.

grenning@objectmentor.com

Compiler incompatibility will make it difficult to port a unit test tool. There are many C++ test kits. Some use less C++ features than other and may be easier to port.

No unit test tool

Writing a unit test tool won't be too bad. It is just a place to hang your unit tests. A starter test framework is just a list of Boolean functions. You can evolve it per the practices of XP.

Safety Critical Systems

Safety critical systems can benefit from the application of XP techniques. The focus on automated testing can catch defects. Will you need to do more than plain vanilla XP in a safety critical system? I think Steve McConnell sums it up nicely: "Generally speaking, widely distributed products have to be more carefully developed than narrowly distributed products. Products whose reliability is important must be developed more carefully than products whose reliability doesn't much matter."[MCCONNEL] I think I would like the people developing safety critical systems to try XP and evolve it to meet their needs for safety criticality.

Conclusion

XP attempts to lower the cost of change curve[BECK2]. Embedded systems development can benefit from the practices of XP. One significant benefit for embedded software developers is the ability to make meaningful progress prior to hardware availability, supported by the practices of unit and acceptance testing.

Tools may be problematic for some embedded systems development. Deploying embedded software may still have high costs, but XP can help to lower the cost of change for much of the embedded software development cycle.

[BECK] Beck, Kent, Extreme Programming Explained, Addison Wesley, 1999

[DOUGLASS] Douglass, Bruce P., Real Time UML – Developing Efficient Object for Embedded Systems, Addison Wesley, 1998, p.18

[SCHMIDT] Schmidt, Douglass, Lavender, R. Greg, Active Object An Object Behavioral Pattern for Concurrent Programming, http://www.cs.wustl.edu/~schmidt/PDF/Act-Obj.pdf

[FOWLER] Fowler, Martin, Refactoring Improving the Design of Existing Code, Reading, MA, Addison Wesley, 1999 p.341

[NEWCOMER] Newcomer, Dr. Joseph M. Optimization: Your Worst Enemy. http://www.codeproject.com/tips/optimizationenemy.asp

[MCCONNEL] McConnell, Steve, From the Editor, IEEE Software, November/December 2001, p.8

[BECK2] Beck, Kent, Extreme Programming Explained, Addison Wesley, 1999, p.21

grenning@objectmentor.com