Extreme Programming and Embedded Software Development

Class #

By James Grenning

There are two (almost) universal truths of embedded systems development. The first truth is shared with software development projects of all kinds. The date is fixed long before the requirements are known. The second truth is unique to embedded development. The target hardware is not available until late in the project. Without hardware in the early part of the project, progress can be limited. Teams are limited to the creation of speculative requirements and design documents. The software development team is often on the critical path for delivering the product. Wouldn't it be helpful if the team could make concrete progress on the software earlier in the development cycle?

What if there was a way to make concrete progress early and often in your embedded project? Can software development get off the critical path? Would you be interested if both those questions could be answered affirmatively? If so, then you probably want to know more about applying the techniques of Extreme Programming[BECK] to embedded systems development.

Writing embedded software is hard. Not writing it until late in the product development cycle is suicide. A team using Extreme Programming (XP) can make concrete progress in embedded software development continuously throughout the development cycle; even prior to hardware availability. XP is an iterative development technique. You can think of Barry Boehm's Spiral Model as a great-grandfather to XP[BOEHM]. Iterative development admits that there are risks and unknowns in software development. XP's techniques support the evolutionary development of software as does Boehm's Spiral Model. XP's techniques result in a high quality design and highly maintainable code while supporting the evolution of the system.

Iterative development allows requirements analysis to occur in parallel with software development. By delivering valuable features of the code regularly in the development cycle, team velocity is determined and risks are reduced by providing measurable progress through delivery of working code. Work can be decoupled from scarce target system hardware.

This article explores advantages XP could provide to embedded software developers. The article also looks at some of the difficulties of applying XP to embedded software development.

Embedded Software Development

As software developers, we are used to being given deadlines that have very little to do with what it takes to deliver the product. We know when the software must be delivered. We just don't know what it is supposed to do. Often we'll get a feature implemented just like the spec says, but when marketing sees the product; they now know the spec was not quite right. We lose a customer, the feature priorities change. We work long hours to deliver against the dates, but do not always make it. Prototype hardware is scarce or unavailable until late in the development cycle. We may have to schedule time to test on this limited resource, or fight for capital investments for more test beds. Too often this is the software developer's world. We can complain about it, or we can get good at dealing with our world. I prefer the latter.

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 many still escaped our best efforts. 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 that can make embedded software development even harder. The development machine architecture is often different from the target machine. The hardware for the target machine is usually developed concurrently with the software, and therefore not available until late in the project. There may be real-time constraints, concurrent processing, and safety issues. Typical human-computer interfaces are not used and the computer operating the machine is hidden from the user. Resource constraints such as limited memory space or processing power are the norm.

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.

The team has the right to do a quality job all the time, be honest with each other and to have a real life outside of work. Team members communicate with each other openly and honestly. Problems are solved simply, designs are kept simple. We get feedback by writing tests and showing the customer what was asked for. We need courage to be honest about what is possible, to improve a design when the pressure is on and to challenge the status quo.

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. I'll refer to this group of people as the customer or the customer team in this paper. The customer is responsible for acceptance testing the product. This means the customer team needs skilled testers.

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. If the story is too big to complete in an iteration, split it into smaller stories. If you are used to use cases, a story is like the name of a use case. The details of the use case are in the acceptance tests.

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 estimate the effort to complete each story. Customers select stories and package them into iterations. Stories are prioritized by the customer according to their business value and cost.

Small Releases – Programmers build the system in small releases. 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 team writes acceptance tests. The tests demonstrate that the story is complete. Acceptance tests provide the details behind the story. Tests are defined prior to completing a story. 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, a small test is written, followed by just 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, helping the software communicate to the programmers.

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. Frameworks and infrastructure evolve as the application evolves.

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

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

Pair Programming – Two programmers collaborate to solve one problem. Programming is not a spectator sport. Both programmers are engaged in the solution of the problem at hand.

Collective Ownership – The team owns the code. Programmer pairs modify any source code as needed. 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. Individuals may have to give up their favorite conventions to conform to the team style. The standard is determined by the team.

Sustainable P*ace* –The team needs to stay fresh to effectively produce software. Creating a product is like running a marathon. We sprint at times during the marathon, but we are careful to reserve our strength so that we can finish the race.

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 has 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 a Project 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 work that can be completed in an iteration. We implement it, and then move on to the next most important set of stories. This process continues until the system has enough functionality to ship.

We don't have any hardware to play with, but we have to get started. What do we do? The first activity of an XP project is exploration. The exploration's goal is to determine the scale of the project, set a direction, and develop the first release plan. We want to quickly move to development and start putting our efforts into the creation of the software.

In exploration, we set the boundaries of the system and 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? It can be hard to break stories into such small pieces, but that's what we do. Stories are fine grain features. Small stories allow the customer to exercise a lot of control over the project. Ideally a story is valuable to the end user in its own right, or is a demonstrable part of a bigger feature.

The customer writes 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… and the system can handle 2000 extensions with outside calls equal to the number of trunks and inside calls involving 25% of the extensions"

That's a big 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 units 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

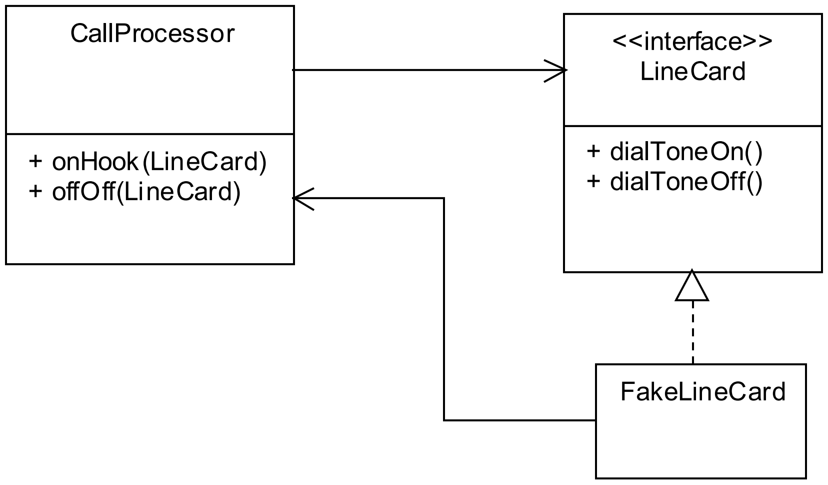
These stories are parts of the bigger story. They provide definite behavior. The customer can choose stories that must be included in the system, stories that are the most important and best understood. Each story demonstrates a small part of the system functionality except for one thing…

Hardware is Not Ready

XP's focus on automated testing can really help get the embedded software effort off the ground early. 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. Stories are vertical slices through the architecture. When a story requires access to some non-existent hardware, we create an interface and a faked-out version of hardware in software. If our application could have any interface to the hardware, a really convenient interface is best. So, we start with a simple, convenient interface. This interface shields the application from the implementation details. It should be pretty easy to ignore the implementation details right now, considering we don't have any implementation details. The details will change as hardware evolves, but the abstract view of the hardware interface should be able to hide the details.

This may sound crazy, but 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.

Let's start developing the call processing part of the application. The "Off-Hook" story could start with a design like this:



One simple call processing system can get offHook and onHook events from a line card, then tell the line card to turn on or off dial tone. We don't have a real line card yet, and we don't need to deal with everything it eventually must do. We start with a simple design. We bump 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. It's implementation will grow. Nevertheless, that does not stop us from an initial simple implementation that we can use to work out architectural ideas and demonstrating progress through our tests. By the way, the testing pattern being used here is called Mock Object.[MACKINNON]

How do we test this? We need a unit test tool. There are many free unit test tools available[1]. For this example, I'll program in C++ and use CppUnitLite for unit testing. I'll be leaving out some C++ to keep the example brief (such as destructors, constructors and assignment operators). I'll also put all the code in the header file, for now.

Here is our initial test. We're using the TEST macro from CppUnitLite. This macro defines a test and installs it into the CppUnitLite TestHarness. This test tells us about the initial state of the system. Initially dial tone is off. The CallProcessor is told that an off-hook was detected on the given line card. We then expect that the line card is generating dial tone. The CHECK macros are simple Boolean checks. The macro succeeds when the value evaluates to true. The TestHarness notifies the user of any errors.

```
//CallProcessorTest.cpp
#include "TestHarness.h"
#include "CallProcessor.h"
#include "FakeLineCard.h"

TEST(CallProcessorTest, DialTone)
{
    CallProcessor* cp = new CallProcessor();
    FakeLineCard* lc = new FakeLineCard();

    CHECK(lc->isDialToneOn() == false);
    cp->offHook(lc);
    CHECK(lc->isDialToneOn());

    delete cp;
    delete lc;
}
```

Here is our first cut at the LineCard interface. The interface is implemented with pure virtual member functions in C++. We can expect that this will evolve into something more complex with more features as stories are added to the system. But don't worry about it yet.

```
//LineCard.h
class LineCard {
    public:
         virtual void dialToneOn() = 0;
         virtual void dialToneOff() = 0;
}
```

The FakeLineCard implements the LineCard interface and adds a method to check the state of the simulation. The FakeLineCard can be substituted anywhere a LineCard is expected.

```
//FakeLineCard.h
#include "LineCard.h"


class FakeLineCard : public LineCard
{
  public:
    FakeLineCard()
    : dialToneOn(false) {
    }

    void dialToneOn() {
        dialToneOn = true;
    }
    void dialToneOff() {
        dialToneOn = false;
    }
    boolean isDialToneOn() {
        return dialToneOn;
    }
  private:
    bool dialToneOn;

}
```

The CallProcessor is told of on-hook and off-hook events and given a reference to the LineCard that detected the event. CallProcessor has no idea if it is working with a real or a fake line card.

```
//CallProcessor.h
#include "LineCard.h"

class CallProcessor {
  public:

    void onHook(LineCard* lc) {
        lc->dialToneOn();

    }
    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 simulates an off-hook, and then checks that there is dial tone. Right now, it gives us firm footing and a report of no errors from CppUnitLite. Wait a second! The test failed. CppUnitLite reports an error! 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. Oh! I see it. I had the logic backwards in CallProcessor. I find that these minor errors happen to me all the time. How about you? This bug would have been a lot harder to find after a day or a week of coding. This test cost us very little, but already paid me back. The tests keep on giving. Every time a change is made all the tests are run. Showing any unexpected side affects immediately. Here is the corrected code.

```
//CallProcessor.h - Version 2

#include "LineCard.h"

class CallProcessor {
  public:

    void onHook(LineCard* lc) {
        lc->dialToneOff();

    }
    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 tells the LineCard to generate dial tone. First let's evolve the tests to match the architectural vision. Notice we are evolving the LineCard to know about its CallProcessor. The test looks like this:

```
//CallProcessorTest.cpp - version 2
#include "TestHarness.h"
#include "CallProcessor.h"
#include "FakeLineCard.h"

TEST(CallProcessorTest, DialTone)
{
    CallProcessor* cp = new CallProcessor();
    FakeLineCard* lc = new FakeLineCard(cp);

    assertTrue(lc->isDialToneOn() == false);
    lc->simulateOffHook();
    assertTrue(lc->isDialToneOn());

    delete lc;
    delete cp;
}
```

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

```
//FakeLineCard.h - version 3
#include "LineCard.h"


class FakeLineCard : public LineCard
{
  public:
    FakeLineCard(CallProcessor cp)
    : dialToneOn(false)
    , callProcessor(cp) {
    }

    void dialToneOn() {
        dialToneOn = true;
    }
    void dialToneOff() {
        dialToneOn = false;
    }
    boolean isDialToneOn() {
        return dialToneOn;
    }
    void simulateOffHook(){
        callProcessor->offHook(this);
    }
  private:
    bool dialToneOn;
    CallProcessor* callProcessor;
}
```

This test and simulation is very simple. It is one of its strengths. It demonstrates our newborn architecture. CallProcessor is very simple right now but it is the real CallProcessor. The interaction between a LineCard and CallProcessor is simple. When we get a real line card we'll evolve the line card design to support the new version of hardware. The CallProcessor won't know or care if it uses a real or simulated LineCard. We continue to test the CallProcessor with the simulation. As the design grows in complexity, the tests that cover it also grow. CallProcessor will undoubtedly grow and divide into other classes. As the design and the tests grow, we 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 catch defects that otherwise make it into production, or at least have to be found some other more expensive way.

A very interesting side affect of using test driven development, is that progress is much more predictable. TDD is more efficient and predictable than code-now-debug-later programming. What is the average cost of fixing a simple bug at your company?

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 developed. This focus on testing leads to modular, decoupled software. I expect that CallProcessor will grow in complexity and need to be split. We can wait, it will not be that hard to do.

We have completely ignored the complexities of concurrency, scalability and performance. We limit the scope of 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 help us reshape the software, as we need.

Acceptance Tests

The customer develops acceptance tests. Acceptance tests must be automated. Acceptance tests demonstrate to the customer team that the agreed upon functionality is in the system. 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
```

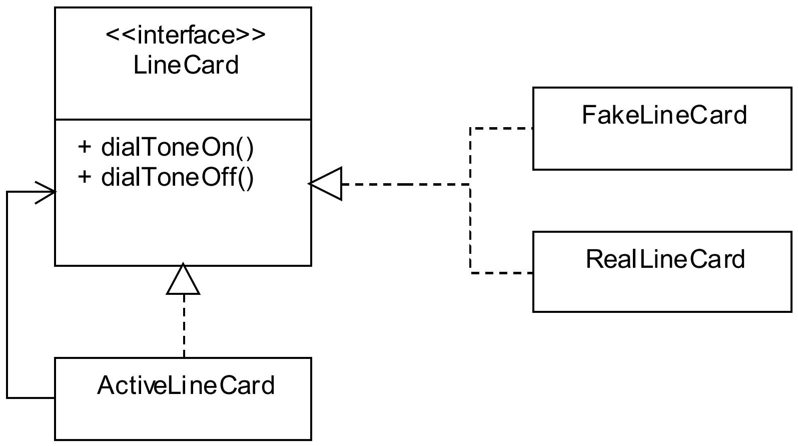
At some point, we want the customer to pick up phones and listen for dial tone, etc. Manual tests may be needed, but they are not consistent and repeatable. Automated tests are the goal. That means that we have to slice away a bit of the outer layer and replace it with a testing interface. We create a piece of code that reads and interprets the script, then calls the appropriate methods on the application. The acceptance test harness captures and verifies the responses. Because of the flexibility of our architecture LineCard implementations can stand in for each other. During acceptance testing, another LineCard implementation simulates the behavior needed to keep the CallProcessor happy and writes out its actions to a log file. The test script's VERIFY command checks for the desired result.

Much of the embedded application logic can be developed and tested. Application logic is verified by the customer team through the techniques described above. We don't need real hardware to make progress. Even when we have real hardware, we continue to test the system through automated scripts. We run them multiple times per day. The tests run fast and do not require target hardware or special external test equipment. We try to test everything possible without the hardware and without manual intervention. Of course, there may be some testing of the fully integrated system, once we have one. We try to minimize the need for manual testing.

Concurrency

As this application grows, there undoubtedly are stories that lead us to need concurrent processing. Up to this point we have code running in a single thread proving the application logic developed. Now our customer plays the story that breaks our single threaded model, such as "Fifty users go off hook at the same time". We could go and start adding threads to our application classes. If we mix threads up with the application logic, it gets hard to test. Keep in mind that the threading structure of a design 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. Let's opt for a good design decision; keep threading logic separate from application logic. We are practicing separation of concerns.

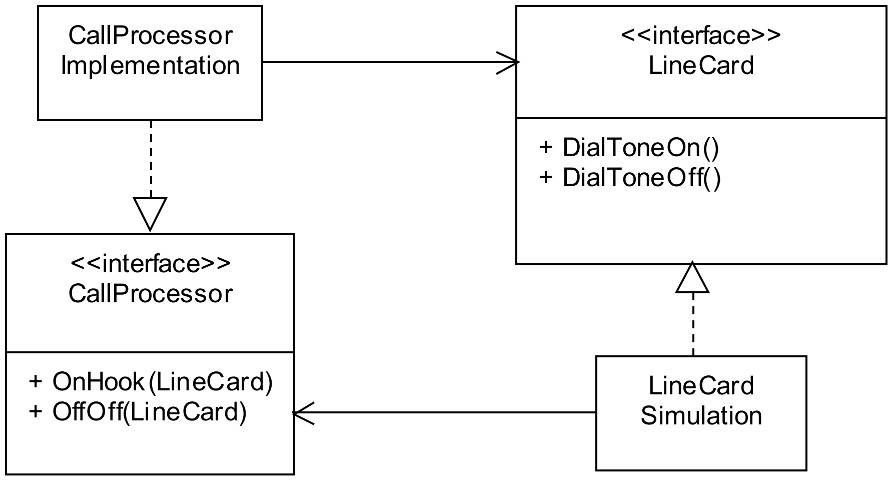
What is a simple way to handle it? For starters, let's say that each physical line card has its own LineCard instance, whose methods run in the LineCard's thread. Here we employ the idea of an ActiveObject[SCHMIDT] in our design. An ActiveObject is an object that runs in its own thread.



The ActiveLineCard is an ActiveObject. It has its own thread. Calls to it cause the specific line card request to be executed in the ActiveLineCard's thread. The ActiveLineCard delegates the calls to a RealLineCard or a FakeLineCard through the LineCard interface. The threading logic is contained in the ActiveLineCard. By keeping the tests for ActiveLineCard and RealLineCard separate, we simplify testing.

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 will probably come across some test, or new user story that forces 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, the design modifications can quickly destroy the design slowing our responsiveness to the changing business needs. 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 the team to understand. When the design starts to deteriorate, refactoring techniques are used to clean up the messy parts of the design. We can refactor with confidence because we have the safety net the automated tests provide. A design surrounded by tests is safer to modify because the tests detect when we make mistakes. These techniques give us confidence that we can evolve the design. Design decisions are made incrementally. The code is kept clean and modular; resulting in a system that is easier to evolve.

Many systems start out with a good clean design. New requirements pummel the system. When the existing design cannot cleanly support the new requirements, the design degrades. The good design is lost forever if the team does not refactor. 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 are not easy to write, but neither is it easy to test the real time performance in a live system.

Let’s say we determine that a specific interrupt service routine must complete within a specific time period. We write unit tests to check the execution time of specific code sections. Run this test on the target platform. We may have to ask our hardware designer for a timer we can use for these kinds of tests. If there is a meaningful way to run it on the development platform, go for it.

We can create acceptance tests that simulate loads and event sequences that may be difficult to generate on a live system. Our tests pump in a known traffic load such as N calls per second with M LineCards. We monitor the amount of CPU used during various loads, and count the number of lost calls. In addition, we 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]. Newcomer tells us that data is needed to locate the performance problems. He says you won’t be successful with 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. Would you like to fix a performance problem on a system with a clean design or a messy design? (this is not a trick question)

Development Tools

One potential obstacle to using XP for 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
- Lack of a unit test tool like JUnit or CppUnitLite for your development or execution environment

No OO programming language

Don’t be discouraged. XP values and many of the practices are still open to you. An OO programming language is not mandatory to do XP. OO facilitates module decoupling. Without OO it is more 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 is difficult to unit test without the advantages of run-time polymorphism. Link-time substitutability may be helpful for some tests. See a C/C++ Users Journal article describing test driven development in C.[KOSS]

Inability to run code on the build system

Don’t give up so easy! If the development system and the target system are different execution environments, figure out how to build for both environments. Can you work around the compiler differences? It’s worth the effort. Often target systems are scarce and extremely expensive. Running simulations and unit tests on the development system keeps the team moving forward. We have a much better chance of our code running on the target if we get it running on the development platform first.

If it is impossible to run the code on the development system, then all execution must be done on the target platform. This slows down development. XP relies on a fast change, compile, and test cycle. The cycle needs to run every few minutes or faster. A slow download delays this productive feedback loop.

If running on the development system is really not an option, port a unit test harness to your target or write your own. Unit test tools are simple. Compiler incompatibility could make it difficult to port a unit test tool. CppUnitLite is a handy test harness that is designed to be highly portable.

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. Start simple and evolve.

Safety Critical Systems

XP has a focus on sufficient quality for the application being developed. To paraphrase, the customer decides to build and pay for the quality that is needed. This is accomplished by defining the stories and the tests for the system. Safety critical systems differ from other applications in that they have safety critical requirements. Skilled individuals must work out the hazard scenarios that must be handled. As those requirements are understood, stories and tests are added to the system that assures the system behaves safely.

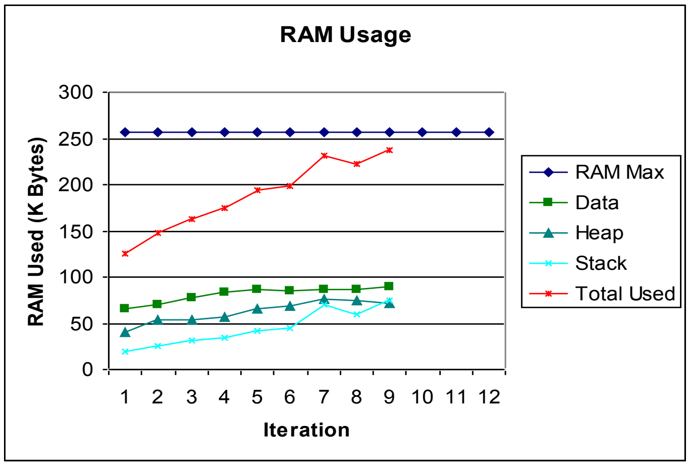
Your company or industry may have specific safety requirements. This may mean the creation of specific documents. When adding documentation to XP ask "What is the objective of the document?" Think "Can the objective be met in other ways?" Documents are not evil, but they are expensive and can easily get out of date. Try to keep the documents executable. Can the automated tests serve the purpose of the document? If the tests are not enough, see if the document can be created automatically. For example a traceability document might start at a user story. The story leads to a set of acceptance tests. Could the trace through the code for a specific acceptance test be generated from the running code?

I think Steve McConnell sums it up nicely: "Products whose reliability is important must be developed more carefully than products whose reliability doesn't much matter."[MCCONNEL] XP practices are designed to result in high quality code that is thoroughly tested. I'd want to have a very skilled team, well understood safety requirements and high test coverage to name a few attributes of a safety critical project.

Resource Constraints

Embedded software often has to run in constrained environments. The first embedded system I designed was a weather radar display system for the FAA. The display code lived happily in 12K of EEPROM. Yes, I mean K as in kilo-bytes. If you have a resource constraint, maybe you can write a test for it. The link map provides the starting and ending address of the code and data segments. Do the math. Create a script that calculates the amount of memory used, and run it as part of your build. Establish a budget. Set up the tests to make sure that you are not exceeding the budget.

In XP we use a thing called a Big Visible Chart or BVC. [BECK2] A BVC is used to track critical metrics. Memory usage could be a critical metric, especially considering that according to the Laws of Computer Programming "Any given program will expand to fill all available memory". Your build process could periodically update the memory usage data in a BVC.



The example RAM Usage BVC is regularly updated and kept visible to the team. The team can look for trends. In this chart it looks like the team was in fair shape until iteration seven. Memory usage took a big hit. The team reacted to the spike in usage by reduced memory usage in iteration eight. If the current trend continues, it looks like the team will run out of memory in iteration ten or eleven.

Conclusion

Extreme Programming is an Agile software development technique. [AGILE] XP and Agile are about having motivated teams of people iteratively building software. The values and principles behind XP and Agile can help build better software faster. Even if certain practices need modification to work in your environment, the values and principles provide guidance.

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. Another benefit is the potential for improved quality with the significant investment in testing that is made. Every software project I have seen can benefit from XP's improved process visibility and date predictability.

Tools may be problematic for some embedded systems development. Deploying embedded software may still have high costs, but XP techniques can help.

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

[BOEHM] Boehm, Barry W., A Spiral Model of Software Development and Enhancement, Computer, September 1987 pp43-57

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

[MACKINNON] Tim Mackinnon, Steve Freeman, Philip Craig, Endo-Testing: Unit Testing with Mock Objects (tim.mackinnon@pobox.com, steve@m3p.co.uk, philip@pobox.com)

[1] There is a good catalog of unit test tools at www.xprogramming.com.

[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

[KOSS] Koss, Dr. Robert, Langr, Jeff Test Driven Development in C/C++, C/C++ Users Journal, October 2002, http://www.cuj.com/articles/2002/0210/0210a/0210a.htm?topic=articles

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

[BECK2] Beck, Kent, Extreme Programming Explained, Addison Wesley, 1999, p.72-73

[AGILE] http://www.agilealliance.com