Reliable Codelines

Stephen P Berczuk - berczuk@acm.org
Skyva International, Inc.

This paper presents some patterns from the Effective Teams pattern language. The Effective teams language addressed how to use version control and related tools to develop software quickly. This paper presents 1 previously developed pattern and 3 new patterns incorporated into part of a language.

Copyright 2001, Stephen P. Berczuk. Permission is granted to make copies for purposes related to the 2001 Pattern Languages of Programs Conference. All other rights reserved.

The patterns in this paper describe structures that allow people to work together simultaneously to develop software.

This paper presents 4 Patterns:

- ‘Active Development Line,” on page7
- ‘Smoke Test,” on page13
- ‘Unit Test,” on page17
- ‘Regression Test,” on page21

Figure 1, “Overview of the Language,” on page 5 shows how these patterns fit into the larger context of the language. Patterns in bold outlines are patterns in this paper. An arrow from pattern one to pattern two means that pattern one sets the context of pattern two, or equivalently, that pattern two completes

pattern one. The relationship between patterns depends on the language. To use the patterns, you would start at the pattern that you want to realize, and then build the other ones

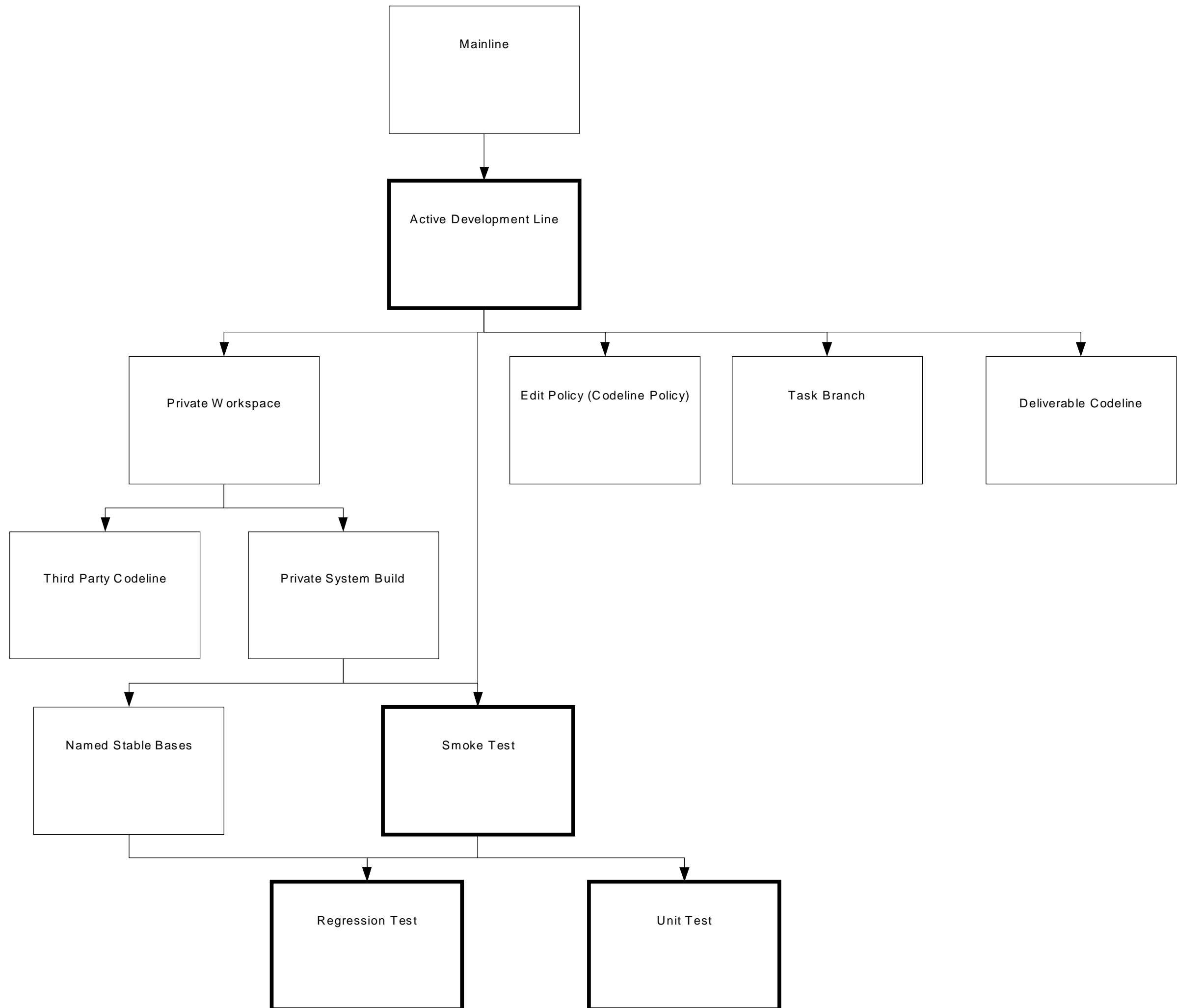

FIGURE 1 - *An Overview of the Language*

Active Development Line

Balancing Stability and Progress

You have an evolving codebase, and a codeline designated to work with a future product release. You are doing most of your changes on a *Mainline (3).* A dynamic development environment means that code is changed concurrently by everyone on the team. Team members are working towards making the system better, but any change can break the system, and any two concurrent changes can conflict. This pattern addresses balancing stability and progress in an active development effort.

How do you keep a rapidly evolving codeline stable enough to be useful?

Team software development is a balance of a number of conflicting forces. We develop in teams because we hope that more people working on a task will allow for concurrent work. For the team as a whole to make progress we need synchronization points. As in any concurrent system, having a synchronization point means that there is a possibility for deadlock or blocking if we don't manage the coordination correctly. If you think of software development as a set of concurrent processes, the codeline in the source control system is the synchronization point. At any point in time the tip of this active development line will have the latest versions of all of the system components.

Change is happening all around; if the product line is young, perhaps there is even dramatic change. You want to be able to grab current code from the source control system and have a reasonable expectation that it will work. Working from a highly tested stable line isn't always an option when you are developing a new feature, since your version control system is the way you want to exchange work in progress for integration. You don't want the process that ensures stability of work in progress code to slow you down too much. A broken codeline slows down everyone who works off of it, but the time it takes to test exhaustively slows down people as well, and in some cases can provide a false sense of security.

You want the codeline to be stable so that it does not interfere with people's work. An easy way to get stability is at the expense of progress.

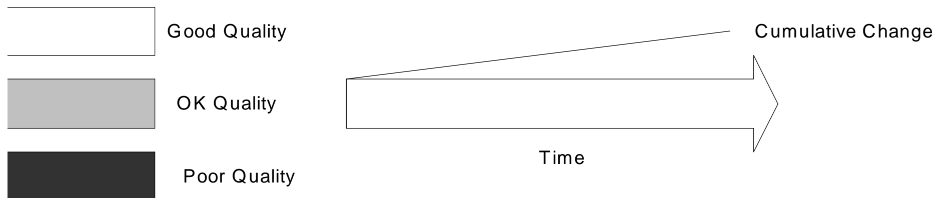

FIGURE 0.1 - ***A Codeline that is stable, but static***

You can require that people perform simple tests before submitting code to the codeline, such as a preliminary build, and some level of testing. These tests take time though, and may work against the some of your greater goals. Even if you do test your code before checkin, concurrency issues mean that two changes, tested individually, will result in the second one breaking the system. And the more exhaustive

-- and longer running -- your tests are, the more likely it is that there may be a non-compatible change submitted.

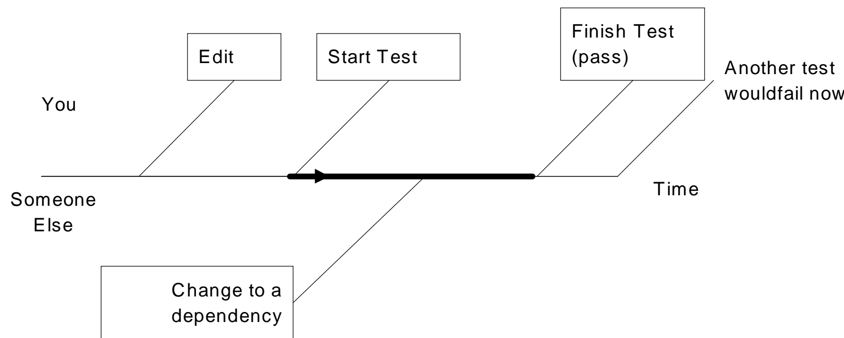

FIGURE 0.2 - ***If the test takes to long, you might not get meaningful results.***

You can also make changes to your codeline structure to keep parts of the code tree stable, creating an branches at various points, but that adds complexity, and requires a merge.

You can go to the other extreme, and make your codeline a free-for all.

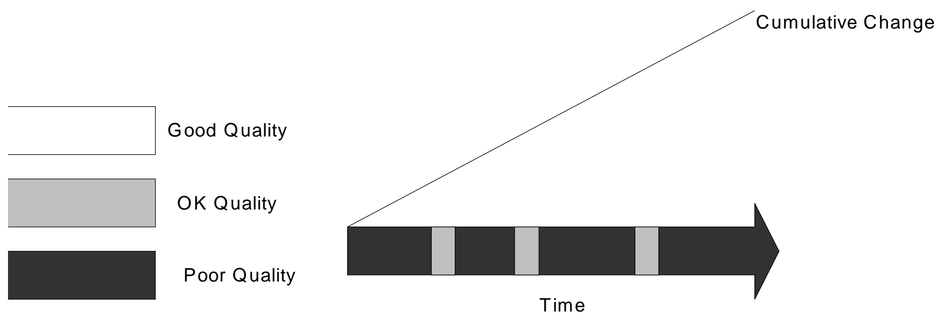

FIGURE 0.3 - ***A very active, but very useless codeline***

The module architecture of the system can also simplify the process if the module are defined so to reduce the likelihood of conflicting change, but, even then you may still have two people changing the code in a way that causes interactions.

You want a balance: an active code line that will more likely than not, be usable most of the time.

Aiming for perfection is likely to fail in all but the most static environments. Stability on a given line of code can be achieved, but with process and synchronization overhead, increased merging, and more complicated maintenance and administration. This is not always worthwhile.

A Story

I have worked at a number of startup companies, and there is a recurring theme that goes like this: Initially, there are only a few people working on the product. They understand what they are doing very well, and even when they step on each other's work, they recover quickly. Then the company grows, and the code in version control is hardly every consistent. The tip of the mainline always breaks. In frustration, someone sets up a test suite that people should run before doing a check in to the source control system. The first cut at this test suite is every test that they can think of. The test suite grows and soon it takes an hour to run the pre-checkin tests. People compensate by checking code in less often, causing pain when there are merges or other integration issues. Productivity goes down as well. Someone suggests shortening the test suites, but they are met with resistance justified by cries of "We are doing this to ensure quality." Someone else comments that "the pain is worth it, considering what we went through last year when we had no tests. But, once we reached a basic level of stability, the emphasis on exhaustive testing lead to diminishing returns as progres as a whole was reduced. This gets worse when the tests are not exhaustive, but simply exhausting to the developers who run them.

Define your goals

Institute policies that are effective in making your main development line stable enough for the work it needs to do. Do not aim for a perfect active development line, but rather for a mainline that is usable and active enough for your needs.

An active development line will have frequent changes, some well tested checkpoints that are guaranteed to be "good," and other points in the codeline are likely to be good enough for someone to do development on the tip of the line.

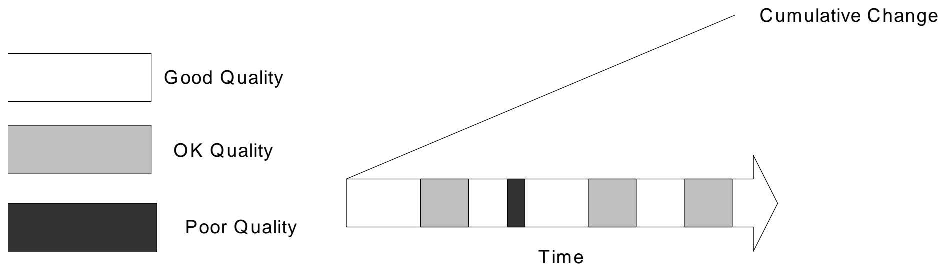

FIGURE 0.4 - ***An active, alive, codeline***

The hard part of this solution is establishing your needs. The process you need to go through is much like doing a requirements analysis for building software system. Your clients want perfection and completeness, and they want it quickly and cheaply. These goals are unattainable in a strict sense. Do an analysis along the following lines:

- Who uses the code line?
- What is the release cycle?
- What test mechanisms do we have in place?
- How much is the system evolving?
- What the real costs will be for a cycle where things are broken

For example, if the codeline is being used by other teams that are also doing active development, some instability is appropriate and the emphasis should be on speed. If this codeline is basically stable, and being used as a standard component, more validation is appropriate. If this is the beginning of the release cycle, more instability is expected. Right before you want to branch or freeze for a release, you want to test more. If you have good unit and regression tests, either run by developers or as part of the system build post checkin, errors will not persist as long, so emphasize speed on checkin. If you do not have this infrastructure, be more careful until you develop it. If you want to add functionality, emphasize speed.

If a client needs a good deal of stability, they should only used named stable bases of the components, so that they can avoid the work in progress. But these clients should then be treated more like external clients than members of the active development team.

Don't be too conservative. People can work with any system as long as they understand the trade-offs and the needs. You don't want to make the checking process too difficult. As you have a pre-checkin process that takes a long time you run the risk of developers doing larger grained, and less frequent

checkins in an attempt to make progress. Less frequent checkins increase the possibility of a conflict during testing.

Establish a criteria for how much to test the code before checkins: “The standard needs to set a quality level that is strict enough to keep showstopper defects out of the daily build, but lenient enough to disregard trivial defects (because undue attention to trivial defects can paralyze progress).” [1](Rapid Development p 407)

If you need a stable code line, perhaps what you want isn’t the active development line, but rather a fully QA’d release line. Remember that there is a fundamental difference between code that close to release and code that is being actively changed. There are significant benefits in the form of catching potential problems early in developing with an Active Development Line.You can also push off your more exhaustive testing to a batch process that creates your *Named Stable Bases (19).*

One factor in determining how you manage this process is your project’s rhythm. Kane and Dikel define rhythm as “the recurring, predictable exchange of work products within an architecture group and across customers and suppliers”[2] A good project rhythm is especially important for architecture based development, but any project that has concurrent work with dependencies needs a good rhythm. The source control structure can influence how the rhythm is executed, but culture plays an important role here.

To prevent total chaos on the main line setup each developer with a *Private Workspace (6)* where they can do a *Private System Build (7)*, *Unit Test (4)* and *Smoke Test (3)*.

Have an integration workspace where snapshots of the code are built periodically an subjected to more exhaustive tests.

Unresolved Issues

Once you have established that a ‘good enough’ codeline is desirable, you need to identify the codeline that will be like this. *Edit Policy (5)* will establish which lines follow this form, and what the checkin/commit process is for these (and other) codelines.

To support the developer side of keeping a codeline active, developers need to work in their own *Private Workspace (6).*

When the need for stability gets close, some work will need to be broken off to a *Deliverable Codeline (17).*

Some long lived tasks may need more stability that an active development line can provide, even though you realize that there may be an integration cost later. For these, use a *Task Branch (10)*. Doing this also insulates the primary code line from high risk changes.

Tool Support

Any SCM tool that supports "triggers" or automatic events that happen after a change is submitted will help automate the process of verifying that you are meeting the quality metric. You can then set up the system to run a build or a set of tests after a change is submitted. You can also set up the system to run less often.

Automated error detection and identification is also important.

References

A great book about getting to the core of the "real" problem is *Are Your Lights On?* [3].

Smoke Test

Simple Checks

An *IntegrationBuild (8)* or a *Private System Build (7)* are useful for verifying build-time integration issues.But even if the code builds, you still need to check for runtime issues that can cause you grief later. This verification is essential if you want to maintain a *Active Development Line (2)*. This pattern addresses the decisions you need to make to validate a build.

How do we know that the system will still work after you make a change?

You hope that you tested the code adequately before checking it in, but it is hard to develop thorough tests, and time consuming to run exhaustive tests. Unstructured and impromptu testing will help you o discover new problems, but it may not have much of an effective yield.

Running detailed tests is time consuming, but if you check in a change that breaks the system, you waste everyone's time.

Rapid development and small grained checkins means that you want the cost of pre-checkin verification to be small.

Story

Smoke tests are important on many levels. At one place I worked, releases were simply built, and the first developer to try them got the pleasure of finding (and sometimes fixing) all of the bugs. At another place, the pre-checkin test process was so exhaustive that developers feared it, matching checkins to do an few as possible (thus not isolating changes) and it had a negative effect on productivity. Also, it was very likely that someone would check in a conflicting change in the 60 minutes that the test ran. So, for both "good enough" pre-checkin validation, and minimal post built testing, a smoke test is essential.

Verify Basic Functionality

Subject each build to a smoke test that verifies that the application has not broken in an obvious way.

A smoke test should be good enough to catch "show stopper" defects, but not disregard trivial defects[1P 407] definition of "trivial" is up to the individual project, but you should realize that the goal of a smoke test is not the same as the goal of the overall quality assurance process.

The scope of the test need not be exhaustive. It should test basic functions, and simple integration issues. Ideally it should be automated so that there is little cost to do it. The *Smoke Test* should not replace deeper integration testing. A suite of unit like tests can form the basis for the smoke test if nothing else is immediately available. Most importantly, these tests should be self scoring. They should return a test status and not require manual intervention to see if the test passed. (An error may well involve some effort to discover the source.)

Developers should run the smoke test should be run manually prior to committing a change. It can also be run as part of the build process in concert with more through tests, when the build is to be a release candidate.

Running a Smoke test with each build does not remove the responsibility for a developer to test his changes before submitting them to the repository. A smoke test is most useful for bug fixes, and for looking for inadvertent interactions between existing and new functionality. All code should be unit tested by the developer, and where reasonable, run through some scenarios in a system environment.

When you add new basic functionality to a system, extend the smoke test to test this functionality as well. But do not put exhaustive tests that better belong in Unit Tests or Regression tests.

Daily Build and Smoke Test [4] describes the role of smoke test in maintaining quality. Having a Smoke Test as part of a Daily build is key to establishing Named Stable Bases, which form the basis for workspaces.

A smoke test should be:

- Quick to run, where 'quick' depends on your specific situation
- Self scoring, as any automated test should be.
- Provide broad coverage across the system that you care about
- Be runnable by developers

The hardest part about a self scoring test is to determine input/output relationships among elements of a complex system.You don't want the testing and scoring infrastructure to be buggy. You want the test to work with realistic data exchanged between parts of the system.

Canned inputs are fine as long as they are realistic enough. If your testing infrastructure is too complicated, you add risks around testing the test.

A Smoke test is an end to end test, more black box than white box.

Unresolved Issues

To get meaningful results from a Smoke Test you need to work off of a consistent build. A *Private System Build (7)* will let you build the system in a way that will give meaningful test results.

References

Rapid Development[1] and Mythical Man Month[5] have some good advice on various testing strategies, including the tradeoffs between completeness and speed.

Unresolved Issues

The trade-off we need to make here involve speed of checkin (The longer the pre-checkin test, the longer the checkin) Longer checkins my encourage developers to have larger granularity commits. This goes against an important goal of using version control.

We still need to ensure a higher level of quality. A pre release QA process can provide some of this, but we can also use a *Regression Test (5)* to do more exhaustive testing to identify changes. Use a *Unit Test (4)* to verify that the module you are changing still works adequately before you check the change in,

If the quality goals are such that you need to do exhaustive testing, consider using Task Branches, or have a different codeline policy. Also consider branching release lines.

Unit Test

Things Change

Sometimes a *Smoke Test (3)* is not enough to test a change in detail when you are working on a module. This pattern shows you how to test detailed changes so that you can ensure the quality of your codeline.

How do you test whether a module or class still works as it should after making a change?

Checking that an element (a class, module or function) still works after you make a change is a basic procedure that will help you maintain stability in your software development. Testing small scale units can seen tedious.

Integration is where most of the problems become visible, but when you have the results of a failed integration test, you are still left with the question: "What broke?" Also, testing integration level functions can take longer to set up, they require many pieces of the system to be stable. You want to be able to see if any incremental change to your code broke something, so being able to run the tests as often as you like had benefits.You also want to run comprehensive tests on the item that you are changing before checkin.

Since a Smoke Test, is by its nature somewhat superficial, you want to be able to ensure that each part of a system works reasonably well.

When a system test, such as a smoke test, fails you want to figure out what part of the system broke. You want to be able to run quick tests in development to see the effect of a change. Additional testing layers add time. Tests that are too complex take more effort to debug than the value that they add.

We want to isolate integration issues from local changes and we want to test the contracts that each element provides locally.

A Story

I'd worked at a number of places where testing was a bit ad-hoc. We did system tests, but never really focused on unit tests. When system tests failed, we'd run code through the debugger, and sometimes we found a problem. Other times we found that the problem was that a client violated an interface contract. It took more effort that we really needed to spend. After the XP book came out, and having been inspired by chatting with Kent Beck and Martin Fowler at OOPSLA, I took unit testing a bit more seriously. The next project my colleague and I wrote unit tests using the CPP unit framework. It took some effort to convince them of the value, but when we started to isolate problems quickly (often to parts of the code that did not have unit tests!), my colleague became convinced. Not only that, but the unit tests made making code changes less scary.

Test The Contract

Develop and Run Unit Tests.

A unit test is a test that tests fine grained elements of a component to see that they obey their contract. A good unit test has the following properties[6]:

- Automatic and Self Evaluating. A unit test can report a boolean result automatically. A user should not have to look at the detailed test results unless there is an error.
- Fine grained. Any significant interface method on a class should be testing using know inputs. It is not necessary to write tests to verify trivial methods like accessors and setters.To put it simply, the test tests things that might break.
- Isolated. A unit test does not interact with other tests. Otherwise one test failing may cause others to fail.
- It should test the contract.The test should be self contained so that external changes do not effect the results. Of course, if an external interface changes, you should update the test to reflect this
- Simple to run. You should be able to run a unit test by a simple command line or graphical tool. There should not be any setup involved.

You should run unit tests while you are coding, just before checking in a change and after updating your code to the current state, and you can also run all of your unit tests when you are trying to find a problem with a smoke test, regression test, or in response to a user problem report.

Try to use a testing framework like JUnit (or cppUnit, PyUnit, and other derived frameworks). This will allow you to focus on the Unit Tests, and not distract yourself with testing infrastructure.

Unit testing is indispensible when making changes to the structure of the code that should not effect behavior, such as when you are refactoring.[7]

Unresolved Issues

There are many to be documented (and perhaps already documented) patterns and best practices about how much to cover in unit tests, and the mechanics of writing and running the tests. (I would welcome any pointers to existing sources.)

References

Unit Testing is a key part of *Extreme Programming* [6, 8]

Regression Test

Regression Happens

If you want to release Named Stable Bases for developers to use, or to establish release candidates, you need to be sure that the product base is robust. This pattern explains how to generate Named Stable Bases(Builds) that work as least as well as they did before a change.

How do you ensure that existing code doesn't doesn't get worse as you make other improvements?

Software systems are complex; changes to a system come with the possibility of breaking something seemingly unrelated to your changes. Without change, you can't make progress, but the impact of a change in hard to measure, especially in terms of how the a unit of code interacts with the rest of the system. Fixing a defect has a substantial chance of introducing another[5].

Exhaustive testing takes time, but if you don't do this testing you waste developer, and perhaps, customer time. Code changes can cause errors in parts of the system that you were not working on. (Even with a good architecture) as well as improve things.

You hope that others have done a good job of checking for negative consequences before they made changes. Even if you and your colleagues make a good faith effort to test, you may still not have tested against all of the changes made by others. There is no easy way to test exhaustively. Software systems are complex, and changes can easily have unexpected consequences.

Some integration tests may need resources that are not on every development machine.

When the system does break, you want to identify some point in time when something broke.

A Story

I worked for a small software product company that had a codebase combined of newer, cleaner code, and also code that evolved. On any given day, it was not clear whether you could get an update from source control and have a working system, or whether you could would have to spend the day getting the system to a point where you could do your work. The problem was that there was no automated testing of the core APIs. People would avoid moving to a current code base in fear of wasting a day, but this eventually caused other problems.

Test for Changess

Run some Regression tests on systems after major changes before commiting the changes. Run more exhaustive regression tests before releasing to Named Stable Bases.

Regression tests are end to end (Black box) tests that cover anticipated failure modes. If a regression test fails, debugging and unit tests may be necessary to determine what low-level component or interface broke.

One approach to building regression tests is to add a test for every error that people find in the QA process, or even at customer deployments.

Regression Tests test changes in Integration behavior. They are large grained, and test for unexpected consequences of integrating software components. Unit tests can be thought through fairly easily. As you add component interactions it is harder to write tests based on 'first principles.'

Run regression tests whenever you have a candidate for named stable bases. Regression testings can involve running all the unit tests, but it is better if the tests involve system input. If something breaks, you can always run the unit tests to localize the change.You also have to investigate if the unit test inputs no longer match the system.

Booch suggests that during evolution you carry out unit testing of all new classes and objects, but also apply gression testing to each new complete release. Institute a policy of automated regression testing tied to each release [9].

Regression Testing is designed to make sure that the software has not taken a step backwards (or regressed) Always run the same tests for each regression cycle. Add tests as you find more conditions or problematic items to test.

Booch (Object Solutions) suggests that during evolution you carry out unit testing of all new classes and objects, but also apply gression testing to each new complete release. Institute a policy of automated regression testing tied to each release.(p 238)

Write regression tests by starting out with system level tests based on requirements. As you discover problems, write a test that reproduces the problem and add that scenario to the test. Over time you will end up with a large suite of tests that cover your most likely problem areas. Since the test may be long running, you may not want to run it for every code change. There are advantages, however to having an automated procedure to run the regression test after each change, so that you can identify the poin at which the system regressed.You will want to run them for each release candidate before subjecting the release candidate to other QA procedures.

Unresolved Issues

(We need references to patterns about the mechanics of building and running regression tests.)

References

Steve McConnell has a lot of information about testing of all kinds in Code Complete[10]

The Art of Software Testing [11]by Glen Meyers is a classic.

Summary

Stable Enough Codelines

We often speak of quailty and speed, but we don't often speak of how to balance the two. The stereotype is that the testing or QA group cares about quality, and the development group (and their management) speed. This is not generally the case. By balancing testing with the need for speed, and using codelines appropriately, we can balance the speed and quality. The table below is one example of how each of the testing patterns fit in various parts of the product codeline lifecycle.

Test	Development	Checkin	Build	Release
Unit Test	X			
Smoke Test		X	X	
Regression Test		X?	X	X

The important thing to remember is to understand your goals and implement your testing and codeline policies to meet those goals. Blindly following strategies because they are "good" will only see profitable in the short term.

Acknowledgements

Thanks to Brandon Goldfedder, my Shepherd for PLoP 2001. My experiences at various growing companies (some of which are no longer with us) greatly influenced what I wrote here. And I must express a lot of appreciation for those who taught me how to do things right early in my professional career.

References

1.McConnell, S., *Rapid Development, Taming Wild Software Schedules.* 1996, Redmond, WA: Microsoft Press.

2.Dikel, D.M., D. Kane, and J.R. Wilson, *Software architecture : organizational principles and patterns*. 2001, Upper Saddle River, NJ: Prentice Hall. cm.

3.Gause, D.C. and G.M. Weinberg, *Are Your Lights On? How to Figure out what the Problem REALLY Is*. 1990, New York, NY: Dorset House.

4.Coplien, J.O., *A Generative Development Process Pattern Language*, in *Pattern Languages of Program Design*. 1995, Addison-Wesley: Reading, MA.

5.Brooks, F.P., *The Mythical Man-Month : Essays on Software Engineering*. 20th Anniversary ed. 1995, Reading, Mass.: Addison-Wesley Pub. Co. xiii, 322.

6.Beck, K., *eXtreme programming eXplained : embrace change*. 2000, Reading, MA: Addison-Wesley. . cm.

7.Fowler, M., *Refactoring: Improving the Design of Existing Code*. Object Technology Series, ed. J.C. Shanklin. 1999, Reading, MA: Addison-Wesley.

8.Jeffries, R., A. Anderson, and C. Hendrickson, *Extreme programming installed*. 2000, Boston, MA: Addison-Wesley. . cm.

9.Booch, G., *Object solutions : managing the object-oriented project*. 1996, Menlo Park, Ca.: Addison-Wesley Pub. Co. xii, 323.

10.McConnell, S., *Code complete : a practical handbook of software construction*. 1993, Redmond, Wash.: Microsoft Press. xviii, 857.

11.Myers, G.J., *The art of software testing*. 1979, New York: Wiley. xi, 177.