A. Tooling

In 1978, I was working at Teradyne on the telephone test system that I described earlier. The system was about 80KSLOC of M365 assembler. We kept the source code on tapes.

The tapes were similar to those 8-track stereo tape cartridges that were so popular back in the ’70s. The tape was an endless loop, and the tape drive could only move in one direction. The cartridges came in 10′, 25′, 50′, and 100′ lengths. The longer the tape, the longer it took to “rewind” since the tape drive had to simply move it forward until it found the “load point.” A 100′ tape took five minutes to go to load point, so we chose the lengths of our tapes judiciously.[1]

Logically, the tapes were subdivided into files. You could have as many files on a tape as would fit. To find a file you loaded the tape and then skipped forward one file at a time until you found the one you wanted. We kept a listing of the source code directory on the wall so that we would know how many files to skip before we got to the one we wanted.

There was a master 100′ copy of the source code tape on a shelf in the lab. It was labeled MASTER. When we wanted to edit a file we loaded the MASTER source tape into one drive and a 10′ blank into another. We’d skip through the MASTER until we got to the file we needed. Then we’d copy that file onto the scratch tape. Then we’d “rewind” both tapes and put the MASTER back on the shelf.

There was a special directory listing of the MASTER on a bulletin board in the lab. Once we had made the copies of the files we needed to edit, we'd put a colored pin on the board next to the name of that file. That's how we checked files out!

We edited the tapes on a screen. Our text editor, ED-402, was actually very good. It was very similar to vi. We would read a "page" from tape, edit the contents, and then write that page out and read the next one. A page was typically 50 lines of code. You could not look ahead on the tape to see the pages that were coming, and you could not look back on the tape to see the pages you had edited. So we used listings.

Indeed, we would mark up our listings with all the changes we wanted to make, and *then* we'd edit the files according to our markups. *Nobody* wrote or modified code at the terminal! That was suicide.

Once the changes were made to all the files we needed to edit, we'd merge those files with the master to create a working tape. This is the tape we'd use to run our compiles and tests.

Once we were done testing and were sure our changes worked, we'd look at the board. If there were no new pins on the board we'd simply relabel our working tape as MASTER and pull our pins off the board. If there *were* new pins on the board we'd remove our pins and hand our working tape to the person whose pins were still on the board. They'd have to do the merge.

There were three of us, and each of us had our own color of pin, so it was easy for us to know who had which files checked out. And since we all worked in the same lab and talked to each other all the time, we held the status of the board in our heads. So usually the board was redundant, and we often didn't use it.

Tools

Today software developers have a wide array of tools to choose from. Most aren't worth getting involved with, but there are a few that every software developer must be conversant with. This chapter describes my current personal toolkit. I have not done a complete survey of all the other tools out there, so this should not be considered a comprehensive review. This is just what I use.

Source Code Control

When it comes to source code control, the open source tools are usually your best option. Why? Because they are written by developers, for developers. The open source tools are what developers write for themselves when they need something that works.

There are quite a few expensive, commercial, "enterprise" version control systems available. I find that these are not sold to developers so much as they are sold to managers, executives, and "tool groups." Their list of features is impressive and compelling. Unfortunately, they often don't have the features that developers actually need. The chief among those is *speed*.

An "Enterprise" Source Control System

It may be that your company has invested a small fortune in an "enterprise" source code control system. If so, my condolences. It's probably politically inappropriate for you to go around telling everyone, "Uncle Bob says not to use it." However, there is an easy solution.

You can check your source code into the "enterprise" system at the end of each iteration (once every two weeks or so) and use one of the open source systems in the midst of each iteration. This keeps everyone happy, doesn't violate any corporate rules, and keeps your productivity high.

Pessimistic versus Optimistic Locking

Pessimistic locking seemed like a good idea in the '80s. After all, the simplest way to manage concurrent update problems is to serialize them. So if *I'm* editing a file, *you'd* better not. Indeed, the system of colored pins that I used in the late '70s was a form of pessimistic locking. If there was a pin in a file, you didn't edit that file.

Of course, pessimistic locking has its problems. If I lock a file and then go on vacation, everybody else who wants to edit that file is stuck. Indeed, even if I keep the file locked for a day or two, I can delay others who need to make changes.

Our tools have gotten much better at merging source files that have been edited concurrently. It's actually quite amazing when you think about it. The tools look at the two different files and at the ancestor of those two

files, and then they apply multiple strategies to figure out how to integrate the concurrent changes. And they do a pretty good job.

So the era of pessimistic locking is over. We do not need to lock files when we check them out anymore. Indeed, we don't bother to check out individual files at all. We just check out the whole system and edit any files we need to.

When we are ready to check in our changes, we perform an "update" operation. This tells us whether anybody else checked in code ahead of us, automatically merges most of the changes, finds conflicts, and helps us do the remaining merges. Then we commit the merged code.

I'll have a lot to say about the role that automated tests and continuous integration play with regard to this process later on in this chapter. For the moment let's just say that we *never* check in code that doesn't pass all the tests. *Never ever*.

CVS/SVN

The old standby source control system is CVS. It was good for its day but has grown a bit long in the tooth for today's projects. Although it is very good at dealing with individual files and directories, it's not very good at renaming files or deleting directories. And the attic . . . . Well, the less said about that, the better.

Subversion, on the other hand, works very nicely. It allows you to check out the whole system in a single operation. You can easily update, merge, and commit. As long as you don't get into branching, SVN systems are pretty simple to manage.

Branching

Until 2008 I avoided all but the simplest forms of branching. If a developer created a branch, that branch had to be brought back into the main line before the end of the iteration. Indeed, I was so austere about branching that it was very rarely done in the projects I was involved with.

If you are using SVN, then I still think that's a good policy. However, there are some new tools that change the game completely. They are the *distributed* source control systems. `git` is my favorite of the distributed source control systems. Let me tell you about it.

`git`

I started using `git` in late 2008, and it has since changed everything about the way I use source code control. Understanding why this tool is such a game changer is beyond the scope of this book. But comparing Figure A-1 to Figure A-2 ought to be worth quite a few of the words that I'm not going to include here.

Figure A-1. FITNESSE under subversion

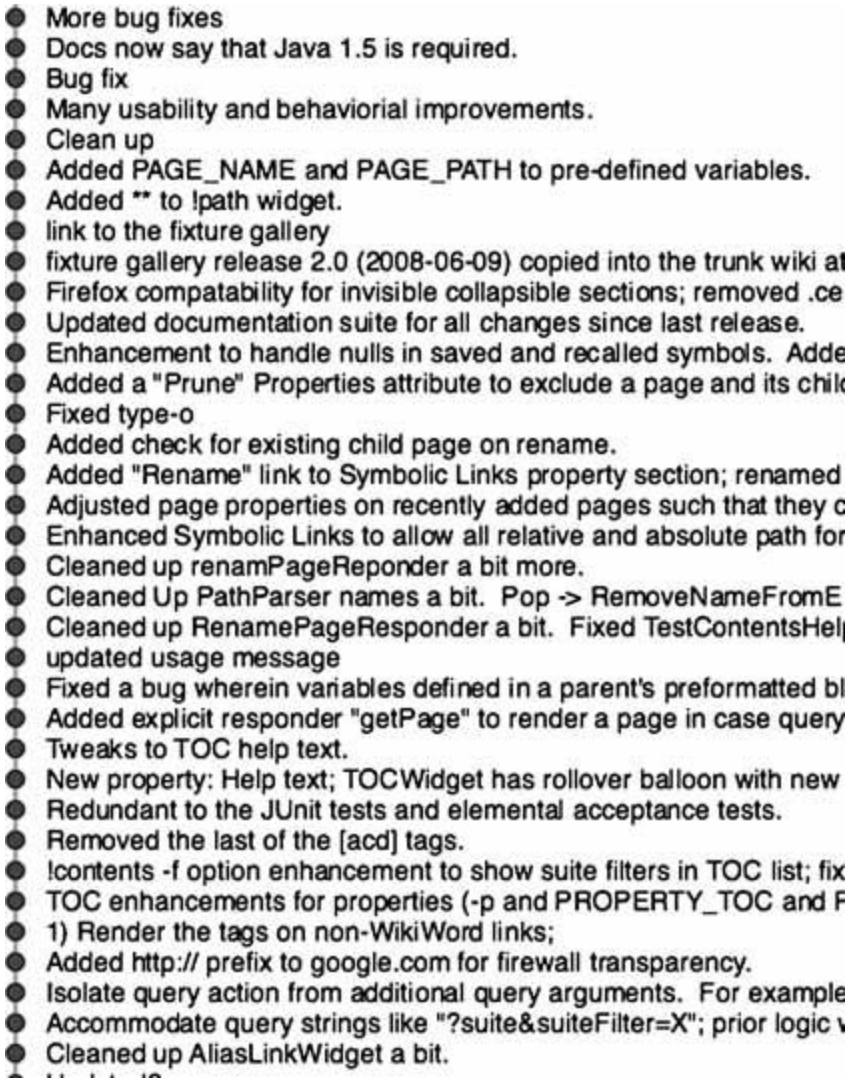

Figure A-2. FITNESSE under `git`

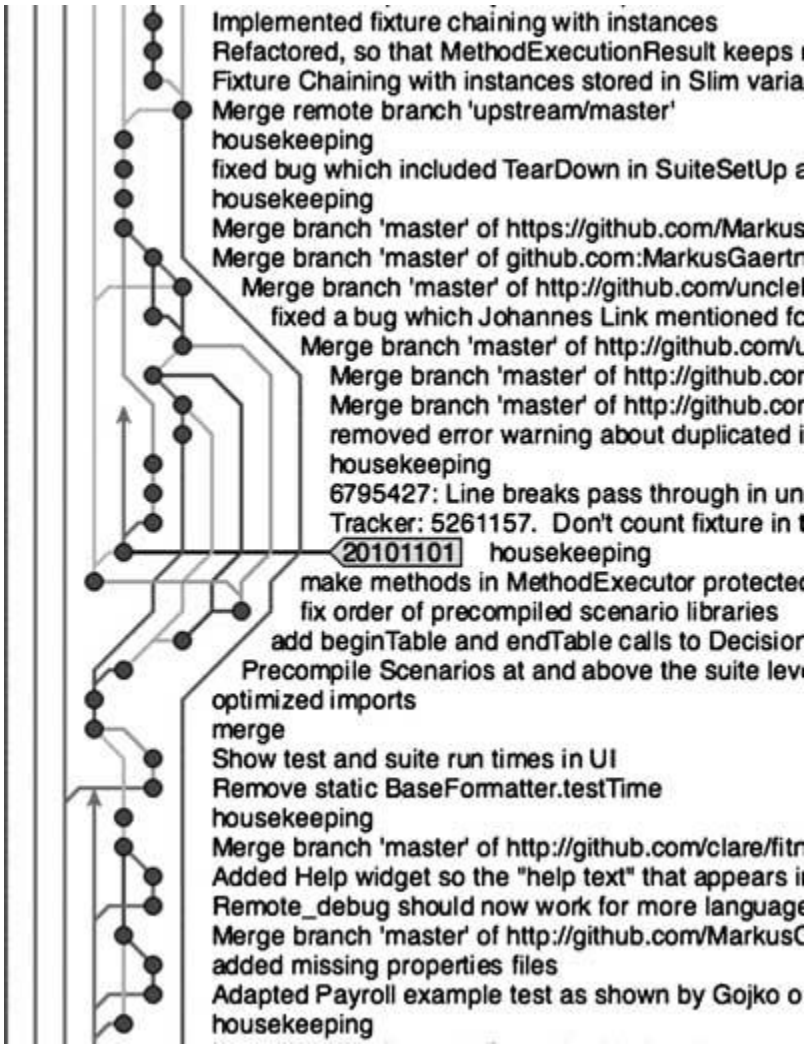

Figure A-1 shows a few weeks' worth of development on the FITNESSE project while it was controlled by SVN. You can see the effect of my austere no-branching rule. We simply did not branch. Instead, we did very frequent updates, merges, and commits to the main line.

Figure A-2 picture shows a few weeks' worth of development on the same project using `git`. As you can see, we are branching and merging all over the place. This was not because I relaxed my no-branching policy; rather, it simply became the obvious and most convenient way to work. Individual developers can make very short-lived branches and then merge them with each other on a whim.

Notice also that you can't see a true main line. That's because *there isn't one*. When you use `git` there's no such thing as a central repository, or a main line. Every developer keeps his or her own copy of the *entire* history

of the project on their local machine. They check in and out of that local copy, and then merge it with others as needed.

It's true that I keep a special golden repository into which I push all the releases and interim builds. But to call this repository the main line would be missing the point. It's really just a convenient snapshot of the whole history that every developer maintains locally.

If you don't understand this, that's OK. `git` is something of a mind bender at first. You have to get used to how it works. But I'll tell you this: `git`, and tools like it, are what the future of source code control looks like.

IDE/Editor

As developers, we spend most of our time reading and editing code. The tools we use for this purpose have changed greatly over the decades. Some are immensely powerful, and some are little changed since the '70s.

vi

You'd think that the days of using vi as the primary development editor would be long over. There are tools nowadays that far outclass vi, and other simple text editors like it. But the truth is that vi has enjoyed a significant resurgence in popularity due to its simplicity, ease of use, speed, and flexibility. Vi might not be as powerful as Emacs, or eclipse, but it's still a fast and powerful editor.

Having said that, I'm not a power vi user any more. There was a day when I was known as a vi "god," but those days are long gone. I use vi from time to time if I need to do a quick edit of a text file. I have even used it recently to make a quick change to a Java source file in a remote environment. But the amount of true coding I have done in vi in the last 10 years is vanishingly small.

Emacs

Emacs is still one of the most powerful editors out there, and will probably remain so for decades to come. The internal lisp model guarantees that. As a general-purpose editing tool, nothing else even comes close. On the other hand, I think that Emacs cannot really compete with the specific-purpose IDEs that now dominate. Editing code is *not* a general-purpose editing job.

In the ’90s I was an Emacs bigot. I wouldn’t consider using anything else. The point-and-click editors of the day were laughable toys that no developer could take seriously. But in the early ’00s I was introduced to IntelliJ, my current IDE of choice, and I’ve never looked back.

Eclipse/IntelliJ

I’m an IntelliJ user. I love it. I use it to write Java, Ruby, Clojure, Scala, Javascript, and many others. This tool was written by programmers who understand what programmers need when writing code. Over the years, they have seldom disappointed me and almost always pleased me.

Eclipse is similar in power and scope to IntelliJ. The two are simply leaps and bounds above Emacs when it comes to editing Java. There are other IDEs in this category, but I won’t mention them here because I have no direct experience with them.

The features that set these IDEs above tools like Emacs are the extremely powerful ways in which they help you manipulate code. In IntelliJ, for example, you can extract a superclass from a class with a single command. You can rename variables, extract methods, and convert inheritance into composition, among many other great features.

With these tools, code editing is no longer about lines and characters as much as it is about complex manipulations. Rather than thinking about the next few characters and lines you need to type, you think about the next few transformations you need to make. In short, the programming model is remarkably different and highly productive.

Of course, this power comes at a cost. The learning curve is high, and project set-up time is not insignificant. These tools are *not* lightweight. They take a lot of computing resources to run.

TextMate

TextMate is powerful and lightweight. It can’t do the wonderful manipulations that IntelliJ and Eclipse can do. It doesn’t have the powerful lisp engine and library of Emacs. It doesn’t have the speed and fluidity of vi. On the other hand, the learning curve is small, and its operation is intuitive.

I use TextMate from time to time, especially for the occasional C++. I would use Emacs for a large C++ project, but I’m too rusty to bother with

Emacs for the short little C++ tasks I have.

Issue Tracking

At the moment I'm using Pivotal Tracker. It's an elegant and simple system to use. It fits nicely with the Agile/iterative approach. It allows all the stakeholders and developers to communicate quickly. I'm very pleased with it.

For very small projects, I've sometimes used Lighthouse. It's very quick and easy to set up and use. But it doesn't come close to the power of Tracker.

I've also simply used a wiki. Wikis are fine for internal projects. They allow you to set up any scheme you like. You aren't forced into a certain process or a rigid structure. They are very easy to understand and use.

Sometimes the best issue-tracking system of all is a set of cards and a bulletin board. The bulletin board is divided into columns such as "To Do," "In Progress," and "Done." The developers simply move the cards from one column to the next when appropriate. Indeed, this may be the most common issue-tracking system used by agile teams today.

The recommendation I make to clients is to start with a manual system like the bulletin board before you purchase a tracking tool. Once you've mastered the manual system, you will have the knowledge you need to select the appropriate tool. And indeed, the appropriate choice may simply be to continue using the manual system.

Bug Counts

Teams of developers certainly need a list of issues to work on. Those issues include new tasks and features as well as bugs. For any reasonably sized team (5 to 12 developers) the size of that list should be in the dozens to hundreds. *Not thousands*.

If you have thousands of bugs, something is wrong. If you have thousands of features and/or tasks, something is wrong. In general, the list of issues should be relatively small, and therefore manageable with a lightweight tool like a wiki, Lighthouse, or Tracker.

There are some commercial tools out there that seem to be pretty good. I've seen clients use them but haven't had the opportunity to work with

them directly. I am not opposed to tools like this, as long as the number of issues remains small and manageable. When issue-tracking tools are forced to track thousands of issues, then the word "tracking" loses meaning. They become "issue dumps" (and often smell like a dump too).

Continuous Build

Lately I've been using Jenkins as my Continuous Build engine. It's lightweight, simple, and has almost no learning curve. You download it, run it, do some quick and simple configurations, and you are up and running. Very nice.

My philosophy about continuous build is simple: Hook it up to your source code control system. Whenever anybody checks in code, it should automatically build and then report status to the team.

The team must simply keep the build working at all times. If the build fails, it should be a "stop the presses" event and the team should meet to quickly resolve the issue. Under no circumstances should the failure be allowed to persist for a day or more.

For the FITNESSE project I have every developer run the continuous-build script before they commit. The build takes less than 5 minutes, so this is not onerous. If there are problems, the developers resolve them before the commit. So the automatic build seldom has any problems. The most common source of automatic build failures turns out to be environment-related issues since my automatic build environment is quite different from the developer's development environments.

Unit Testing Tools

Each language has it's own particular unit testing tool. My favorites are JUNIT for Java, RSPEC for Ruby, NUNIT for .Net, Midje for Clojure, and CPPUTEST for C and C++.

Whatever unit testing tool you choose, there are a few basic features they all should support.

1. It should be quick and easy to run the tests. Whether this is done through IDE plugins or simple command line tools is irrelevant, as

long as developers can run those tests on a whim. The gesture to run the tests should be trivial.

For example, I run my CPPUTEST tests by typing `command-M` in TextMate. I have this command set up to run my `makefile` which automatically runs the tests and prints a one-line report if all tests pass. JUNIT and RSPEC are both supported by INTELLIJ, so all I have to do is push a button. For NUNIT, I use the RESHARPER plugin to give me the test button.

2. The tool should give you a clear visual pass/fail indication. It doesn't matter if this is a graphical green bar or a console message that says "All Tests Pass." The point is that you must be able to tell that all tests passed quickly and unambiguously. If you have to read a multiline report, or worse, compare the output of two files to tell whether the tests passed, then you have failed this point.
3. The tool should give you a clear visual indication of progress. It doesn't matter whether this is a graphical meter or a string of dots as long as you can tell that progress is still being made and that the tests have not stalled or aborted.
4. The tool should discourage individual test cases from communicating with each other. JUNIT does this by creating a new instance of the test class for each test method, thereby preventing the tests from using instance variables to communicate with each other. Other tools will run the test methods in random order so that you can't depend on one test preceding another. Whatever the mechanism, the tool should help you keep your tests independent from each other. Dependent tests are a deep trap that you don't want to fall into.
5. The tool should make it very easy to write tests. JUNIT does this by supplying a convenient API for making assertions. It also uses reflection and Java attributes to distinguish test functions from normal functions. This allows a good IDE to automatically identify all your tests, eliminating the hassle of wiring up suites and creating error-prone lists of tests.

Component Testing Tools

These tools are for testing components at the API level. Their role is to make sure that the behavior of a component is specified in a language that

the business and QA people can understand. Indeed, the ideal case is when business analysts and QA can *write* that specification using the tool.

The Definition of *Done*

More than any other tool, component testing tools are the means by which we specify what *done* means. When business analysts and QA collaborate to create a specification that defines the behavior of a component, and when that specification can be executed as a suite of tests that pass or fail, then *done* takes on a very unambiguous meaning: "All Tests Pass."

FitNesse

My favorite component testing tool is FITNESSE. I wrote a large part of it, and I am the primary committer. So it's my baby.

FITNESSE is a wiki-based system that allows business analysts and QA specialists to write tests in a very simple tabular format. These tables are similar to Parnas tables both in form and intent. The tests can be quickly assembled into suites, and the suites can be run at a whim.

FITNESSE is written in Java but can test systems in any language because it communicates with an underlying test system that can be written in any language. Supported languages include Java, C#/.NET, C, C++, Python, Ruby, PHP, Delphi, and others.

There are two test systems that underlie FITNESSE: Fit and Slim. Fit was written by Ward Cunningham and was the original inspiration for FITNESSE and it's ilk. Slim is a much simpler and more portable test system that is favored by FITNESSE users today.

Other Tools

I know of several other tools that could classify as component testing tools.

- RobotFX is a tool developed by Nokia engineers. It uses a similar tabular format to FITNESSE, but is not wiki based. The tool simply runs on flat files prepared with Excel or similar. The tool is written in Python but can test systems in any language using appropriate bridges.

• Green Pepper is a commercial tool that has a number of similarities with FITNESSE. It is based on the popular confluence wiki.

• Cucumber is a plain text tool driven by a Ruby engine, but capable of testing many different platforms. The language of Cucumber is the popular Given/When/Then style.

• JBehave is similar to Cucumber and is the logical parent of Cucumber. It is written in Java.

Integration Testing Tools

Component testing tools can also be used for many integration tests, but are less than appropriate for tests that are driven through the UI.

In general, we don't want to drive very many tests through the UI because UIs are notoriously volatile. That volatility makes tests that go through the UI very fragile.

Having said that, there are some tests that *must* go through the UI—most importantly, tests *of* the UI. Also, a few end-to-end tests should go through the whole assembled system, including the UI.

The tools that I like best for UI testing are Selenium and Watir.

UML/MDA

In the early '90s I was very hopeful that the CASE tool industry would cause a radical change in the way software developers worked. As I looked ahead from those heady days, I thought that by now everyone would be coding in diagrams at a higher level of abstraction and that textual code would be a thing of the past.

Boy was I wrong. Not only hasn't this dream been fulfilled, but every attempt to move in that direction has met with abject failure. Not that there aren't tools and systems out there that demonstrate the potential; it's just that those tools simply don't truly realize the dream, and hardly anybody seems to want to use them.

The dream was that software developers could leave behind the details of textual code and author systems in a higher-level language of diagrams. Indeed, so the dream goes, we might not need programmers at all.

Architects could create whole systems from UML diagrams. Engines, vast and cool and unsympathetic to the plight of mere programmers, would transform those diagrams into executable code. Such was the grand dream of Model Driven Architecture (MDA).

Unfortunately, this grand dream has one tiny little flaw. MDA assumes that the problem is code. But code is *not* the problem. It has never been the problem. The problem is *detail*.

The Details

Programmers are detail managers. That's what we do. We specify the behavior of systems in the minutest detail. We happen to use textual languages for this (code) because textual languages are remarkably convenient (consider English, for example).

What kinds of details do we manage?

Do you know the difference between the two characters \n and \r? The first, \n, is a line feed. The second, \r, is a carriage return. What's a carriage?

In the '60s and early '70s one of the more common output devices for computers was a teletype. The model ASR33[2] was the most common.

This device consisted of a print head that could print ten characters per second. The print head was composed of a little cylinder with the characters embossed upon it. The cylinder would rotate and elevate so that the correct character was facing the paper, and then a little hammer would smack the cylinder against the paper. There was an ink ribbon between the cylinder and the paper, and the ink transferred to the paper in the shape of the character.

The print head rode on a carriage. With every character the carriage would move one space to the right, taking the print head with it. When the carriage got to the end of the 72-character line, you had to explicitly return the carriage by sending the carriage return characters (\r = 0 × 0D), otherwise the print head would continue to print characters in the 72nd column, turning it into a nasty black rectangle.

Of course, that wasn't sufficient. Returning the carriage did not raise the paper to the next line. If you returned the carriage and did not send a line-

feed character (\n = 0 × 0A), then the new line would print on top of the old line.

Therefore, for an ASR33 teletype the end-of-line sequence was "\r\n". Actually, you had to be careful about that since the carriage might take more than 100ms to return. If you sent "\n\r" then the next character just might get printed as the carriage returned, thereby creating a smudged character in the middle of the line. To be safe, we often padded the end-of-line sequence with one or two rubout[3] characters (0 × FF).

In the '70s, as teletypes began to fade from use, operating systems like UNIX shortened the end-of-line sequence to simply '\n'. However, other operating systems, like DOS, continued to use the '\r\n' convention.

When was the last time you had to deal with text files that use the "wrong" convention? I face this problem at least once a year. Two identical source files don't compare, and don't generate identical checksums, because they use different line ends. Text editors fail to word-wrap properly, or double space the text because the line ends are "wrong." Programs that don't expect blank lines crash because they interpret '\r\n' as two lines. Some programs recognize '\r\n' but don't recognize '\n\r'. And so on.

That's what I mean by *detail*. Try coding the horrible logic for sorting out line ends in UML!

No Hope, No Change

The hope of the MDA movement was that a great deal of detail could be eliminated by using diagrams instead of code. That hope has so far proven to be forlorn. It turns out that there just isn't that much extra detail embedded in code that can be eliminated by pictures. What's more, pictures contain their own accidental details. Pictures have their own grammar and syntax and rules and constraints. So, in the end, the difference in detail is a wash.

The hope of MDA was that diagrams would prove to be at a higher level of abstraction than code, just as Java is at a higher level than assembler. But again, that hope has so far proven to be misplaced. The difference in the level of abstraction is tiny at best.

And, finally, let's say that one day someone does invent a truly useful diagrammatic language. It won't be architects drawing those diagrams, it will be programmers. The diagrams will simply become the new code, and programmers will be needed to *draw* that code because, in the end, it's all about detail, and it is programmers who manage that detail.

Conclusion

Software tools have gotten wildly more powerful and plentiful since I started programming. My current toolkit is a simple subset of that menagerie. I use `git` for source code control, Tracker for issue management, Jenkins for Continuous Build, IntelliJ as my IDE, XUnit for testing, and FITNESSE for component testing.

My machine is a Macbook Pro, 2.8Ghz Intel Core i7, with a 17-inch matte screen, 8GB ram, 512GB SSD, with two extra screens.