Gerhard's Ridiculously Short Guide
to
Software Development Methodologies

This guide is targeted at developers who want to improve what they are doing in their own groups and want a clue as to what is out there to help them.

Twenty Thousand Foot View

From my viewpoint there are several sources of help:

Organizational Standards – the standards which help organizations create consistently better software. These are the "heavy weight" methodologies. Key among these are SEI CMM, ISO 9001, ISO 12207, and ISO 15504. The classic implementation tool is Rational Rose.

Team Methodologies – integrated techniques which can be implemented at a team level. These are the "light weight" methodologies. The current champion is Kent Beck's Extreme Programming. The classic implementation tool is a white board.

Good Advice – non-integrated techniques aka war stories. These are books written by those who have been through the fire and reveal what worked and didn't work for them on an individual level.

Academic Advice – recommendations of those to study how the industry works, but aren't knee deep in daily development.

Organizational Standards

SEI CMM

The Capability Maturity Model, Software from the Software Engineering Institute at Carnegie Mellon is the most important standard for software development. Its not actually a standard as it is a set of practices which will help organization develop software better.

CMM describes organizations as being in one of five levels depending on their practices. Level 5 is the best an organization can do and Level 1 is the worst.

The five levels and key practices are:

1 – Initial. Few processes in place, success depends on individual/team efforts.

- No key practices

2 – Repeatable. Basic process in place to track cost, schedule and functionality. Processes in place which allow the organization to repeat earlier successes.

- Software Configuration Management
- Software Quality Assurance
- Software Subcontract Management

- Software Project Tracking and Oversight
- Software Project Planning
- Requirements Management

3 – Defined. Standardized, documented, integrated software processes.

- Peer Reviews
- Intergroup Coordination
- Software Product Engineering
- Integrated Software Management
- Training Program
- Organization Process Definition
- Organization Process Focus

4 – Managed. Detailed measures of software process and product quality are collected.

- Software Quality Management
- Quantitative Process Management

5 – Optimizing. Continuous process improvement using feedback and pilot projects.

- Process Change Management
- Technology Change Management
- Defect Prevention

For the real lowdown on this see: http://www.sei.cmu.edu/sei-home.html

ISO 9000/9001

OK, I don't know much about this one. Basically just document your procedures then follow the documented procedure.

ISO 12207

OK, I don't know much about this one either. Describes basic software development procedures.

ISO 15504

The SPICE – theoretically a combination of all of the above. I haven't seen this one either. (If your wondering why I don't much about these, its just because I haven't read them. You actually have to pay money to get copies of them and most of the literature I found on the web is for sale not for free.)

Team Methodologies

The heavy-weight methodologies have two things in combination: a) they are designed to be adapted by organizations b) everything needs to be in writing. While some items can be adopted by teams, many are just not applicable at the team level. The light-weight methodologies are easier to implement at the team level.

Extreme Programming

The current most popular light-weight methodology is Extreme Programming by Kent Beck. For complete information see http://www.extremeprogramming.org.

Extreme Programming is a rapid prototyping spiral development methodology. It is based on four values (communication, simplicity, feedback, and courage), four basic activities (coding, testing, listening, and designing), and the following practices:

- The Planning Game – realistic scope of next release
- Small Releases – small system in production followed by lots of small releases
- Metaphor – simple shared story of how whole system works
- Simple Design – design as simply as possible with no extra functionality
- Testing – continuos unit testing
- Refactoring – restructure system without changing functionality
- Pair Programming – two programmers, one machine
- Collective Ownership – anyone can change any code at any time
- Continuous Integration – integrate and build the system many times a day
- 40-hour Week – work no more than 40 hours a week
- On-site Customer – include a real, live user on the team
- Coding Standards – all code written to same standard emphasizing communication

Patterns

Design patterns are not truly a methodology. However, they are a key facilitator to team methodologies. Design patterns seek to identify and name coding techniques. The idea is that if the technique has a name, then it can be discussed. This helps a team doing design work in that they will have a common language for way to solve some common problems.

The classic text for patterns is: *Design Patterns* by Erich Gamma, Richard Helm, Ralph Johnson, John Vlisside (aka the Gang of Four).

Here is a description of the patterns I use most often (Java is assumed):

Factory

Interface WorkerBee = WorkerBeeFactory.makeWorkerBee(parameters)

Your program needs WorkerBee objects to do its job. WorkerBees are described with an interface, not a class. However, there are several different types of WorkerBee objects depending on the circumstances at the time of creation. You use a Factory to create all the different kinds of WorkerBees depending on the parameters passed into the factory. Only the factory understands that there are actually different WorkerBees - the rest of the program only knows the WorkerBee interface. If you find that you need yet another type of WorkerBee, you only need to modify the factory.

Abstract Factory

Interface WorkerBee = WorkerBeeFactoryInterface

Here the factory itself is not a class but an interface. You use different Factories to create different WorkerBees instead of using a single factory to make all the WorkerBees. Use when only one type of WorkerBee will be needed for the life of the factory. Example is the Swing UI factories for Windows, Motif and Metal. Once you have the UI factory, all objects will be of correct type for that particular interface.

Singleton

Only allow one object of that class in an application. Good for globals or for something that locks resources (like access to a single log file). To code this in Java, use a static factory method to get the one instance. The static method will either create the instance or return the already created instance. The constructor must be private so that only the static method in the class can create an instance of the class.

Immutable

Once the object is created, its values can not be changed. Enforced by keeping variables private, not providing any set methods, and not changing the value with any other class methods. Useful for objects that are output by one function which are to be consumed by another function. Works best when the variables in the class are base types not objects - otherwise the variables might allow themselves to be changed.

Wrapper

You have a framework using interface A. You buy class B. You make Class B work with the framework by placing it in a wrapper of interface A. The wrapper does the class B to interface A conversions for the framework.

Decorator

Recursively add functionality. Idea is to keep base class simple & extend as needed with decorators. Can build up exactly the functionality needed by decoration process - avoiding complex classes with too much (or the wrong) functionality. Classic Java examples are streams and panes (e.g. start with a stream and then add buffering). If you find that you have complex internal logic based on flags, you may be able to use decorator classes to replace the flags.

Proxy

Placeholder for the real thing with 1 to 1 functionality. Example: RMI stubs. Use when you don't need the full object hanging around. Basically, proxy is a communication mechanism. Idea is that the real object to do the communication will be snapped in under the proxy when needed at run-time. In the meantime, the proxy will hold its place at compile time.

Composite

Object created with composition. That is, an object containing other objects. Extremely basic concept used all over the place in object-oriented programming.

MVC

Model, View, Controller. Model is logic. View is display. Controller is what changes the model. Nice theoretical mode for doing user interface. Fits nicely with J2EE[1] where EJBs[2] are the model, JSPs[3] are the view, and Servlets are the controller.

Model-Delegate

Derivative of MVC pattern. Combines the View and the Controller into the Delegate. Model is same as in MVC. Often used in GUI work where the View and Controller are too interconnected to be meaningfully separated. Used extensively in Swing (Java's advanced GUI toolkit).

Good Advice

Let's be honest. Good advice is hard to come by. Books filled with good advice from folks who have used it are even harder to find. However, with the right materials, you can self-train yourself in computer programming. I did. (My only two computer courses were APL and COBOL way back in 1976.)

My absolute favorite for this category is *Rapid Development: Taming Wild Software Schedules* by Steve C McConnel. My bible. Buy this book for yourself and use it to help you bosses understand how to do things correctly. It has all the usual advice and studies, is an easy read and does not try to be pedantic.

Here are some other titles to get you going:

Dynamics of Software Development by Jim McCarthy. Good read with some reasonable ideas.

The Mythical Man-Month by Frederick P. Brooks, Jr. Ancient history now (originally published in 1972), but still a must read. Key axiom: throwing more programmers on a late project will make it even later.

1Java 2, Enterprise Edition
2Enterprise Java Beans
3Java Server Pages

Software Engineering, A Practitioner's Approach by Roger S. Pressman. This is just a software engineering 101 text book I happen to buy. Any one of a dozen would suffice. If you didn't read one in college read one now - so you'll at least have heard of the classic approaches to software development.

Code Complete by Steve McConnell. All kinds of good nuggets, targeted at the individual programmer. This is not language specific.

Data Modeling Essentials by Graeme Simsion. Database design and normalization is not a mysterious complicated processes and this book proves it. Great book even if you just want to understand database designs you are given. Even if you think you will never work with database this book can be helpful - you can normalize the data structures in you program. In fact, I recommend using database normalization techniques for object-oriented design.

The Essential Client/Server Survival Guide by Robert Orfali, Dan Harkey, and Jeri Edwards. I really wish there were more books like this. Instead of trying to teach you a specific technology, this book give a brief overview of all the current big technologies. This gives you a good idea of when to use what technology. If you are a system architect, read this book.

The Concise Guide to the IDEF0 Technique, by Steven C. Hill and Lee A. Robinson. Don't rush out and buy this book. IDEF0 is a great business process reengineering technique which is not popular and does not have good software tools. I think it could be really helpful for requirements analysis, especially for large systems (the technique was developed by the Air Force). If you need to use this technique, this is the book.

An Introduction to Information Engineering by Clive Finkelstein. Good academic discussion of database design.

Writing Solid Code by Steve Maguire. Its been a while since I read this, but it seemed a good book at the time.

How to Make Meetings Work by Doyle, M. and D. Straus. Discusses facilitated meeting techniques which eventually worked its way into JAD[4] sessions. Since you will be in many meetings in your life, you may as well try and make them productive.

The Cathedral and the Bazaar by Eric S. Raymond. The open source manifesto. Sold me.

Here are some books I just ordered (I'll update this once I've read them):

- *Creating a Software Engineering Culture* by Karl E. Wiegers
- *The Pragmatic Programmer: From Journeyman to Master* by Andrew Hunt
- *Software Project Survival Guide* by Steve C McConnell
- *Software Requirements* by Karl E. Wiegers

Here are some books that looked good in the book store, but which I was not ready to buy:

- Gerald M. Weinberg's *The Psychology of Computer Programming: Silver Anniversary Edition*. If I just bought all the books from this guy which looked good he'd be my favorite author. Unfortunately, I haven't bought any yet.
- Edward Yourdon's *Death March*. Boy doesn't this sum up an IT career! (Well, let's hope not.)

Academic Advice

This category is for less useful books. The academic label comes from the fact that the books tend to take a higher view with very dry, droll prose. Harder to read and less practical than the books listed above.

4Joint Application Development

A highly recommended book in this category is *The Science of Computing: Exploring the Nature and Power of Algorithms* by David Harel. I never got all the way through this book, but it forever changed my thinking. I learned to try and see the algorithm first, and the programming details later. Since programming languages and environments are forever changing, its important to understand the what, not the how. Most importantly, it clearly shows how some types of problems are computationally impossible.

Books I read and were interesting at the time but which I would not rush out to buy again:

- *Assessment and Control of Software Risks* by Caspers Jones. A classic, but honestly I don't remember it anymore.
- *Decline and Fall of the American Programmer* by Edward Yourdon. Depressing. So far the analysis has been incorrect. See Mr. Yourdon's later book: *Rise & Resurrection of the American Programmer*

Books sitting on my shelf waiting (over a year) for me to read:

- *Constantine on Peopleware* by Larry J. Constantine
- *Information System in Organizations* by Richard Maddision and Geoffrey Darnton
- *Program Smarter, Not Harder* by Jay Johnson, Rod Skoglund and Joe Wisniewski

Classics books which I never got around to buying or reading:

- *Software Engineering Economics* by Barry Boehm.
- *Peopleware by* DeMarco and Lister

Interesting looking book which I never got around to buying or reading:

- *Exploring Requirements Quality Before Design* by Donald C. Gause and Gerald M. Weinberg
- *Are Your Lights On? How to Know What the Problem Really Is* by Donald C. Gause and Gerald M. Weinberg, (Dorset House Publishing, 1989).
- *Rethinking Systems Analysis and Design* by Donald C. Gause and Gerald M. Weinberg, (Dorset House Publishing, 1988).