Based on

* The pragmatic Programmer
* Programming Pearls
* Programming tips from all over

# Tips:

<http://swreflections.blogspot.com.au/2013/11/applying-8020-rule-in-software.html>

* **Take Responsibility. Evaluate risks and mitigate** (no “the cat ate my source code”).   
  Run the conversation in your head – what will they say? What will you answer? Before you go and tell them the bad news, is there anything else you can try?  
  Don’t say it can’t be done; explain what can be done to salvage the situation.  
  **Provide Options, Don’t Make Lame Excuses**
* **Don’t Live with Broken Windows**  
  Don’t leave “broken windows” (bad designs, wrong decisions, or poor code) unrepaired. If there is insufficient time to fix it properly, then board it up (comment it out, return a ‘not implemented’ error, comment etc)
* **Be a catalyst for change – Stone Soup**  
  People find it easier to join an ongoing success. Show them a glimpse of the future and you’ll get them to rally around.   
  To push for a big change – do something small/partial to gather support and then add more.
* **Remember the big picture – Frog Soup**Most software disasters start out too small to notice and most project overruns happen a day at a time. Systems drift from their specs feature by feature… **Stop once in a while (schedule it) and look at the big picture!**
* **Make Quality a Requirements Issue.**  
  The scope and quality of the system you produce should be specified as part of that system’s requirements. Many users would rather use software with some rough edges today than wait a year for the multimedia version. Great software today is often preferable to perfect software tomorrow. If you give your users something to play with early, their feedback will often lead you to a better eventual solution
* **Know When to Stop.**  
  Don’t spoil a perfectly good program by over embellishment and overrefinement. Move on, and let your code stand in its own right for a while. It may not be perfect. Don’t worry: it could never be perfect.
* **Your Knowledge Portfolio**
  1. **Invest regularly.** Just as in financial investing, you must invest in your knowledge portfolio regularly. Even if it’s just a small amount, the habit itself is as important as the sums. It will expand your thinking.
     + ***Learn at least one new language every year*.** Broaden your thinking.
     + **Read a technical book each quarter.** Start with your current technology and then branch out.
     + **Read nontechnical books, too.** Don’t forget your users and programmers - the human side of the equation.
     + **2 courses a year.** Take at least two courses a year.
     + **Participate in local user groups/projects.** Don’t just listen, but actively participate. Also networking.
     + **Experiment with different environments.** Work with different IDEs, source control, build environment and Oss
     + **Stay Current.** Subscribe to a journal. After covering your current project needs, search for other technology.
     + **Get Wired.** Subscribe to a newsgroup/blog/podcast.
  2. **Diversify.** The more different things you know, the more valuable you are. As a baseline, you need to know the ins and outs of the particular technology you are working with currently. But don’t stop there. The face of computing changes rapidly—hot technology today may well be close to useless (or at least not in demand) tomorrow. The more technologies you are comfortable with, the better you will be able to adjust to change.
  3. **Manage risk.** Technology exists along a spectrum from risky, potentially high-reward to low-risk, low-reward standards. It’s not a good idea to invest all of your money in high-risk stocks that might collapse suddenly, nor should you invest all of it conservatively and miss out on possible opportunities. Don’t put all your technical eggs in one basket.
  4. **Buy low, sell high.** Learning an emerging technology before it becomes popular can be just as hard as finding an undervalued stock, but the payoff can be just as rewarding. Learning Java when it first came out may have been risky, but it paid off handsomely for the early adopters who are now at the top of that field.
  5. **Review and rebalance.** This is a very dynamic industry. That hot technology you started investigating last month might be stone cold by now. Maybe you need to brush up on that database technology that you haven’t used in a while. Or perhaps you could be better positioned for that new job opening if you tried out that other language. . . .
  6. **Critical Thinking.** Critically analyse what you read and hear. Don’t get swayed by media hype. Think for yourself.
* **Communicate!**
  1. **Know what you want to say**. What’s your aim?
  2. **Know your audience**. What is their interest/motivation?
  3. **Choose your moment**.
  4. **Choose a style.** Match it to the audience – how much details? How technical? Wordy/tables/graphs?
  5. **Make it look good**. Presentation
  6. **Involve your audience.** Before (prepare) and after (feedback, follow ups)
  7. **Be a listener.** Reflect what people are saying, turn a meeting in to a dialog.
  8. **Get back to people.** Always respond to e-mails and voice mails, even if the response is simply “I’ll get back to you later”.
  9. Ask **expectations** and **prioritization**
  10. Growth Mindset: Schedule **feedback sessions** and be **open to constructive criticism**.
  11. Bring **solutions** not problems.

# Development

## DRY

* Duplicated Data Structures (server/client, different languages etc) -  
  Can you generate the code/structures from a metadata structure automatically?
* Comments in Code:  
  **Comment only high-level explanations. Good code doesn’t require low-level comments.**
* Documentation and Code:  
  Automatically generate the tests from the test-plan from the requirements? Search for tools that can do this.
* Inadvertent Duplication:  
  that’s when you don’t mean to duplicate information but your design create dependency between different components (e.g. start point, end point, length ).  
  **Actively think about avoiding duplication in your design!**
* Impatient Duplication:  
  Don’t get tempted to make short-cuts in coding or design that will cost you more in the long run!
* Inter-developer Duplication:  
  Create a library of common code that can be reused in all projects. Make your code easy to reuse.

## Orthogonality

* Two or more things are orthogonal if changes in one do not affect any of the others. In a well-designed system, the database code will be orthogonal to the user interface: you can change the interface without affecting the database, and swap databases without changing the interface.
* **Eliminate Effects Between Unrelated Things**
* Test if your design is orthogonal:  
  If you dramatically change the requirement or design of a particular function, **how many modules will be affected?** The answer should be as close as possible to ‘one’.
* **Keep you code decoupled.** Write **shy code**: modules don’t reveal anything unnecessary to other modules.
* **Avoid global data!**Explicitly pass any required context to your modules. On object oriented code, it can be as parameters to an object’s constructor or structures containing the context in other code.
* **Avoid similar functions**If you have functions that do the same task with different algorithm – **think Strategy Pattern** (Duck has a Quack Behaviour).
* **Orthogonal Teams**In teams too, people should have orthogonal tasks and not depend too heavily on one another. This will improve efficiency and reduce frictions. A test of team orthogonality: **how many people need to be involved in discussing each requested change?**

## Reversibility

* It is a mistake to assume that any decision is cast in stone—and in not preparing for the contingencies that might arise. Instead of carving decisions in stone, think of them more as being written in the sand at the beach. A big wave can come along and wipe them out at any time.  
  **Do Risk Analysis and include Changes in Technology & Decisions.**

## Tracer Bullets – Agile Development

* Especially when creating a new system, working with new technology etc. Both you and the customers don’t know what you want/need. Instead of specifying everything to death and then praying, use tracer bullets.   
  **Tracer code implement part of the functionality that can demonstrate the system or the risky/new parts of it.** It gets us from a requirement to some aspect of the final system quickly, visibly, and repeatedly. This code can be given to the customer to play with in the real environment to get early feedback on issues and requirements.
* Tracer code is **not disposable:** you write it for keeps. It contains all the error checking, structuring, documentation, and self-checking that any piece of production code has. It simply is not fully functional. However, once you have achieved an end-to-end connection among the components of your system, you can check how close to the target you are, adjusting if necessary. Once you’re on target, adding functionality is easy.
* **Users get to see something working early.   
  I**f you have successfully communicated what you are doing (see Great Expectations, your users will know they are seeing something immature.
* **Developers build a structure to work in.**   
  once you have the framework of the system in place, it’s much easier to add functionality. This makes everyone more productive, and encourages consistency.
* **You have an integration platform.**This will allow you to integrate often and in small chunks. Making finding and debugging problems easier and faster and keep the input from the customers coming.
* **You have something to demonstrate.**
* **You have a better feel for progress.** In a tracer code development, developers tackle use cases one by one.
* **Tracer Code Sometimes Misses.**  
  You may find that this is not what the costumer wanted, you have performance issues or the system doesn’t work like you expected. That’s the whole point of the tracer code – finding these things sooner, with less investment. Use it especially when there are a lot of unknowns in the technology and/or requirements.
* **Prototyping is not Tracer Code**. Prototyping generates disposable code of specific parts of the system, it can be in a different language or with different algorithms.   
  Tracer code is lean but complete, and forms part of the skeleton of the final system. Think of prototyping as the intelligence gathering that takes place before a single tracer bullet is fired.

## Prototype to Learn:

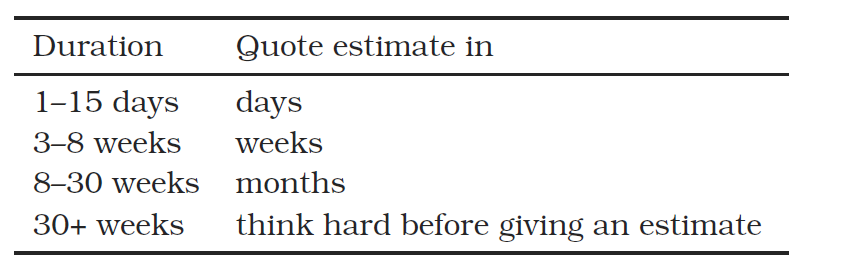
* We use prototyping to analyse and expose risk and to offer a change for correction at a greatly reduced cost. We can target a prototype to test one or more specific aspects of a project.
* **What to Prototype:**We’ll want to prototype anything that carries risk and hasn’t been done before, especially critical aspects of the final system. Anything we’re not comfortable with or not sure how to do. It can be anything:
  1. Architecture
  2. New functionality
  3. Structure or contents of external system
  4. Performance issues
  5. User interface design
* **Different Tools:  
  Prototyping doesn’t have to be with code**: we can use post-it notes for prototyping dynamic things like application logic and workflow, whiteboard for drawing and interface builders for user interfaces.
* **Give up the Details:**We’ll usually concentrate on one aspect of the system and do without the other parts of it. For example: if we’re prototyping the GUI, we can use constant data with no computation etc.
* **Disposable**:  
  **make sure that everyone understands** that prototyping is disposable code!
* **When not to use Prototypes:**
  1. **Complete:** If we can’t give up the details of other parts of the system, maybe we should use tracer bullet instead?
  2. **Non Disposable:** make sure that everyone understands that prototyping is disposable code! If you can’t guarantee that management won’t force you to deploy the prototype – use trace bullets instead.

## Domain Languages

* Computer languages influence how you think about a problem. You should always try to code using the vocabulary of the application domain. Developing and using a domain language is the next step when you’re coding using the vocabulary, syntax and semantics of the problem domain.
* Program Close to the Problem Domain  
  Whether it’s a simple language to configure and control an application program, or a more complex language to specify rules or procedures, we think you should consider ways of moving your project closer to the problem domain. By coding at a higher level of abstraction, you are free to concentrate on solving domain problems, and can ignore petty implementation details.
* It also makes the communication with the users/customers more affective and precise.
* Also report domain-specific errors in a language our users can understand.

## Estimating

Estimate to Avoid Surprises

* Understand how accurate is enough? Depends on the situation: planning feasibility require less accuracy than signing delivery contract with a customer.
* The units you use imply the degree of accuracy in your estimate (‘130 working days’ vs ‘around 6 months’). Scale estimates as follows:  
  
* To estimate:
  1. What is being asked? You have to **understand the scope of the domain**. Think of **all the assumptions** before starting to guess (“assuming there are no traffic accidents and ….”). This should also be included in your answer.
  2. The best starting point for initial estimate is a previous similar project either by you or by someone else.
  3. **Build a mental model of your system**. Include the main steps needed for the project and a rough idea on how it will be implemented.  
     You might run into underlying issues that weren’t apparent at the beginning or even other alternative solutions that can change the scope of the project.  
     You will need to learn when to stop refining (when the effort in continued estimation doesn’t match the additional accuracy).
  4. **Break the model into components**
  5. **Give each component a value**. Try to **concentrate on the critical components** – those that influence the final result in the most critical way.
  6. Calculate the final answer. In most time it will be a range of time and not one specific answer. A lot of times it will also have conditions/assumptions associated with it.
  7. Keep track of your Estimations. To review and improve your future estimations.
  8. **Review your estimates.** If they were off, understand why to improve future estimations.
* **Estimating Projects:**
  1. Usually, the only way to determine the timetable for a project is by gaining experience on that same project (unless you already have experience with a similar project with the same team and technology).
  2. To estimate a project, use incremental development, repeating the following steps:
     + Check requirements
     + Analyse risk and estimate
     + Design, implement, integrate
     + Validate with users.
  3. Don’t try to estimate the whole project in advance – you will just be guessing – you can make initial guesses but treat them as such. Estimate each iteration before starting it. As the project progresses, you’ll gain more understanding and confident in it and your estimations will improve.
  4. Iterate the schedule with the code!
* **I’ll get back to you**That’s the only correct answer to give when asked for an Estimate!

# Tools Box

## Plain Text

### What?

**Keep knowledge in plain text!**  
Save data in plain text (can be also XML/HTML etc) and not in binary so it can be understood without the program interpreting it!

**If you can’t save all the data in plain text: store metadata:**If storing the data in plain text:

* Requires too much space or
* Takes too long to process

You can store the data itself in binary format but add some metadata about the raw data in plain text.

NOTE: storing the data in binary format is no more secure than saving it in plain text. If you want to keep the data secret, encrypt it. If you want to keep it consistent, sign it.

### Why?

Why keep the data in plain text:

* **Insurance against Obsolescence:**  
  Even when the original application no longer works or you don’t know how to run it, you can still recover and use the data.
* **Leverage:**  
  Every tool in the computers universe know how to work with text files. This makes using power tools such as text comparison tools, source control, signing and encrypting etc very easy.
* **Easier Testing:**  
  if the input and output data is in plain text, you can easily create tests and analyse results.
* **Lowest Common Denominator:**  
  The text file will always be supported by all present and future systems not matter what other sophisticated binary-based conventions will be created and scrapped.

## Command Shell

Gain familiarity with the shell and you’ll find that your productivity soars.

**Shell Utilities**

* Cygwin – include a lot of Linux power-tool such as grep, find etc.  
  Comes with Bash shell.

## Power Editor

## Source Control

Use source control for everything you do not just code: use it also for memos, phone lists and anything else that changes over time.

## Debugging

* Make sure your **compiler’s warning levels is as high as possible** and the compilation is clean – it doesn’t make sense to waste time trying to find problems that the compiler can find for us!
* Always **try to discover the root cause of a problem**, not just this particular appearance of it.
* You may need to interview the user who reported the bug in order to gather more data than you were initially given.
* Artificial tests (such as the programmer’s single brush stroke from bottom to top) don’t exercise enough of an application. You must brutally test both boundary conditions and realistic end-user usage patterns. You need to do this systematically.
* Try using a debugger with visualization capabilities e.g. DDD debugger
* Tracing statements (printfs) in the code are invaluable in any system where time itself is a factor: concurrent processes, real-time and event-driven systems.
* **Rubber Ducking:**A very simple but particularly useful technique for finding the cause of a problem is simply to explain it to someone else. The other person should look over your shoulder at the screen, and nod his or her head constantly (like a rubber duck bobbing up and down in a bathtub). They do not need to say a word; the simple act of explaining, step by step, what the code is supposed to do often causes the problem to leap off the screen and announce itself.  
  You can also explain in writing to get this clarity of thought.
* **Select Isn’t Broken:**  
  It is generally more profitable to assume that the application code is incorrectly calling into a library than to assume that the library itself is broken. Even if the problem does lie with a third party, you’ll still have to eliminate your code before submitting the bug report.
* **The Element of Surprise**
  1. When you come across a surprise bug, beyond merely fixing it, you need to determine why this failure wasn’t caught earlier. Consider whether you need to amend the unit or other tests so that they would have caught it.
  2. Also, if the bug is the result of bad data that was propagated through a couple of levels before causing the explosion, see if better parameter checking in those routines would have isolated it earlier .
  3. While you’re at it, are there any other places in the code that may be susceptible to this same bug? Now is the time to find and fix them. Make sure that whatever happened, you’ll know if it happens again.
  4. If it took a long time to fix this bug, ask yourself why. Is there anything you can do to make fixing this bug easier the next time around? Perhaps you could build in better testing hooks, or write a log file analyser.
  5. Finally, if the bug is the result of someone’s wrong assumption, discuss the problem with the whole team: if one person misunderstands, then it’s possible many people do.
* **Don’t Assume It – Prove It**Don’t gloss over a routine or piece of code involved in the bug because you “know” it works. Prove it. Prove it in this context, with this data, with these boundary conditions.
* **Debugging checklist**
  1. Is the problem being reported a direct result of the underlying bug, or merely a symptom?
  2. Is the bug really in the compiler? Is it in the OS? Or is it in your code?
  3. If you explained this problem in detail to a co-worker, what would you say?
  4. If the suspect code passes its unit tests, are the tests complete enough? What happens if you run the unit test with this data?
  5. Do the conditions that caused this bug exist anywhere else in the system?

## Code (or Data) Generators

There are two main types of code/data generators:

### Passive code generators

Are run once to produce a result. From that point forward, the result becomes freestanding—it is divorced from the code generator.

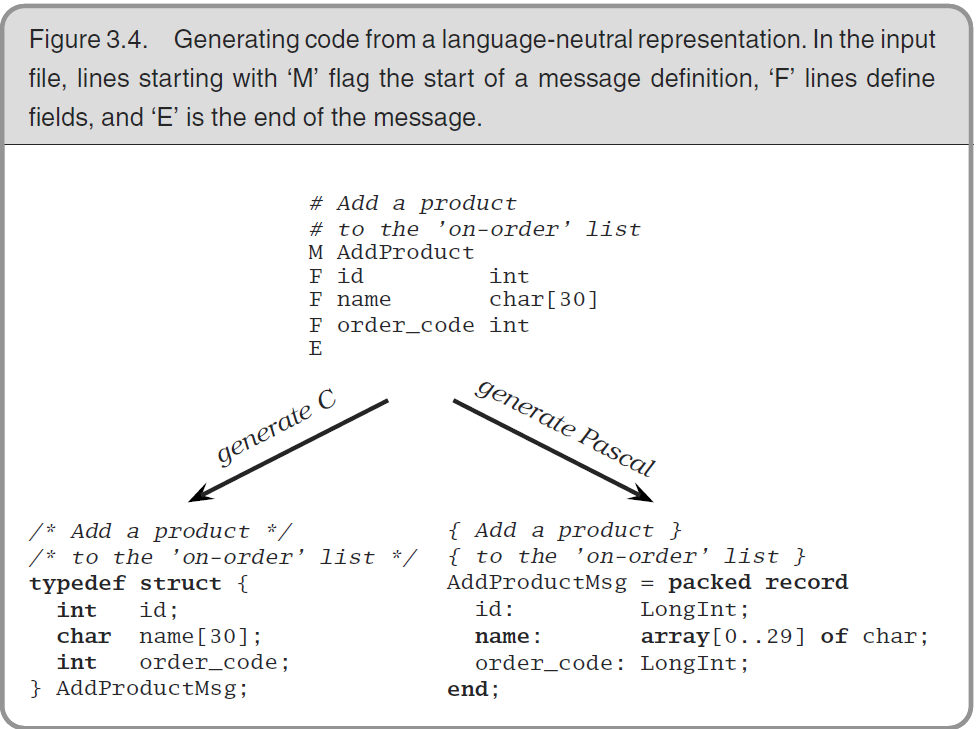
Uses:

* Create new source files. For example to create main or the general structure of a class with comments etc.
* Perform a one-off conversion between programing languages e.g. for porting code.  
  This is an interesting feature of passive code generators: they don’t have to be totally accurate. You get to choose how much effort you put into the generator, compared with the energy you spend fixing up its output.
* Producing lookup tables and other resources that are expensive to compute at runtime e.g. tables of sine and cosine in early graphics systems.

### Active code generators

Are used each time their results are required.  
The result is a throw-away—it can always be reproduced by the code generator. Often, active code generators read some form of script or control file to produce their results.

Active code generators are an important tool to follow the **DRY** principle. Examples:

* Data base scheme and the code that uses it. If we’ll generate the code’s data structures from the database scheme itself, if it changes, the code data structures will change and the compiler will fail to remind us that we need to remove all the code that uses these data structures.
* Data that is used be two different systems/languages. For example: commands and error codes in multi-CPU system. We can have a simple input file that the generator uses to generate the code for the two systems.  
  

**Important:  
The active code generator must be part of the compilation**. Otherwise, it will degrade and won’t be used when required.

# Pragmatic Paranoia

**Protect your code** against other bad code/users, including yourself! The same as you would protect yourself against driving mistakes on the road.  
Stay alert.

**‘When everyone is really out to get you, Paranoia is just good thinking’** (woody Allen)

## Design Principles

* Think about space and time. Do you have limitations? Can you trade more space for better performance or vice versa?
* “A designer knows he has arrived at perfection not when there is no longer anything to add, but when there is no longer anything to take away.” Simple programs are usually more reliable, secure, robust and efficient than their complex cousins, and easier to build and to maintain.

## Design by Contract

Just like between people, every piece of code should have a contract that explicitly state the following:

Pre-conditions: what its conditions on the input

Post conditions: what it guarantees on the output

Invariants: conditions that are always kept true by the code. For example: in a sorted array – the array will remain sorted after the function returns.

In coding language/environment that does not support DBC in the compiled code, use comments for every function and class.

In coding language the does support DBC in code, use this and the compiler will flag any code that violates the contract! This is a very powerful technic!

An excellent answer to the question: why did Design by Contract didn’t catch on (Stack Overflow):

TDD and DbC are two different strategies. DbC permits **fail-fast at runtime** while TDD act "at **compile time**" (to be exact it add a new step right after the compilation to run the unit tests).

That's a big advantage of TDD over DbC : it allows to get earlier feedback. When you write code the TDD way, you get the code and its unit-tests at the same time, you can verify it "works" according to what you thought it should, which you encoded in the test. With DbC, you get code with embedded tests, but you still have to exercise it. IMO, this certainly is one reason that Dbc is not so popular.

Other advantages: TDD creates an **automatic test suite** that allow detecting (read preventing) regressions and make [Refactoring](http://www.refactoring.com/) safe, so that you can **grow your design incrementally**. DbC does not offer this possibility.

Now, using DbC to fail-fast can be very helpful, especially when your code interfaced other components or has to rely on external sources, in which case testing the contract can save you hours.

## Crash, Don’t Trash

Sometimes it’s better to crash the program early when a problem is discovered (e.g. invalid input) than to drag the error on, trying to recover from it at a later stage.

Crashing early can give the user a clearer understanding on what they did wrong.

In addition, it might prevent data corruption later on which can be much worse than crashing.

when your code discovers that something that was supposed to be impossible just happened, your program is no longer viable. Anything it does from this point forward becomes suspect, so terminate it as soon as possible. A dead program normally does a lot less damage than a crippled one.

## Assertive Programming

If It Can’t Happen, Use Assertions to Ensure That It Won’t!

The easiest way to do this is with assertions. In most C and C++ implementations, you’ll find some form of *assert* or *\_assert* macro that checks a Boolean condition. These macros can be invaluable.

Don’t use assertions in place of real error handling. Assertions check for things that should never happen.

Just because the supplied assert macros call exit when an assertion fails, there’s no reason why versions you write should. If you need to free resources, have an assertion failure generate an exception, longjmp to an exit point, or call an error handler. Just make sure the code you execute in those dying milliseconds doesn’t rely on the information that triggered the assertion failure in the first place.

### Leave Assertion Turned On

Assertions add a small overhead for the program performance. However, they might also be critical for detecting errors in the code or environment or preventing data corruption.

You might need to turn off some of the assertions but always leave the critical ones in the production code!

## Use Exceptions for Exceptional Problems

Exceptions should rarely be used as part of a program’s normal flow; exceptions should be reserved for unexpected events.

If the program doesn’t know if a file should exist or not, it should not throw an exception but instead return a conventional error. If on the other hand, a program relies of a file existing (e.g. password file on Linux), it can throw an exception if it disappears.

An exception represents an immediate, nonlocal transfer of control—it’s a kind of cascading goto. Programs that use exceptions as part of their normal processing suffer from all the readability and maintainability problems of classic spaghetti code. These programs break encapsulation: routines and their callers are more tightly coupled via exception handling.

### Error Handlers Are an Alternative

An error handler is a routine that is called when an error is detected. You can register a routine to handle a specific category of errors. When one of these errors occurs, the handler will be called.

This allows the calling code to decide how to handle specific types of errors and the low level code does not need to know the details. It just calls the passed error handler.

## How to Handle Resources

### Finish What You Start

The routine or object that allocates a resource should be responsible for

deallocating it.

This will prevent problems with leaving resources open.

### Nest Allocations

1. Deallocate resources in the opposite order to that in which you allocate them. That way you won’t orphan resources if one resource contains references to another.

2. When allocating the same set of resources in different places in your code, always allocate them in the same order. This will reduce the possibility of deadlock. (If process A claims resource1 and is about to claim resource2, while process B has claimed resource2 and is trying to get resource1, the two processes will wait forever.)

### Encapsulating Resources in Objects

If you are programming in an object-oriented language, you may find it useful to encapsulate resources in classes. Each time you need a particular resource type, you instantiate an object of that class. When the object goes out of scope, or is reclaimed by the garbage collector, the object’s destructor then deallocates the wrapped resource.

This method is also very useful when you’re working with exceptions and want to verify that all the pointers you allocate get deleted in all cases (both normal execution and exception). You can initialise the pointer object at the start of the routine and know that it will be destroyed when it goes out of scope. Since its destruction delete the memory it uses, you will not have memory leaks.

In Java, since it uses garbage collector and you can’t control when it’s run, you can use the exception ‘finally’ clause to release any allocated resources.   
Note: the ‘finally’ will be called after the ‘try’ clause whether the execution was successful or not and therefore, you can rely on it.

### When you Can’t Balance Resources

There are times when the basic resource allocation pattern just isn’t appropriate.

The trick here is to establish a semantic invariant for memory allocation. You need to decide who is responsible for data in an aggregate data structure. What happens when you deallocate the top-level structure?

You have three main options:

1. The top-level structure is also responsible for freeing any substructures that it contains. These structures then recursively delete data they contain, and so on.

2. The top-level structure is simply deallocated. Any structures that it pointed to (that are not referenced elsewhere) are orphaned.

3. The top-level structure refuses to deallocate itself if it contains any substructures.

The choice here depends on the circumstances of each individual data structure. However, you need to make it explicit for each, and implement your decision consistently. Implementing any of these options in a procedural language such as C can be a problem: data structures themselves are not active. Our preference in these circumstances is to write a module for each major structure that provides standard allocation and deallocation facilities for that structure. (This module can also provide facilities such as debug printing, serialization, deserialization, and traversal hooks.)

### Checking the Balance

It is always a good idea to build code that actually checks that resources are indeed freed appropriately. For most applications, this normally means producing wrappers for each type of resource, and using these wrappers to keep track of all allocations and deallocations.

At certain points in your code, the program logic will dictate that the resources will be in a certain state: use the wrappers to check this.

For example, a long-running program that services requests will probably have a single point at the top of its main processing loop where it waits for the next request to arrive. This is a good place to ensure that resource usage has not increased since the last execution of the loop.

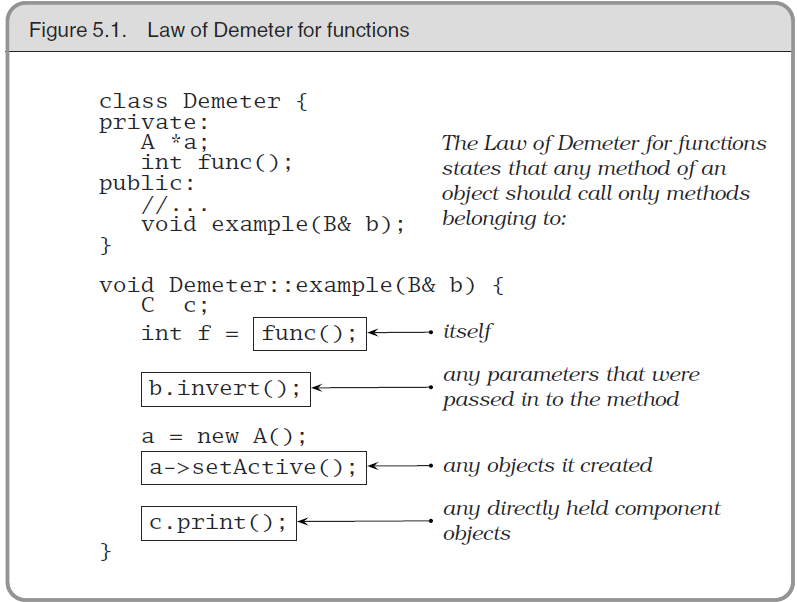
At a lower, but no less useful level, you can invest in tools that (among other things) check your running programs for memory leaks. Purify (www.rational.com) and Insure++ (www.parasoft.com) are popular choices.

# Bend or Break

## Decoupling and the Law of Demeter

The Law of Demeter for functions attempts to minimize coupling between modules in any given program. It tries to **prevent you from reaching into an object to gain access to a third object’s methods**.

By writing “shy” code that honours the Law of Demeter as much as possible, we can achieve our objective: Minimize Coupling Between Modules



Calling a function on an object that we received from calling another function on one of the allowed objects is not allowed

Example:

**public void** showBalance(BankAccount acct) {

Money amt = acct.getBalance(); // ok – passed parameter

printToScreen(amt.printFormat()); // not ok – calling a // function of an   
 // object we don’t own

}

Using The Law of Demeter will make your code more adaptable and robust, but at **a cost**: as a “general contractor”, your module must delegate and manage any and all subcontractors directly, without involving clients of your module. In practice, this means that you will be writing **a large number of wrapper methods that simply forward the request on to a delegate**. These wrapper methods will impose both a runtime cost and a space overhead, which may be significant—even prohibitive—in some applications.

**As with any technique, you must balance the pros and cons for *your* particular application.** In database schema design it is common practice to “denormalise” the schema for a performance improvement

## Metadata

program for the general case, and put the specifics somewhere else—outside the compiled code base:

Put Abstractions in Code, Details in Metadata

Benefits of using metadata:

* It forces you to decouple your design, which results in a more flexible and adaptable program.
* It forces you to create a more robust, abstract design by deferring details—deferring them all the way out of the program.
* You can customize the application without recompiling it. You can also use this level of customization to provide easy work-arounds for critical bugs in live production systems.
* Metadata can be expressed in a manner that’s much closer to the problem domain than a general-purpose programming language might be.
* You may even be able to implement several different projects using the same application engine, but with different metadata.

Implementation Notes:

* The best format for the metadata is in a text file. This way, users can look at it, understand and change it without running the program.
* You can load and configure the program according to the metadata on startup. However, if it’s a long-running server for example, you might want to enable it to re-configure while running so you won’t have to restart it if the configuration has changed.

**Examples:**

* Parameters of the serial port to use (COM-PORT, bits, parity etc)
* Sample values and results for use in unit testing.

## Temporal Coupling

Time is an often ignored aspect of software architectures.

There are two aspects of time that are important to us: concurrency (things happening at the same time) and ordering (the relative positions of things in time).

### Workflow

On many projects, we need to model and analyse the users’ workflows as part of requirements analysis. We’d like to find out what *can* happen at the same time, and what must happen in a strict order. One way to do this is to capture their description of workflow using a notation such as the *UML activity diagram* You can use activity diagrams to maximize parallelism by identifying activities that *could be* performed in parallel, but aren’t.

### Design Using Services

A way to get quick and dirty load balancing:

Among multiple consumer processes: the *hungry consumer* model:

In a hungry consumer model, you replace the central scheduler with a number of independent consumer tasks and a centralized work queue. Each consumer task grabs a piece from the work queue and goes on about the business of processing it. As each task finishes its work, it goes back to the queue for some more. This way, if any particular task gets bogged down, the others can pick up the slack, and each individual component can proceed at its own pace. Each component is temporally decoupled from the others.

### Design for Concurrency

With linear code, it’s easy to make assumptions that lead to sloppy programming. But concurrency forces you to think through things a bit more carefully—you’re not alone at the party anymore. Because things can now happen at the “same time,” you may suddenly see some time-based dependencies.

To begin with, **any global or static variables must be protected from concurrent access.**

In addition, **you need to make sure that your objects are always in a consistent and meaningful state.** I need to make sure that you present consistent state information, regardless of the order of calls. For example, when is it valid to query the state of your object? If your object is in an invalid state between certain calls, you may be relying on a coincidence that no one can call your object at that point in time.

### Cleaner Interfaces

Thinking about concurrency and time-ordered dependencies can lead you to design cleaner interfaces as well.

An example on non-thread safe code and how not to do things is the C library function strtok.

An example of how to design a string tokens extractor in a better, thread-safe way is the Java library class StringTokenizer

### Deployment

Once you’ve designed an architecture with an element of concurrency, it becomes easier to think about handling *many* concurrent services:

the model becomes pervasive.

Now you can be flexible as to how the application is deployed: standalone, client-server, or *n*-tier. By architecting your system as independent services, you can make the configuration dynamic as well. By planning for concurrency, and decoupling operations in time, you have all these options—including the stand-alone option, where you can choose *not* to be concurrent.

Going the other way (trying to add concurrency to a non-concurrent application) is *much* harder. If we design to allow for concurrency, we can more easily meet scalability or performance requirements when the time comes—and if the time never comes, we still have the benefit of a cleaner design.

## It’s Just a View

Events is a great way to decouple objects.

### Publish/Subscribe

This is the Observer pattern. In can be implemented if different ways depending on its intended use:

* Objects may use publish/subscribe on a peer-to-peer basis (classic Observer pattern)
* You can use a “software bus” where a centralized object maintains the database of listeners and dispatches messages appropriately. An example for this is the COBRA Event Service that allows implementing the publish/Subscribe pattern across different programing languages, systems and locations.
* You might even have a scheme where critical events get broadcast to all listeners—registered or not.

### Model-View-Controller

* **Model:** The Model is the data that you operate on, including common operations to manipulate it.   
  ***The model have not knowledge of any Viewers or Controllers.***
* **View:** The View is the way the data is displayed e.g. web-page, spreadsheet, graph etc. In can also be a non-graphical way for example, different subsets of the data or the data manipulated in different ways (e.g. currencies?)  
  ***It subscribes to changes in the Model and logical events from the Controller.***
* **Controller:** The Controller allows user interaction with the View e.g. buttons on forms etc. In can also be non-graphical way for example: different sets of commands for different types of users (e.g. administrators vs users).  
  ***It publishes events to both the Model and the View.***

This is the key concept behind the Model-View-Controller (MVC) idiom:  
Separating the model from both the GUI that represents it and the controls that manage the view.  
By doing so, you can take advantage of some interesting possibilities. You can support multiple views of the same data model. You can use common viewers on many different data models. You can even support multiple controllers to provide nontraditional input mechanisms.

Note that **The view and controller are tightly coupled**, and in some implementations of MVC the view and controller are a single component.

By loosening the coupling between the model and the view/controller, you buy yourself a lot of flexibility at low cost. In fact, this technique is one of the most important ways of maintaining reversibility.

## Blackboard Pattern

### Pattern

The Blackboard pattern is a [design pattern](http://social.technet.microsoft.com/wiki/contents/articles/13207.software-design-pattern.aspx), used in software engineering, to **coordinate separate, disparate systems that need to work together**, or in sequence, continually prioritizing the actors (or knowledge sources).  
  
It is defined as a [behavioral design pattern](http://social.technet.microsoft.com/wiki/contents/articles/13209.behavioural-design-pattern.aspx) because it affects when and how programs react and perform.  
  
The blackboard consists of a number of stores or "global variables", like a repository of messages, which can be accessed by separate autonomous processes, which could potentially be physically separate. A "controller" monitors the properties on the blackboard and decides which actors (or knowledge sources) to prioritize.

**Use Blackboards to Coordinate Workflow where the results of one process affect other processes.**

Think police investigation balckboard where:

* The blackboard contains all the relevant data for the investigation
* Different people with different skills can put relevant data on the boards
* Different people with different skills can read the board and use the data on it to further the investigation.

The main operations in the Blackboard Pattern are:

* **Read:** Search for and retrieve data from the space.
* **Write:** Put an item into the space.
* **Take:** Similar to read, but removes the item from the space as well.
* **Notify:** Set up a notification to occur whenever an object is written that matches the template.

**Usage Examples:**

A common example of this pattern is in speech recognition. Separate threads can process different parts of the sound sample, updating the blackboard with words that have been recognized. Then another process can pick up these words and perform grammar and sentence formation. Meanwhile more words and meanings are coming in, and eventually even higher level processes can pick up the formed sentences and various alternative guesses and begin to formulate it's meaning, then further intelligence systems can start to choose the most appropriate answer. All these systems have access to the blackboard and work together through it's central platform.

# While You Are Coding

## Don’t Code by Coincidence

### Don’t Rely on Undocumented Behaviour

When using libraries, don’t assume that you understand how a function behaves only because it returned some value for some function call. It can be an accident that will be fixed in the next release or it can be a coincidence you don’t fully understand.

For routines you call, **rely only on documented behaviour**. If you can’t, for whatever reason, then document your assumption well.

**For code you write** that others will call, the basic principles of good modularization and of hiding implementation behind **small, well-documented interfaces** can all help. A well-specified contract can help eliminate misunderstandings.

### Don’t Assume – Prove/Document

Never code under undocumented assumptions.

If you can, prove your assumptions (e.g. rely on documented and well tested library). Otherwise, document your assumptions so that they will be clear to your future self/other developers.

### Program Deliberately

* Don’t play around until your code work. **Plan what you want to do and understand why you’re doing it.**
* **Don’t code blindfolded**. Understand the application you’re building as well as any new technology you plan on using.
* **Rely only on reliable things**. Don’t depend on accidents or assumptions. If you can’t tell the difference in particular circumstances, **assume the worst**.
* **Document your assumptions**. *Design by Contract* can help clarify your assumptions in your own mind, as well as help communicate them to others.
* Don’t just test your code, but **test your assumptions as well**. Don’t guess; actually try it. Write an assertion to test your assumptions. If your assertion is right, you have improved the documentation in your code. If you discover your assumption is wrong, then count yourself lucky.
* **Prioritize your effort**. Spend time on the important aspects; more than likely, these are the hard parts. If you don’t have fundamentals or infrastructure correct, brilliant bells and whistles will be irrelevant.
* **Don’t be a slave to history**. Don’t let existing code dictate future code. All code can be replaced if it is no longer appropriate. Even within one program, don’t let what you’ve already done constrain what you do next—be ready to refactor. This decision may impact the project schedule. The assumption is that the impact will be less than the cost of *not* making the change.

**Next time something seems to work, but you don’t know why,**

**Make sure it isn’t just a coincidence.**

## Algorithm Speed

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **O()** | **Description** | **Example** |
| Simple Loops | O(n) | Exhaustive searches/manipulation in and array | Find min/max, calculate checksum etc. |
| Nested Loops | O(m x n) O(n^2) | Simple sorting algorithms e.g. bubble sort |  |
| Binary Chop | O(lg(n)) | If your algorithm halves the set of things it considers each time around the loop | Binary search or a sorted loop, traversing a binary tree |
| Divide and Conquer | O(n lg(n)) | Algorithms that partition their input, work  on the two halves independently, and then combine the result | Quicksort (on average. On worst case scenario – it’s O(n^2) |
| Combinatoric | O(n!)  O(c^n) | Whenever algorithms start looking at the permutations  of things, their running times may get out of hand |  |

Don’t Reinvent the Wheel - Most library implementation will outperform anything you’ll write.

However, using loops to go over or manipulate data pop up time and time again. Whenever you find yourself writing a simple loop, you know that you have an O(n) algorithm. If that loop contains an inner loop, then you’re looking at O(n^2). You should be asking yourself how large these values can get:

* **If the numbers are bounded, then you’ll know how long the code will take to run**.
* **If the numbers depend on external factors** (such as the number of records in an overnight batch run, or the number of names in a list of people), **then you might want to stop and consider** the effect that large values may have on **your running time or memory consumption**.

How to overcome potential problems:

* If you have an inefficient algorithm, try to see if you can divide and conquer to reduce the complexity of it (e.g. from O(n^2) to O(n lg(n))
* If you’re not sure how long your code will take or how much memory it consumes, try running it with different data sizes and plot the result. You should be able to see the shape of the curve. You can also use code profilers if it’s tricky getting accurate timings.
* Considered different types of input (e.g. random data vs ordered list)
* Test your estimations with real data.
* be wary of *premature optimization*. It’s always a good idea to make sure an algorithm really is a bottleneck before investing your precious time trying to improve it.

## Refactor Early, Refactor Often

When you come across a piece of code that isn’t quite as it should be, fix both it and everything that depends on it. Manage the pain: **if it hurts now, but is going to hurt even more later, you might as well get it over with**.

Any number of things may cause code to qualify for refactoring:

* Duplication. You’ve discovered a violation of the *DRY* principle
* Nonorthogonal design. You’ve discovered some code or design that could be made more orthogonal.
* Outdated knowledge. Things change, requirements drift, and your knowledge of the problem increases. Code needs to keep up.
* Performance. You need to move functionality from one area of the system to another to improve performance.

**Keep track of the things that need to be refactored**. If you can’t refactor something immediately, make sure that it gets placed on the schedule.

Make sure that users of the affected code *know* that it is scheduled to be refactored and how this might affect them.

Rules for easy refactoring:

1. Don’t try to refactor and add functionality at the same time.
2. Make sure you have good tests before you begin refactoring. Run the tests as often as possible. That way you will know quickly if your changes have broken anything.
3. Take short, deliberate steps: move a field from one class to another, fuse two similar methods into a superclass. Refactoring often involves making many localized changes that result in a larger-scale change. If you keep your steps small, and test after each step, you will avoid prolonged debugging.
4. It can also be helpful to make sure that drastic changes to a module—such as altering its interface or its functionality in an incompatible manner—break the build. That is, old clients of this code should fail to compile. You can then quickly find the old clients and make the necessary changes to bring them up to date.

## Code that’s Easy to Test

**Test your Software, or Your Users Will**

### Unit Testing

Each module should have unit tests that check and verify it against its contract. It will verify that:

* The unit meets the contract
* The contract means what we think it means.

### Test Window

We also need a way to monitor inside a running system in production environment. These are some of the methods:

* Log files. Make sure they are consistent and can be automatically parsed.
* “Hot-Key” combination. A secret (from the users) key combinations that cause the program to start printing diagnostic/debug information.
* Implement an HTTP web server as the debug and diagnostic port where users can go to see debug output and control.

# Before the Project

## The Requirements Pit

**Make sure you understand and solve the right problem** and not some representation of the problem mixed in with user assumptions, business policies etc.

Many times policies get mixed up with requirements, creating a harder system to maintain. For example:

“Only an employee’s supervisors and the personnel department may view that employee’s records.”

Mixes the requirement:

“An employee record may be viewed only by a nominated group of people.”

With the business policy (that may change over time) that we saw above.

Document these policies separately from the requirement, and hyperlink the two. Make the requirement the general statement, and give the developers the policy information as an example of the type of thing they’ll need to support in the implementation.

Documenting the reasons behind requirements will give your team invaluable information when making daily implementation decisions.

**Work with a User to Think Like a User**

As well as giving you invaluable insight into how the system will *really* be used, you’d be amazed at how the request “May I sit in for a week while you do your job?” helps build trust and establishes a basis for communication with your users.

## Requirements Checklist

The following should be addressed as part of the requirements, in additional to the business and functional requirements:

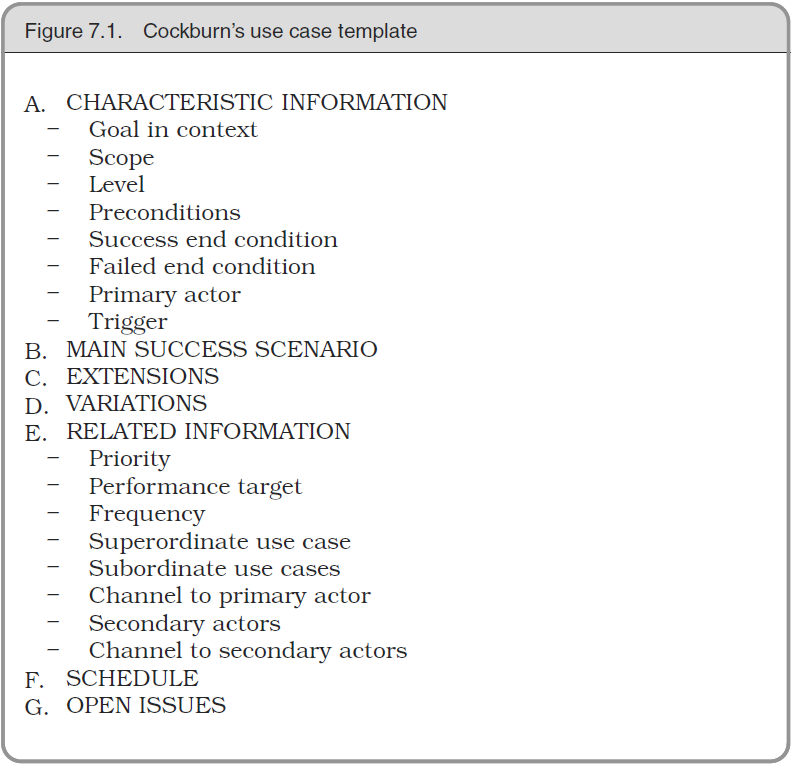
* Environment – what operating system/server etc.
* Error Handling and Fault Tolerance
* Quality – what is the quality requirements of the system?
* Security
* Performance
* Scalability

## Requirements Documentation

**Use cases is an excellent way to capture requirements** since they tend to stick to what the user actually requires without mixing in design decisions as requirements.

One way of looking at use cases is to emphasize their goal-driven nature. Alistair Cockburn has a paper that describes this approach, as well as templates that can be used (strictly or not) as a starting place

By using a formal template, you can be sure that you include all the information you need in a use case: performance characteristics, other involved parties, priority, frequency, and various errors and exceptions that can crop up ( “nonfunctional requirements”).  
This is also a great place to record user comments such as ”oh, except if we get a xxx condition, then we have to do yyy instead. “ The template also serves as a ready-made agenda for meetings with your users.



### Don’t Over-Specify

A big danger in producing a requirements document is being too specific.

Where requirements are concerned, the simplest statement that accurately reflects the business need is best. This doesn’t mean you can be vague—you must capture the underlying semantic invariants as requirements. However, document the specific or current work practices as policy and not as an integral part of the requirements.

**Requirements are not architecture. Requirements are not design, nor are they the user interface. Requirements are *need*.**

Be careful not to document design decision as requirements that will prevent you from changing the design or implementation later on.

### Requirements Creep

The key to managing growth of requirements is to **point out each new feature’s impact on the schedule to the project sponsors.** When the project is a year late from initial estimates and accusations start flying, it can be helpful to have an accurate, complete picture of how, and when, requirements growth occurred.

It’s easy to get sucked into the “just one more feature” maelstrom, but by tracking requirements you can get a clearer picture that “just one more feature” is really the fifteenth new feature added this month.

### Maintain a Glossary

As soon as you start discussing requirements, users and domain experts will use certain terms that have specific meaning to them.

Create and maintain a *project glossary*—one place that defines all the specific terms and vocabulary used in a project. All participants in the project, from end users to support staff, should use the glossary to ensure consistency. This implies that the glossary needs to be widely accessible—a good argument for Web-based documentation.

## Solving Impossible Puzzles

### Don’t Think Outside the Box – Find the Box

When faced with an intractable problem, **enumerate *all* the possible avenues you have before you.** **Don’t dismiss anything, no matter how unusable or stupid it sounds.** Now go through the list and explain why a certain path cannot be taken. Are you sure? Can you *prove* it?

Consider the Trojan horse—a novel solution to an intractable problem. How do you get troops into a walled city without being discovered? You can bet that “through the front door” was initially dismissed as suicide.

Categorize and prioritize your constraints. When woodworkers begin a project, they cut the longest pieces first, then cut the smaller pieces out of the remaining wood. In the same manner, we want to **identify the most restrictive constraints first, and fit the remaining constraints within them**.

## There Must Be an Easier Way!

Sometimes you will find yourself working on a problem that seems much harder than you thought it should be. Maybe it feels like you’re going down the wrong path—that there must be an easier way than this! Perhaps you are running late on the schedule now, or even despair of ever getting the system to work because this particular problem is “impossible.”

That’s when you **step back a pace and ask yourself these questions:**

* *Is* there an easier way?
* Are you trying to solve the right problem, or have you been distracted by a peripheral technicality?
* *Why* is this thing a problem?
* What is it that’s making it so hard to solve?
* Does it have to be done this way?
* Does it have to be done at all?

Many times a surprising revelation will come to you as you try to answer one of these questions. Many times a reinterpretation of the requirements can make a whole set of problems go away—just like the Gordian knot.

**All you need are the real constraints, the misleading constraints, and the wisdom to know the difference.**

## Not before You’re Ready

When you feel a nagging doubt, or experience some reluctance when faced with a task, heed it. You may not be able to put your finger on exactly what’s wrong, but give it time and your doubts will probably crystallize into something more solid, something you can address. Software development is still not a science. Let your instincts contribute to your performance.

### When in Doubt - Prototype

how can you tell when you’re simply procrastinating, rather than responsibly waiting for all the pieces to fall into place?

**Start prototyping:**  
Choose an area that you feel will be difficult and begin producing some kind of proof of concept. One of two things will typically happen. Shortly after starting, you may feel that you’re wasting your time. This boredom is probably a good indication that your initial reluctance was just a desire to put off the commitment to start. Give up on the prototype, and hack into the real development.

On the other hand, as the prototype progresses you may have one of those moments of revelation when you suddenly realize that some basic premise was wrong. Not only that, but you’ll see clearly how you can put it right. You’ll feel comfortable abandoning the prototype and launching into the project proper. Your instincts were right, and you’ve just saved yourself and your team a considerable amount of wasted effort.

**Don’t forget to dispose:** remember, prototype is throw-away code. When you’re ready to start the real development, throw away the prototype and start writing the real code. The last thing you want is to find yourself several weeks into serious development before remembering that you started out writing a prototype.

## The Specification Trap

Finally, there is the straightjacket effect. A design that leaves the coder no room for interpretation robs the programming effort of any skill and art. Some would say this is for the best, but they’re wrong. Often, it is only during coding that certain options become apparent.

We recognize that there are times where incredibly detailed specifications are demanded—for contractual reasons, because of the environment where you work, or because of the nature of the product you are developing.

Just be aware that you reach a point of diminishing, or even negative, returns as the specifications get more and more detailed. Also be careful about building specifications layered on top of specifications, without any supporting implementation or prototyping; it’s all too easy to specify something that can’t be built.

The longer you allow specifications to be security blankets, protecting developers from the scary world of writing code, the harder it will be to move on to hacking out code. **Don’t fall into this specification spiral: at some point, you need to start coding!** If you find your team all wrapped up in warm, comfy specifications, break them out. Look at prototyping, or consider a tracer bullet development.

# Project Management

## Team Principles

* **No broken windows:**  
  Teams as a whole should not tolerate broken windows. The team must take responsibility for the quality of the product, supporting developers who understand the no broken windows philosophy and encouraging those who haven’t yet discovered it.
* **Boiled Frogs:**It’s even easier for teams as a whole to get boiled. Fight this. Make sure everyone actively monitors the environment for changes. Maybe appoint a chief water tester. Have this person check constantly for increased scope, decreased time scales, additional features, new environments—anything that wasn’t in the original agreement.  
  The team needn’t reject changes out of hand—you simply need to be aware that they’re happening. Otherwise, it’ll be you in the hot water.
* **Communicate:**  
  Great project teams have a distinct personality. People look forward to meetings with them, because they know that they’ll see a well-prepared performance that makes everyone feel good. The documentation they produce is crisp, accurate, and consistent. The team speaks with one voice. They may even have a sense of humor.
* **Don’t Repeat Yourself:**
  + Some teams appoint a member as the **project librarian**, responsible for coordinating documentation and code repositories. Other team members can use this person as the first port of call when they’re looking for something. A good librarian will also be able to spot impending duplication by reading the material that they’re handling.
  + When the project’s too big for one librarian (or when no one wants to play the role), **appoint people as** **focal points** for various functional aspects of the work. If people want to talk over date handling, they should know to talk with Mary. If there’s a database schema issue, see Fred.
  + And don’t forget the value of **groupware systems** and local Usenet newsgroups for communicating and archiving questions and answers.

## Team Creation

**Organise around functionality, not job description**

* Instead of a team of architects, a team of developers and a team of testers, have teams with all these rolls (or have the team divide the rolls themselves) that are responsible of one part of the system from start to finish.
* Organize your teams using the same techniques we use to organize code, using techniques such as contracts (Design by Contract), decoupling and orthogonality. This help isolate the team as a whole from the effects of change. If the user suddenly decides to change database vendors, only the database team should be affected. Should marketing suddenly decide to use an off-the-shelf tool for the calendar function, the calendar group takes a hit. Properly executed, this kind of group approach can dramatically reduce the number of interactions between individuals’ work, reducing time scales, increasing quality, and cutting down on the number of defects. This approach can also lead to a more committed set of developers. Each team knows that they alone are responsible for a particular function, so they feel more ownership of their output.
* However, this approach works only with responsible developers and strong project management. Creating a pool of autonomous teams and letting them loose without leadership is a recipe for disaster**. The project needs at least two “heads”—one technical, the other administrative.**  
  The technical head sets the development philosophy and style, assigns responsibilities to teams, and arbitrates the inevitable “discussions”between people. The technical head also looks constantly at the big picture, trying to find any unnecessary commonality between teams that could reduce the orthogonality of the overall effort.   
  The administrative head, or project manager, schedules the resources that the teams need, monitors and reports on progress, and helps decide priorities in terms of business needs. The administrative head might also act as the team’s ambassador when communicating with the outside world.
* Teams on larger projects need additional resources: a librarian who indexes and stores code and documentation, a tool builder who provides common tools and environments, operational support, and so on.

## Automation

Automation is an essential component of every project team. To ensure that things get automated, appoint one or more team members as *tool builders* to construct and deploy the tools that automate the project drudgery. Have them produce makefiles, shell scripts, editor templates, utility programs, and the like. And of course, tie everything together with a continuous integration and deployment system.

### Compiling the Project

We generally compile projects with makefiles, even when using an IDE environment. There are several advantages in using makefiles. It is a scripted, automatic procedure. We can add in hooks to generate code for us, and run regression tests automatically. IDEs have their advantages, but with IDEs alone it can be hard to achieve the level of automation that we’re looking for. We want to check out, build, test, and ship with a single command.

### Generating Code

In *The Evils of Duplication*, we advocated generating code to derive knowledge from common sources. We can exploit make’s dependency analysis mechanism to make this process easy. It’s a pretty simple matter to add rules to a makefile to generate a file from some other source automatically. We can use the same sort of rules to generate source code, header files, or documentation automatically from some other form as well

## Testing

### Unit Testing

### Integration Testing

### Validation and Verification

Apart from testing that the system comply with the requirements, the testing should be conscious of the end-user usage and access patterns and how they differ from the developer test data.

### Resource Exhaustion, Errors and Recovery

Now that you have a pretty good idea that the system will behave correctly under ideal conditions, you need to discover how it will behave under *real-world* conditions, when it runs out of disk space, the network will not work or the battery will run out.

You need to define all the scenarios that your system needs to handle and how it should handle them and test it!

### Performance Testing

Performance testing, stress testing, or testing under load may be an important aspect of the project as well.

### Usability Testing

Usability testing is different from the types of testing discussed so far.

It is performed with real users, under real environmental conditions. Look at usability in terms of human factors. Were there any misunderstandings during requirements analysis that need to be addressed?

Does the software fit the user like an extension of the hand?

As with validation and verification, you need to perform usability testing as early as you can, while there is still time to make corrections.

### Regression Testing

### Test Data

* Real world data
* Synthetic data – generated specifically to test certain conditions/boundaries

### Testing GUI Systems

* Decoupling will allow you to test all the application’s logic, without the GUI and then test the GUI without worrying about the application’s logic.

### Testing the Tests

Use Saboteurs to Test Your Testing - The saboteur’s role is to take a separate copy of the source tree, introduce bugs on purpose, and verify that the tests will catch them.

### Testing Thoroughly

* Coverage analysis tools can give you a general feel for how comprehensive your testing is.
* However, they won’t give you the whole picture. Try to **focus on testing State Coverage** and not Code Coverage when you are writing your test plan.

### When to Test

Early, always and automatically!

### Tightening the Net

If a bug slips through the net of existing tests, you need to add a new test to trap it next time.

### Learning

Some new developers are afraid to enter into the realm of testing, but it represents one of the most effective ways of understanding the flow of execution and functionalities of a piece of software. It can be especially helpful for people who are new at a project and want to learn the structure and workings of it.

If you want to learn how a particular open source framework works, you can improve your grasp of it by writing tests. This way, you will not only understand that framework a lot better, but your tests can contribute to the project as well.

In cases where you are working on a large project and something is unclear to you, you can break the execution flow down into several components and write tests for each one of them.

Writing tests will not only help your understanding, but it will also allow you to find potential bugs, and as a result contribute to the project overall.

## Documentation

The palest ink is better than the best memory. **Chinese Proverb**

### Comments in Code

Comments should discuss *why* something is done, its purpose and its goal. The code already shows *how* it is done, so commenting on this is redundant—and is a violation of the *DRY* principle.

Commenting source code gives you the perfect opportunity to document those elusive bits of a project that can’t be documented anywhere else: engineering trade-offs, why decisions were made, what other alternatives were discarded, and so on.

We like to see a simple module-level header comment, comments for significant data and type declarations, and a brief per-class and per-method header, describing how the function is used, parameters and returned values and anything that it does that is not obvious.

With meaningful comments in place, tools such as JavaDoc [URL 7] and DOC++ [URL 21] can extract and format them to automatically produce API-level documentation. This is one specific example of a more general technique we use—executable documents.

### Executable Documents

Where possible, use your documentation as the source for creating your source code (for example when creating data base schema and queries). This will prevent DRY and help keep your code and documentation consistent.

It will ensure that your document is an integral part of the project development. The only way to change the schema is to change the document. You are guaranteeing that the specification, schema, and code all agree. You minimize the amount of work you have to do for each change, and you can update the views of the change automatically.

We can also generate API-level documentation from source code using tools such as JavaDoc and DOC++ in a similar fashion. The model is the source code: one view of the model can be compiled; other views are meant to be printed out or viewed on the Web. Our goal is always to work on the model—whether the model is the code itself or some other document—and have all views updated automatically

## Gently Exceed Your Users’ Expectations

The success of a project is measured by how well it meets the *expectations* of its users. A project that falls below their expectations is deemed a failure, no matter how good the deliverable is in absolute terms.

### Communicate and Manage Expectations

As your understanding of their needs develops, you’ll find areas where their expectations cannot be met, or where their expectations are perhaps too conservative. Part of your role is to communicate this. Work with your users so that their understanding of what you’ll be delivering is accurate. And do this throughout the development process. Never lose sight of the business problems your application is intended to solve.

### The Extra Mile

Try to surprise your users. Not scare them, mind you, but *delight* them.

Give them that little bit more than they were expecting. The extra bit of effort it requires to add some user-oriented feature to the system will pay for itself time and time again in goodwill.

Listen to your users as the project progresses for clues about what features would really delight them. Just remember not to over complicate or break the system adding these new features.

## Change Requests

Make sure everyone knows the cost of requirements changes, including time and cost of the product.

Set up a system where change requests can be submitted and reviewed at specific points. Don’t allow the requirements to keep changing on the go – separate the base requirements from the change requests so that you can control the scope and schedule of the project.

Agile programing, deliverable sprints and customer involvement can help identify changing requirements and accommodate change requests throughout the project.

# Programing Resources

## How to Learn Effectively

1. Plan exactly what you want to learn. In practical terms: no learn language x but instead: I want to be able to build a specific x function/program with this language/technology.
2. If you want to learn fast - don’t learn more than one thing at a time. Don’t learn a new domain while learning a new language and vice versa.
3. Divide into small bites:
   1. I want to display ‘Hello World’ screen
   2. I want to add a button that does x
   3. And so on.

* Don’t read books cover to cover – read only the parts that are relevant to the task at hand and ignore the rest.
* Learn by solving problems not by passively reading/watching tutorials. Constantly seek new challenges for your current level of experience.

## Refactoring

<https://www.youtube.com/watch?v=0rsilnpU1DU&t=13s>

### Explicit Instead of Implicit

Try to make anything that is implicit in your code such as anything that you need to have additional knowledge to understand or something that you’ll need to ask a team mate and try making it explicit.

### Extract Compound Conditions to Private Methods

Extract compound conditions and create a functions from them.  
This make the code more readable and makes explicit what was previous explicit. Example:  
Before:

**Before**:

**if** user\_created\_account\_today? && user\_has\_unconfirmed\_email?

prevent\_user\_from\_posting

**end**

**After**:

**if** user\_has\_high\_spam\_risk?

prevent\_user\_from\_posting

**end**

private

**def** **user\_has\_high\_spam\_risk?**

user\_created\_account\_today? && user\_has\_unconfirmed\_email?

**end**

2. Policy object:  
If my conditions become too complicated and extracting the function is not simple enough, create a policy object that define the

## Coding Quality Challenge

## Books

### Rails

* Michael Hartl's Rails tutorial (<https://www.railstutorial.org/>) is a great example of this. It'll run you through building a twitter clone and introduce you to git, heroku, a bit of CSS/HTML, and even goes into AJAX a bit. I can't recommend it enough to people looking to get into rails.

### Python

#### DJango

Django Unleashed is written as a step-by-step guide to building a single Django project. The link below has links to Amazon or to Pearson's own site (DRM-Free!).

<http://django-unleashed.com/>

Full Disclosure: I'm the author.

If you're looking to go nuts, I recommend reading the first twelve chapters of Django Unleashed, following it up with Harry Percival's Test Driven Development with Python book (also written as a step-by-step guide), and then finishing Unleashed.

[http://chimera.labs.oreilly.com/books/1234000000754/index.ht...](http://chimera.labs.oreilly.com/books/1234000000754/index.html)

Two Scoops of Django (not a guide) is great to have after that.

[https://www.twoscoopspress.com/products/two-scoops-of-django...](https://www.twoscoopspress.com/products/two-scoops-of-django-1-8)

Hope that helps!

#### Flask

Miguel Grinberg's "Flask Web Development" [1] is an excellent introduction Python-based web development. You build a Twitter-clone. The book is an adaptation of the authors 18-part tutorial on the same topic [2].

[1] <http://shop.oreilly.com/product/0636920031116.do>

[2] [https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial...](https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world)

### Linux

|  |  |
| --- | --- |
| |  | | --- | | Linux From Scratch: <http://www.linuxfromscratch.org/lfs/>  Goes through building a Linux system from the ground up, and gives a pretty thorough overview of why everything is working the way that it is.  [reply](https://news.ycombinator.com/reply?id=13662155&goto=item%3Fid%3D13660086%2313662155) | |
| |  | | --- | | [captainlego](https://news.ycombinator.com/user?id=captainlego) [2 days ago](https://news.ycombinator.com/item?id=13665004) [[-]](javascript:void(0))  Any idea about how long this would take from start to finish for someone who has a bit of experience using Linux and higher-level programming languages?  [reply](https://news.ycombinator.com/reply?id=13665004&goto=item%3Fid%3D13660086%2313665004) | |
| |  | | --- | | [elorm](https://news.ycombinator.com/user?id=elorm) [1 day ago](https://news.ycombinator.com/item?id=13673621) [[-]](javascript:void(0))  If you're just doing basic LFS packages and don't come up on any hitches, You should be done in 24 - 48 hours  [reply](https://news.ycombinator.com/reply?id=13673621&goto=item%3Fid%3D13660086%2313673621) | |
| |  | | --- | | [jajern](https://news.ycombinator.com/user?id=jajern) [2 days ago](https://news.ycombinator.com/item?id=13664254) [[-]](javascript:void(0))  I did this a couple years ago. It was pretty enjoyable and I learned quite a bit about Linux that I still apply almost every day.  [reply](https://news.ycombinator.com/reply?id=13664254&goto=item%3Fid%3D13660086%2313664254) | |

### Java Script

|  |  |
| --- | --- |
| |  | | --- | | Eloquent Javascript - <http://eloquentjavascript.net/>  You build 5 projects through the book - a programming language, paint program, a dom game and a skill sharing website using node js.  [reply](https://news.ycombinator.com/reply?id=13661145&goto=item%3Fid%3D13660086%2313661145) | |
| |  |  |  | | --- | --- | --- | | https://news.ycombinator.com/s.gif |  | [gingerbread-man](https://news.ycombinator.com/user?id=gingerbread-man) [2 days ago](https://news.ycombinator.com/item?id=13665754) [[-]](javascript:void(0))  For anyone who hasn't read this book, it's great, but very much an intro-to-programming book, at least for the first 6-or-so chapters.  [reply](https://news.ycombinator.com/reply?id=13665754&goto=item%3Fid%3D13660086%2313665754) | |
| |  |  |  | | --- | --- | --- | | https://news.ycombinator.com/s.gif |  | [imdsm](https://news.ycombinator.com/user?id=imdsm) [3 days ago](https://news.ycombinator.com/item?id=13662105) [[-]](javascript:void(0))  Added bonus: beautiful typography.  [reply](https://news.ycombinator.com/reply?id=13662105&goto=item%3Fid%3D13660086%2313662105) | |

### Inform 7 – Text Adventure Language

|  |  |
| --- | --- |
| |  | | --- | | Aaron Reed has a terrific book, "Creating Interactive Fiction with Inform 7" ([http://inform7.textories.com](http://inform7.textories.com/)) that walks you through building a complete (and fairly sophisticated) text adventure.  Inform 7 is very much a niche programming language, but it's really interesting and unusual, well worth investigating if you want to broaden your horizons. Vaguely Prolog-like, but written in natural language.  [reply](https://news.ycombinator.com/reply?id=13661372&goto=item%3Fid%3D13660086%2313661372) | |
| |  |  |  | | --- | --- | --- | | https://news.ycombinator.com/s.gif |  | [throwaway7645](https://news.ycombinator.com/user?id=throwaway7645) [3 days ago](https://news.ycombinator.com/item?id=13661914) [[-]](javascript:void(0))  I've played many games created with Inform7. All beautiful works of art. Emily short has a few on her site. Just download something like Gargoyle to play the image.  [reply](https://news.ycombinator.com/reply?id=13661914&goto=item%3Fid%3D13660086%2313661914) | |

### Game Development

|  |  |
| --- | --- |
| |  | | --- | | Two decent ones for game development:  <https://howtomakeanrpg.com/>  Walks through building a JRPG-style game with Lua. Pretty impressive for the price, especially with all the royalty-free assets that are included.  [https://books.google.com/books/about/Programming\_an\_RTS\_Game...](https://books.google.com/books/about/Programming_an_RTS_Game_with_Direct3D.html?id=jVkmAQAACAAJ)  A little dated now, (although DirectX 9 isn't quite dead yet), but this one has some pretty interesting topics. Good chapter on procedural terrain generation, some basic pathfinding, minimaps and fog-of-war.  [reply](https://news.ycombinator.com/reply?id=13661536&goto=item%3Fid%3D13660086%2313661536) | |
| |  |  |  | | --- | --- | --- | | https://news.ycombinator.com/s.gif |  | [makufiru](https://news.ycombinator.com/user?id=makufiru) [2 days ago](https://news.ycombinator.com/item?id=13669145) [[-]](javascript:void(0))  Thanks for sharing! I'm just getting into games and these look really helpful.  [reply](https://news.ycombinator.com/reply?id=13669145&goto=item%3Fid%3D13660086%2313669145) | |
| |  |  |  | | --- | --- | --- | | https://news.ycombinator.com/s.gif |  | [throwaway7645](https://news.ycombinator.com/user?id=throwaway7645) [3 days ago](https://news.ycombinator.com/item?id=13662066) [[-]](javascript:void(0))  Lua and RPGs? Bookmarked! | |

### C++

|  |  |
| --- | --- |
| |  | | --- | | I remember seeing a C++ (Borland) book that was entirely about building a flight simulator. Google is betraying my attempts to find it though, and i'm not sure who the actual publisher was. That was during the 90's when I was 14 or something so it was a bit above my head, but I remember almost buying it anyway.  Edit: It may have been this: [https://www.amazon.com/Build-Your-Own-Flight-Sim/dp/15716902...](https://www.amazon.com/Build-Your-Own-Flight-Sim/dp/1571690220)  [reply](https://news.ycombinator.com/reply?id=13661729&goto=item%3Fid%3D13660086%2313661729) | |
| |  |  |  | | --- | --- | --- | | https://news.ycombinator.com/s.gif |  | [sigstoat](https://news.ycombinator.com/user?id=sigstoat) [3 days ago](https://news.ycombinator.com/item?id=13662524) [[-]](javascript:void(0))  Flights of Fantasy  [https://www.amazon.com/Flights-Fantasy-Programming-Video-Gam...](https://www.amazon.com/Flights-Fantasy-Programming-Video-Games/dp/1878739182/)  [reply](https://news.ycombinator.com/reply?id=13662524&goto=item%3Fid%3D13660086%2313662524) | |

### Java

 Each chapter builds on the last by adding something new to the same project

I bet this is what you are looking for:

[https://www.amazon.com/Compiler-Construction-Using-Java-Java...](https://www.amazon.com/Compiler-Construction-Using-Java-JavaCC/dp/0470949597)

This book taught me how to write a compiler.

Here is its description from its website:

\* Comprehensive treatment of compiler construction.

\* JavaCC and Yacc coverage optional.

\* Entire book is Java oriented.

\* Powerful software package available to students that tests and evaluates their compilers.

\* Fully defines many projects so students can learn how to put the theory into practice.

\* Includes supplements on theory so that the book can be used in a course that combines compiler construction with formal languages, automata theory, and computability theory.

If you already know C or C++ or Java then this book is for you. In my opinion, you can learn many computer science concepts and be able to apply to your field. The book will teach you how to write a grammar then write a parser from it then eventually be able to improve it as you go on reading and doing the exercises. It was a great moment when I feel comfortable writing recursive functions since grammars are composed of recursive functions. You'll also learn a nice way on how you can get your compiler to generate assembly code. Another feature of the book is the chapter on Finite Automata wherein you'll learn how to convert between regular expressions, regular grammars and finite automata and eventually write your own 'grep' which was for me is a mind-blowing experience. There are lots of other stuffs in this book that you could learn. Thank you Anthony J. Dos Reis for writing great books for people like me.

### Calculus

This is a slightly different response from others, but I think fits the intent of the question: The Better Explained Guide to Calculus: <https://betterexplained.com/calculus/>

Kalid basically iterates the series around the concept of deriving the formula for the area/perimeter of a circle, and then builds up to deriving the surface area/volume of a sphere. The focus throughout is the building up of an intuition of calculus before leaping into formulas. Even with uni-level calculus, I did strengthen my intuition of what's going on by reading through his book.

It's pretty fun, and I actually spent some time visualizing the calculus of geometric solids afterward i.e <http://www.trinco.io/blog/derivative-of-x3>

## Programing Challenges/Katas:

**Recommended:**

https://www.topcoder.com/

\* https://codility.com/programmers/lessons/1-iterations/

\* Cracking the coding interview

\* Programing pearls

**Additional resources:**

<https://projecteuler.net/>

<https://github.com/garora/TDD-Katas>

<http://www.codekatas.org/>

Ruby Quiz: <http://rubyquiz.com/>

# Soft Skills

## How to criticize

Your message should be: "I support what you're doing. I want you to succeed. You've really disappointed me though, because you're better than this. I’m not an attacker but an ally"

* Don’t say ‘you’re this and that’. Say what you’re feeling etc.
* Establish ‘why should I listen to you’? Make sure that you have the knowledge/authority to critic and that you come across as a supporter and not a hater.
* Offer the criticism in a nice way
* Re-iterate that you’re a supporter and not an attacker

## How to Respond to Criticism

* Acknowledge the other side’s feeling / viewpoint / effort etc.
* Rather than finding the point of conflict and hammering the other person with the "facts" of why you're right and they are wrong, look for a point where you both agree.  
    
  Call out that agreement—then start to build your case from there.  
    
  Often you'll find this will cut the tension and allow you to make your case more effectively than if you'd come out guns blazing.