How to Make Software

An introduction to software development

by Nick Jenkins

Table of Contents

Introduction

Who should read this book ?

Programmers, Developers, Coders, Hackers, Crackers and Joe Public.

Process is not a dirty word

To most of us process is an anathema. There's something about the creative, independent and egotistical nature of software development that breeds a deep seated loathing for established process. If someone suggests that we need to do things in a defined order we usually roll our eyes and moan out loud. Maybe it's a creative thing… or maybe it's just a human thing.

In either case I'd like to put forward the contentious suggestion that process is not a dirty word.

Process is necessary to help us do things in a repeatable and organised fashion. It's only by establishing processes that we can produce predictable results, it's what differentiates software engineering from hacking. Many programmers I talk to agree with me, yet when I propose imposing the most meagre of processes I'm met with howls of derisive laughter. "We tried that before at company XYZ," they say, "it just didn't work, can we go back to our code now ?"

Open source advocates in particular seem to have a particular disdain for anything that smacks of formalised process. Mind you most commercial software development organisations are no better. Commercial organisations tend to have a an amount of process or methodology that ranges from shaky-word-of-mouth to lengthy, fascist dogma detailed in twenty seven volumes. The observance of process can be lip-service only however and at the grass roots people keep hacking away at the code to get things done. The best processes are the ones that are organic to the work involved.

What I am trying to say is that process do exist, that do work and that help. Generally they are simple, easy to use and understand and assist rather than impede your work. This book is an attempt to describe some simple processes that worked for me and I hope will work for you.

Keep an open mind.

In the real world, as you work to design and implement software, you have several concerns to keep in mind -- several "monkeys on your back."

Each monkey competes with the others for your attention, trying to convince you to take its particular concern to heart as you work. One large, heavy monkey hangs on your back with its arms around your neck and repeatedly yells, "You must meet the schedule!"

Another monkey, this one perched on top of your head (as there is no more room on your back), beats its chest and cries, "You must accurately implement the specification!" Still another monkey jumps up and down on top of your monitor yelling, "Robustness, robustness, robustness!" Another keeps trying to scramble up your leg crying, "Don't forget about performance!"

And every now and then, a small monkey peeks timidly at you from beneath the keyboard. When this happens, the other monkeys become silent. The little monkey slowly emerges from under the keyboard, stands up, looks you in the eye, and says, "You must make the code easy to read and easy to change."

With this, all the other monkeys scream and jump onto the little monkey, forcing it back under the keyboard. With the little monkey out of site, the other monkeys return to their customary positions and resume their customary activities.

From Interface Design by Bill Venners - http://www.artima.com/interfacedesign/introduction.html

Ten Axioms for Success

Writing software is not as complicated as it seems and many projects fail because of the simplest of causes. You don't have to be a genius to deliver software on time, nor do you have to be steeped in the folklore of a complicated methodology to join the 'elite' of the software development world.

If an averagely competent programmer can't deliver software successfully after reading this book then I will run buck naked through Times Square on my 75th birthday. See if I don't.

To help you get started here's ten (self evident) truths :

I. Know your goal

It sounds obvious but if you don't have an end-point in mind, you'll never get there. You must be able to clearly describe the final state of your software so that anyone can understand it. If you can't describe it in one sentence then your chances of achieving it are pretty slim.

II. Know your team

Are you working alone or with other people ? If you're collaborating with other people or working in a team, know everyone's strengths and weaknesses, know what makes them tick and what motivates them. Invest some time ensuring that everyone knows what they have to contribute to the bigger picture. Dish out reward as well as criticism, provide superior working conditions and lead by example.

III. Know your stakeholders

Stakeholders, customers, clients, end-users, call them what you will – they are the ultimate beneficiaries of your work. Spend some time with them and get to know what they want and how they want it. Shake hands and kiss babies as necessary and grease the wheels of the bureaucratic machine so that your software has the smoothest ride possible.

IV. Spend time on planning and design

One big mistake traditionally committed by developers is to leap into programming before they are ready. When you're under pressure to deliver, the temptation is to 'get the ball rolling'. The ball however, is big and heavy and it's very, very difficult to change its direction once it gets moving. Spend some time thinking about how you're going to solve your problem in the most efficient and elegant way. Try some things out and then go back and think about it again. Up to 40% of the total project time should be spent on design and planning!

V. Promise low and deliver high

Try and deliver happy surprises and not unpleasant ones. By promising low (understating your goals) and delivering high (delivering more than your promised) you :

- Build confidence in yourself, the software and the team
- Buy yourself contingency in the event that things go wrong
- Generate a positive and receptive atmosphere

Consider this : if you follow the above advice and finish early everyone will be happy; if something goes wrong you might still finish on time and everyone will still be happy; if things goes really badly you might still not deliver what you anticipated but it will still be better than if you over-promised!

VI. Iterate! Increment! Evolve!

Most problems worth solving are too big to swallow in one lump. Any serious software will require some kind of decomposition of the problem in order to solve it.

You can try and do this all at the start of the project or you can break it down into phases and tackle it part-by-part. The first just doesn't work. The second works but only with close attention to how each piece is analysed and resolved and how the whole fits together. Without a systematic approach you end up with a hundred different solutions instead of one big one.

The simplest way to do this is through a cycle of Design-Develop-Evaluate. Take a set of requirements and *design* a solution. *Develop* that solution into a working example and *evaluate* it against your requirements. Repeat until that part is finished. Repeat with each part until the whole project is finished.

VII. Stay on track

Presumably you are writing software with an end goal in mind. Maybe it's your job and your salary is your end goal, maybe your business depends upon it or maybe you're going to revolutionise the world with the next Google, the next World Wide Web or the next Siebel/SAP/Oracle.

If this is the case you need to maintain control of the project and track your progress towards a goal. If you don't do this then you're a hobby programmer or hacker at best.

VIII. Cope with change

We live in a changing world. As your project progresses the temptation to deviate from the plan will become irresistible. Stakeholders will come up with new and 'interesting' ideas, your team will bolt down all kinds of rat holes and your carefully crafted ideas will have all the solidity and permanence of a snowflake in quicksand.

This doesn't imply that there should be single, immutable design which is written down for all time and all other ideas must be stifled. You need to build a flexible approach that allows you to accommodate changes as they arise.

It's a happy medium you're striving for - if you are too flexible your project will meander like a horse without a rider and if you are too rigid your project will shatter like a pane of glass the first time a stakeholder tosses you a new requirement.

IX. Test Early, Test Often

Making software is a creative discipline loaded with assumptions and mistakes. The only way to eliminate errors is through testing. Sure you can do a lot of valuable work to prevent these mistakes being introduced to the code but, to err is human, and some of those errors <u>will</u> make it into the code. Testing is the only way to find and eliminate errors.

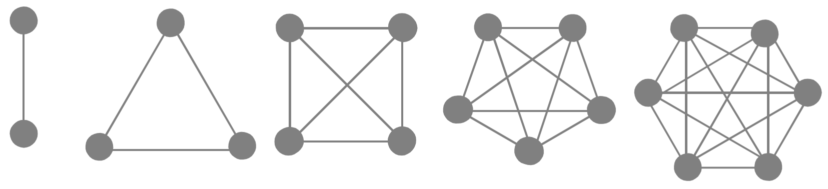
X. Keep an open mind!

Be flexible! The essential outcome is delivery of the finished software to a customer who is happy with the result. <u>Any</u> means necessary can be used to achieve this and every rule listed above can be broken in the right circumstances, for the right reasons. Don't get locked into an ideology or a particular tool if the circumstances dictate otherwise.

The Mythical Man Month

In 1975 during the pioneering days of software development a man named Frederick Brooks penned a number of articles. His most famous is "No Silver Bullet", in which he pointed out that software development could expect no thunderbolt solution to its various problems of quality, cost and complexity other than to adopt rigorous methodology. Only slightly less famous than "No Silver Bullet" is another Brook's paper, "The Mythical Man Month".

In "The Mythical Man Month" Brooks argues that adding people to a development doesn't speed it up. While it is true that more resources *can* speed up the delivery of software, the increase is not directly proportional to the amount of resource added. To put it another way, you can't just throw people at a development project and expect to finish earlier. The reason for this is the increased complexity of communications which results from adding more people. As each person is added to the project team the complexity of communications goes up exponentially.



People	2	3	4	6	6	(n)
Interfaces	1	3	6	10	15	$\frac{n^2-n}{2}$

The diagram above demonstrates the principle graphically. The more interfaces you add, the more complexity you add and the more communications overhead you add. Every additional person brought into the development cycle will need to be trained and briefed and assigned tasks. As more and more people are added, more of the original team must be devoted to managing the overall structure. This is a truism of all types of management, not just software development.

Yet, while this might seem obvious, the same mistake is committed time and time again. The first reaction to any 'slow-down' in the schedule is to throw more people at it. This rarely works in a well-controlled project and <u>never</u> in a badly controlled project.

There are a few things to learn from this :

1. Small autonomous teams are more efficient than large bureaucratic ones - divide your software up into manageable chunks and let a group work on each of them.
2. If you want to add people to a project, you had better plan carefully plan how those people are introduced into the team, there will be a lag before they become productive and they might even be a drain on the productivity of other members of the team.
3. You have other options than adding people – one is to cut functionality. While it hurts it has a number of benefits: first, it's the only thing that will <u>reduce</u> the complexity of what you have deliver; second it is finite and easily understood; and third it has a knock on effect as it reduces effort required in things like testing and deployment.

One particular project I was involved with illustrated to me the truth behind the "mythical man month" more than any other. I was a consultant testing a high profile software project on behalf of a major bank. The developer, one of the world's largest IT companies had flown in a team from the other side of the world and as the deadlines slipped they started flying in more and more people to 'help'.

It was chaos. Developers were sitting around waiting for instructions. Graphic designers were busily designing interfaces for screens whose business logic hadn't even been finalised. There were at least three different versions of the specs floating around and no one knew which one was current. We had a field day! Every release was turned back with major bug and as far as I know the system never went live even after nine months and several million dollars.

The Life of Software

My favourite quote on software, from Bruce Sterling's "The Hacker Crackdown"

> The stuff we call "software" is not like anything that human society is used to thinking about. Software is something like a machine, and something like mathematics, and something like language, and something like thought, and art, and information.... but software is not in fact any of those other things. The protean quality of software is one of the great sources of its fascination. It also makes software very powerful, very subtle, very unpredictable, and very risky.

What's a Model ?

A model is a communications tool.

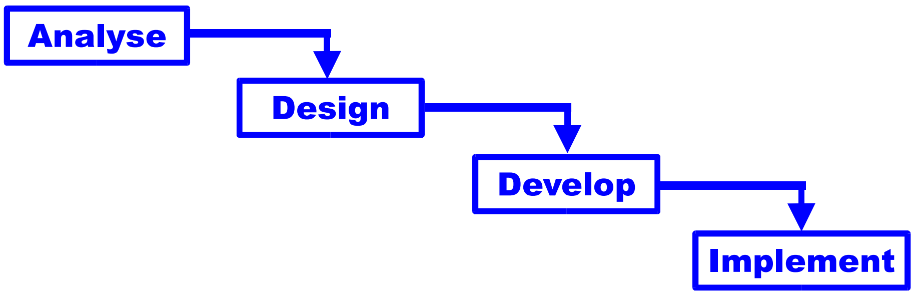
I use a model here to describe the process of software development so that you can understand it better. If everyone on your team uses the same model then they start with the same conceptual understanding. This helps with communications because then everyone knows what you mean when you talk about the "design phase" or the "implementation phase".

Traditional models have often had the drawback of being overly complicated or convoluted to understand and explain. While making software is a complex discipline there is no reason why the concepts behind it shouldn't be easily explained. To this end the model I present here is a simple one which allows people in day-to-day roles to apply these ideas in a practical way.

Ancient History – The Waterfall Model

Making something can be thought of as a linear sequence of events. You start at A, you do B and then go to C and eventually end up at Z. This is neither particularly accurate nor particularly realistic but it does allow you to visualise the series of events in the simplest way. It also emphasises the importance of delivery with steps being taken towards a conclusion.

Below is the "Waterfall Model" which shows typical development tasks flowing into each other. Early in the history of software development it was adapted from engineering models to be a blue print for software development.



The four steps outlined are :

- *Analyse* the requirements of the project and decide what it is supposed to do;
- *Design* a solution to meet these requirements;
- *Develop* the design into a working product;
- *Implement* the finished product with end-users.

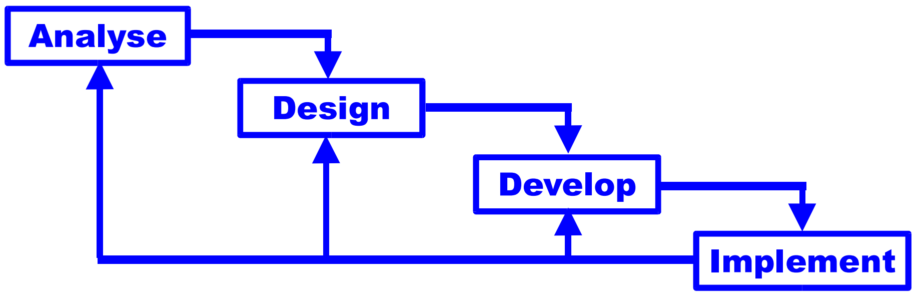
The Waterfall Model was widely adopted in the early days of software development and a lot of blame has been laid at its door. The model was derived from engineering models and while it does prove a useful point it is intrinsically flawed.

It is very rare that requirements analysis can be entirely completed before design and design before development and so on. In a small project this is not a problem since the span from "analyse" to "implement" may be a period of weeks or even days. For a large scale project which span months or even years the gap becomes significant. The more time that passes between analysis and implementation, the more a gap exists between the delivered project and the requirements of end-users.

Think about a banking or finance system which is 'analysed' one year, designed the next year and developed and implemented the following year. That's three years between the point at which the requirements of the system are captured and the system actually reaches its end users. In three years its likely that the business, if not the whole industry, will have moved on considerably and the requirements will no longer be valid. The developers will be developing the wrong system! Software of this scale is not uncommon either.

A definition of requirements may be accurate at the time of capture but decay with frightening speed. In the modern business world, the chance of your requirements analysis being valid a couple of months after it has been conducted is very slim indeed. You need to move much faster than that.

The shortcomings of this model became apparent and other versions of the waterfall model have been developed. One, the Iterative Waterfall Model, includes a loop as shown below:

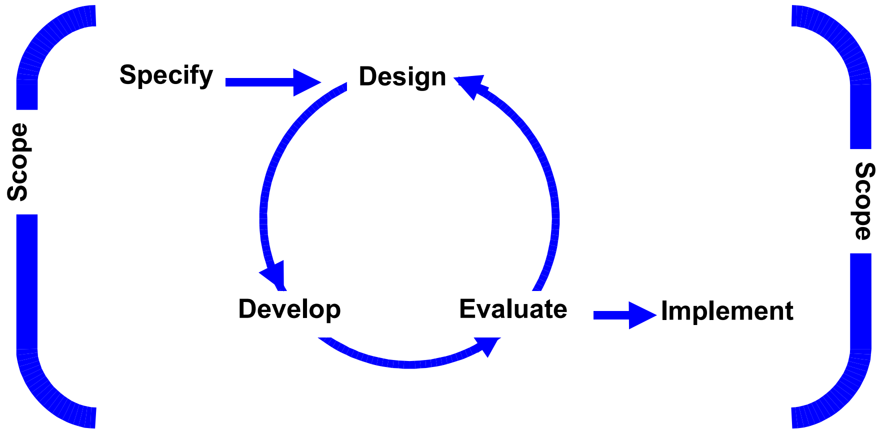


This model attempts to overcome the limitations of the original model by adding an "iterative" loop to the end of the cycle. That is, in order to keep up with changing requirements the "analysis" phase is revisited at the end of the cycle and the process starts over again.

This alleviates the situation somewhat but still introduces a considerable lag between analysis and implementation. The waterfall model implies you have to complete all the steps before you start the process again. If requirements change during the life of the project the waterfall model requires the completion of a full cycle before they can be revisited.

Iterative Software Development

I wanted to do this section without using a diagram, but a picture is worth a thousand words...



The model above shows the five phases in the development life-cycle bounded by the *Scope* :

- *Specify* the requirements of stakeholders
    - *Design* an iteration of your software to meet those requirements
    - *Develop* the iteration to deliver the design
    - *Evaluate* the delivered iteration to measure your success

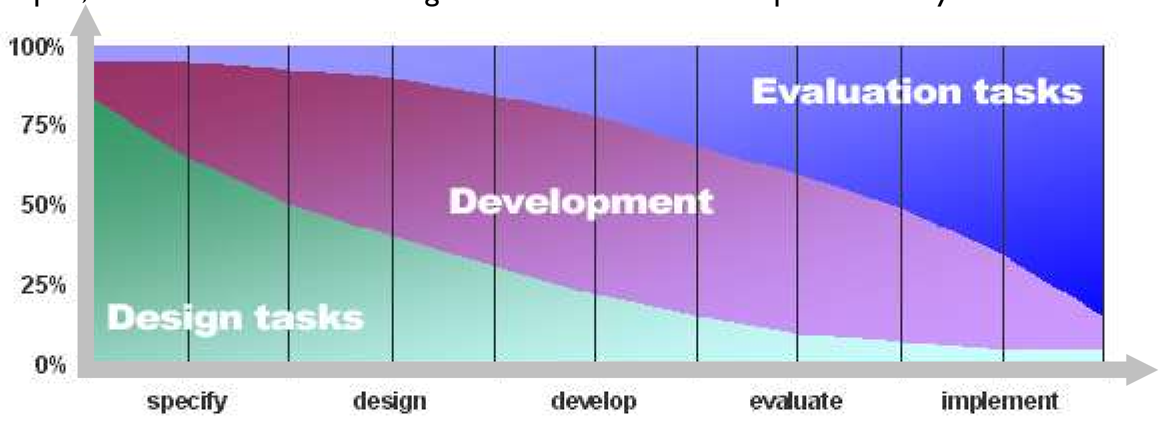
... and repeat the cycle of design, develop and evaluate until the software is finished and...

- *Implement* your system by delivering the final product to the end-users

The emphasis in this model is on fast iterations through the cycle. Prototypes are designed, developed and evaluated with users, involving them in the process. The model is particularly suited to projects in rapidly changing environments where the team needs to adapt to different situations.

Incremental Development

Note that not all iterations need be complete, fully functional software, nor should they necessarily focus on the same areas of functionality. It is in fact desirable to deliver separate 'increments' of functionality and assemble the whole project only at the conclusion of the production phase. This way you can take partial steps towards your completed goal. Each unit can be individually developed, tested and then bolted together to form the overall product or system.



The diagram above indicates progress through the development life-cycle in an iterative / incremental development. Early on the iterations focus on 'design tasks' and the emphasis is on making design decisions and prototypes. As the project progresses tasks shift to development where the bulk of the coding is done. Finally, the emphasis is on testing or evaluation to ensure that what has been developed meets requirements and is solid, stable and bug free (ha!ha!).

Failure to nail down exactly what it is that you hope to achieve is a major source of failure in software development. Requirements specification is the process of refining the goals of a project to decide what must be achieved to satisfy the clients.

Sometimes requirements specification takes place before the formal commencement of a project to identify and select the right for a solution and technology. Often this is known as a feasibility study or project analysis. More typically however some requirements are thrown together (maybe in a proposal) and the real requirements specification occurs only after the project has started.

A good example of the importance of requirements capture came from a friend of mine.

He worked in the IT department of a large commercial law firm. His department head had initiated a project which took eighteen months to complete and was a master stroke of technical genius. The system allowed automated searching of 'precedence' documents, a laborious manual task that took up many hours of legal assistants time. When a court case was being prepared they were forced to sort through stack after stack of mouldering documents to see if there was pertinent ruling that might apply to the new case. The new system could do this in a matter of minutes.

The system was launched with much fanfare and fell flat on its face some two months later. Why ? It turns out that a substantial portion of the income generated in each case came from the billable hours attributable to the legal assistants time spent in the archives. All the time they spent riffling through files was directly billable to the customer in increments of 6 minutes, known as ticks. The partners of the law firm so no good reason why they should have all those capable, money generating legal assistants, sitting around on their hands when they could be out making money!

The broader business community might have taken a different view, but the partners ran the show and the project flopped because no one had asked them!

Functional Requirements

Functional requirements are the obvious day-to-day requirements end-users and stakeholders will have for the system. They revolve around the functionality that must be present in the project for it to be of use to them.

A functional requirement typically states as "the system X must perform function Y". This is known as an 'assertion'. An assertion asserts or affirms a necessary or desirable behaviour for the system or product in the eyes of a stakeholder.

Without clear assertions requirements are nothing more than vague discussions which have a regrettable tendency to clutter up your desk and your mind.

Compare the two following, contrasting, functional requirements:

- *The financial system must produce a detailed customer invoice as per Appendix A.*
- *Producing an invoice for customers is important. Invoices should contain all the pertinent information necessary to enable a customer to supply payment for goods.*

The first is a functional requirement stated as an assertion. It indicates that the *financial system is* responsible for producing a *detailed customer invoice* which contains all the information in *Appendix A.* While it could be more specific, the reader is left in no doubt as to what the financial system *must* do in order to be a successful financial system.

The second could be the introduction for a chapter in an accounting book. Although it states that invoices are important it gives no indication of who or what is responsible for producing them. It then rambles on about what goes in an invoice which everyone already knows anyway. Such a statement does not belong in a requirements specification.

> The second 'requirement' compounds the problem by looking solid but really being vague and ambiguous. What does *pertinent information* mean? *To enable a customer to supply payment* is superfluous since that's what an invoice is for. The statement, while accurate, contributes nothing towards our understanding of what the system must do to be successful.

Here are some more 'better' statements of requirements:

- *A customer account record must contain a unique account reference, a primary contact name, contact details and a list of all sales to the customer within the past sales year*
- *Contact details must consist of a phone number, an address and an optional email address*
- *For each contact up to five separate phone numbers should be catered for*

Non-Functional Requirements

Most requirements capture will focus upon functional requirements, but it is essential to consider the non-functional requirements too : .

Performance	Are their any minimum performance levels the software must reach in order to satisfy the customer?. Performance usually covers areas such as responsiveness, processing times and speed of operation.
Usability	How "easy-to-use" will the finished product be ? For example do you cater for disabled or handicapped users ? Generic ease of use should be considered though, more than one product has failed by supplying full functionality with an obscure or convoluted interface.
Reliability	Reliability requirements deal with the *continuous availability* of the product to users. They should state what availability is necessary and desirable.
Security	In software which deals with confidential or sensitive information, security considerations should be taken into account. Requirements for different levels of access, encryption and protection should be gathered.
Financial	There may be financial considerations which will determine the success or failure of the project. For example a bank or investor might specify certain financial constraints or covenants which must be satisfied during the project.
Legal	There may be legal requirements that must be met due to the environment in which your software will operate. Consult a legal expert for these.
Operational	There may be a number of day-to-day operational issues that need to be considered. Failure to accommodate these will not delay project launch but may limit or halt its uptake by end-users once it has been launched.

Stakeholders

Stakeholders are an integral part of a project. They are the end-users or clients, the people from whom requirements will be drawn, the people who will influence the design and, ultimately, the people who will reap the benefits of your completed project.

It is extremely important to involve stakeholders in all phases of your project for two reasons: Firstly, experience shows that their involvement in the project significantly increases your chances of success by building in a self-correcting feedback loop; Secondly, involving them in your project builds confidence in your product and will greatly ease its acceptance in your target audience.

There are different types of stakeholders and each type should be handled differently :

Executive	Executive stakeholders are the guys who pay the bills. Typically they are managers or directors who are involved with commercial objectives for the project. They should restrict themselves to commercial considerations and be actively discouraged from being involved in technical design, their experience and skills are vastly different to that of 'typical' end-users.
End-user	These are the guys that are going to use your product. No one knows more about what the product is supposed to do when it hits their desks than they do. No one ! Including you ! You may think you know better but if you don't listen to them you're kidding yourself.
Expert	Sometimes you need input from experts in other fields. People like graphic designers, support reps, sales or sometime lawyers and accountants.

Requirements capture

Requirements capture is the process of harvesting the raw requirements of your stakeholders and turning them into something useful. It is essentially the interrogation of stakeholders to determine their needs. This can take many forms, with questionnaires or interviews being the most popular. The usual output is a 'requirements specification' document which details which of the stakeholder requirements the project will address and, importantly, which it will not.

The focus in requirements capture must be in gathering of information. Keep your ears open and your mouth shut! Listen carefully to what people want before you start designing your product. Later you can focus on *how* things are to be achieved for now you need to find out *what* must be achieved. (Design decisions involve making assumptions, this can result in 'leading' the stakeholders and delivering a product that you want and not one that they want).

In reality this is difficult to achieve. Technical people complain that stakeholders often "don't know what they want". This is not true. Stakeholders know exactly what they want – they want less hassle, easier jobs and so on. The problem is that they can't design the system for you, they can't tell you how to achieve what they want. The trick in requirements specification is to take what the stakeholders give you and distil it into something you can use to help you make decisions on how to implement their wishes. One way to think of this is as finding the ideal solution to the stakeholder's current problems.

Requirements capture also needs to be fast. Development has a tendency to bog down at this stage and to produce reams and reams of documentation, but no useful output. The aim of requirements capture is not to produce an endless tome detailing the answer to every possible question but to provide enough clarity for the project team so that the objectives are clear.

Questionnaires

Questionnaires are a typical way of gathering requirements from stakeholders. By using a standard set of questions the project team can collect some information on the everyone's needs. While questionnaires are efficient for rapidly gathering a large number of requirements their effectiveness can be limited since it is a one way process. If you don't ask the right questions you don't get the right answers. There is no way to seek clarification or resolve conflict. To resolve this, questionnaires are usually used in conjunction with other methods, such as interviews.

Interviews

The same questions posed in a questionnaire can be put across a table in an interview. The benefit of the interview is that it allows more exploration of topics and open ended discussion on the requirements for the project. It's a two way process.

Interviews can either be structured as single or group sessions. Particular stakeholders can be interviewed individually or a group session can be used thrash out ideas amongst a larger number of stakeholders (and hopefully obtain consensus). In a group session you can use a formal structure or a more open style where ideas are thrown, a brainstorming session. The best ideas that survive and can be adopted by the project team as part of the requirements.

The down-side to interviews is that it is time and people intensive. Not only must the interview be set up and conducted but minutes must be taken, distributed and reviewed, and it may be necessary to hold one or more follow-up meetings. Using questionnaires *and* interviews is a common and efficient practice; questionnaires can be distributed beforehand and an overview of the stakeholder's requirements collected. The interviews are then focussed and directed towards the clarification of those requirements.

User observation

Another method of requirements capture is direct end-user observation or evaluation.

The purpose of user observation is to capture requirements that the end-users may not be consciously aware of. For example individuals using a cheque processing system in a large finance system may conduct a number of manual steps outside of the system that could be automated. Because they don't regard these steps as being part of the system they will not mention them when questioned about the incumbent system. Only by direct observation will the development team become aware of the importance of these steps.

You can either use free form observation or you can take a typical group of end users are set a series of tasks and monitor their processing of these tasks. Notes are made on the way they conduct the tasks, any obstacles they encounter and you could even video tape it (if they agree).

Direct user observation is particularly powerful because, unlike the first two methods of requirements capture, it relies on observed fact and not upon opinions. It is however, the most resource intensive of the three techniques.

Conflicting Requirements

One or more requirements may contain elements that directly conflict with elements of other requirements. For example, a performance requirement may indicate that a core system must be updated in real time but the size and scope of the system (as defined by other requirements) may preclude this. Updating such a large system may not be possible in real time.

One of the most effective ways of resolving the conflict between requirements is to impose some form of prioritisation on the requirements. This allows the potential for negotiation since it provides a basis for assessing conflicting requirements. For example if, from the previous example, the requirement for real time updates was rated at a much higher priority than the inclusion of full customer data then a compromise could be reached. The main 'online' database could contain only the barest essential of customer details (allowing real time updating) and a separate 'archive' database could be established which contained customer histories.

Requirements are often heavily interlocked with other requirements and much of the time it seems your stakeholders have diametrically opposed points of view and cannot come to terms. If you pick apart any set of requirement you can come to some sort of compromise with both parties and achieve consensus. This can be a difficult and even emotional process.

The very best way I have seen to resolve conflicting requirements works like this :

1. The stakeholders draw up a list of their requirements
2. The list is organised in strict priority order with the most important at the top, down to the least important at the bottom.
3. The project team then looks at the schedule and draws a line through the list based upon what they believe they can deliver with the time/money available
4. The stakeholders assess the development position and can then re-prioritise their requirements if required or negotiate over the nature and importance of requirements
5. Once consensus has been achieved both sides sign-off and work starts

This is not a simple or easy way to achieve consensus however. Even the basic ordering of the list of requirements is no mean feat in a project of any size.

Documenting Requirements

You need a way of documenting your requirements for your project team. Stakeholders are also often asked to sign off their requirements as a confirmation of what they desire. Often this is where money starts to change hands.

Documenting requirements is much more than just the process of writing down the requirements as the user sees them. The requirements specification is an essential link in the total design of the whole project and attempts to give meaning to the overall goals of the project.

Whatever form of documentation is used, it should cover not only *what* decisions have been made but also *why* they have been made. Understanding the reasoning that was used to arrive at a decision is critical in avoiding repetition. For example, if a particular feature has been excluded because it is not feasible, then that fact needs to be recorded. If it is not, the project risks wasted work and repetition when a stakeholder later requests the feature be reinstated.

Documenting the decision process is also useful for stakeholder's because it allows them to better understand what to expect from the final product. A basic statement of requirements without any underlying discussion can be difficult for a layman or end-user to understand.

SMART requirements

One useful acronym for to remember is SMART :

Specific	A goal or requirement must be specific. It should be worded in definite terms that do not offer any ambiguity in interpretation. It should also be concise and avoid extraneous information.
Measurable	A requirement must have a measurable outcome, otherwise you will not be able to determine when you have delivered it.
Achievable	A requirement or task should be achievable, there is no point in setting requirements that cannot realistically be achieved.
Relevant	Requirements specifications often contain more information than is strictly necessary which complicates documentation. Be concise and coherent.
Testable	In order to be of value requirements must be testable. You must be able to prove that the requirement has been satisfied. Requirements which are not testable can leave the project in limbo with no proof of delivery.

The Language and Layout of Requirements Specifications

Requirements specs. often have a lot of information in them. Mainly because they have multiple audience. They are used by the project team to deliver the product, and they are used by stakeholders to verify what is being done. While I advocate including contextual information to illustrate why a particular decision was taken there needs to be some way of easily separating the discussion from the "assertions".

So I apply the following rules of thumb:

- For each requirement there should be an "assertion"; in essence, a decision
- Where assertions are documented they should consist of a single sentence which states what a product "must" or "should" do.
- "Must" indicates a necessary requirement and "should" indicates a "nice-to-have".
- There must be simple (visual) way to distinguish assertion from discussion

Below is an example with some discussion and the assertion highlighted with *italics*:

> **1.1 Usability** – usability of the system is seen as very important to adoption of the system. The client raised some concerns about the timescales required to implement usability testing but the project team as a whole supported extending the time line for the development phase to include usability testing..
>
> *The system must be simple and easy to use and must follow the standard UI style as laid out in the company design handbook.*

In this case the discussion is preserved for future reference but the requirement stands out distinctly from the body of the text. A project member reading the specification can skip through it and pull out the requirements quickly and easily. A manager reading through the document can do the same but also can review the context of the decision to understand how it came about.

Sometimes classification of requirements is made using the MoSCoW rules:

Must-haves are fundamental to the project's success

Should-haves are important, but the project's success does not rely on these

Could-haves can easily be left out without impacting on the project

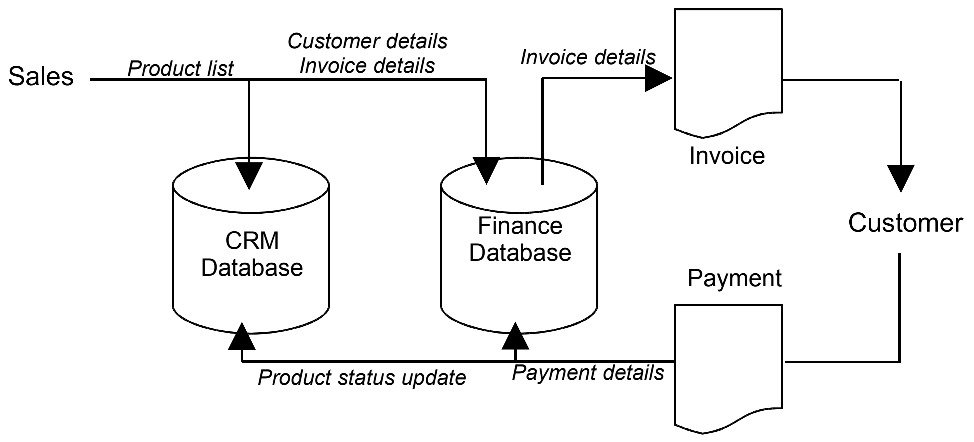
Won't-have-this-time-round can be left out this time and done at a later date

I feel that the distinction between "should have" and "could have" is never clear and is usually the subject of much debate between client and project manager so I normally omit "could have". Every requirement then either becomes necessary ("must have") or optional ("should have"). Stick to prioritising them.

Diagrammatic Methods

While the emphasis in this chapter has been on requirements specification in the form of text, other more graphical methods are available. As the saying goes, a picture is worth a thousand words, and this is both a blessing and a curse. While representing information with a picture or diagram can be extremely informative it can also be extremely confusing. Diagrams can often imply requirements without actually stating them and leave details open to interpretation.

For this reason I see graphical methods as supporting standard textual methods of description. They should be used wherever appropriate, where the graphical nature of a representation will more closely represent the nature of the requirement. If you find it difficult to explain in words the nature of a particular structure or process then by all means insert an appropriate diagram.



A Sample Requirements Specification

The most common method is to break down the requirements in an outline fashion :

1. **Current product status** – a clear and public indicator of the current status of a product
2. **Dates** - dates for each major milestone were also recognised as necessary. Although some of these dates will remain in the public domain others will be available only to "private" users. Private users will have the ability to publicise dates as they see fit.

   The dates specified are:

   2.1 Development sign off

   2.2 Testing sign off...

Even more structure can be put into the document by splitting up requirements categories :

1. Functional Requirements

1.1. **Product list** – *the system should produce a list of products available or under development*

1.2. **Current product status** – all parties highlighted the need for a clear and public indicator of the current status of a product. *There should be a simple flag which indicates at-a-glance whether the product is ready for release.*

1.3. **Dates** – *for each product, dates for each major milestone must be shown.* Although some of these dates will remain in the public domain others will be available only to "private" users. *Private users should have the ability to publicise dates as they see fit.*

The relevant dates are listed below:

1.3.1. Design sign off

1.3.2. Development sign off

1.3.3. Testing sign off...

2. Non-Functional Requirements

2.1. **Performance** – *the system must be updated daily and information available to all international users within 1min of the information being posted by head office.*

2.2. **Usability** – etc etc...

Modelling Workflows

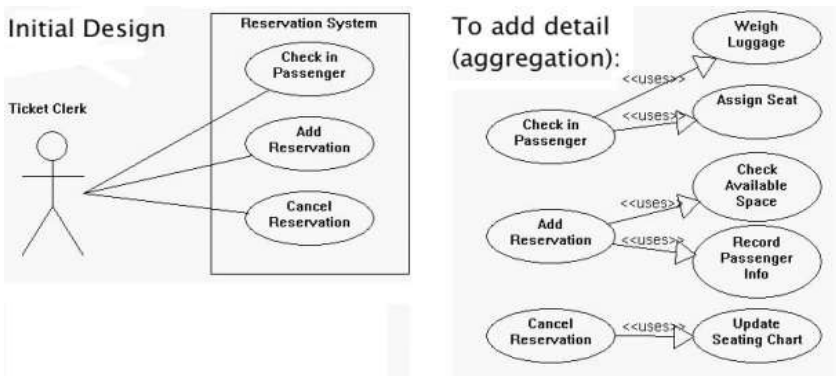
There are many and varied ways to model workflows or systems but the premier candidate at the moment has to be the Unified Modelling Language or UML. UML arose out of three separate object-oriented development methodologies developed by Grady Booch, James Rumbagh and Ivor Jacobsen in the 1990's. Each of the three was a proponent of object-oriented development and each had a unique methodology with its own adherents. In the mid 90s, however, they joined forces and UML was the result.

UML constitutes a set of models in the form of a diagrammatic language for describing the architecture and use of a system. A set of standard diagrammatic elements can be used to describe one of three kinds of model of the system.

The first type of model is the "use case" model which describes the required system from a user's point of view. Use case models are a representation of typical user interaction with the system. From this conclusions are drawn about the user's requirements for the system.

The second type of model is the "static" model which describes the elements of the system and their relationships in much the same way as traditional systems modelling tools. The final type of model is a "dynamic" model which is used to design the behaviour of the model over time.

Below is an example of a use case model:



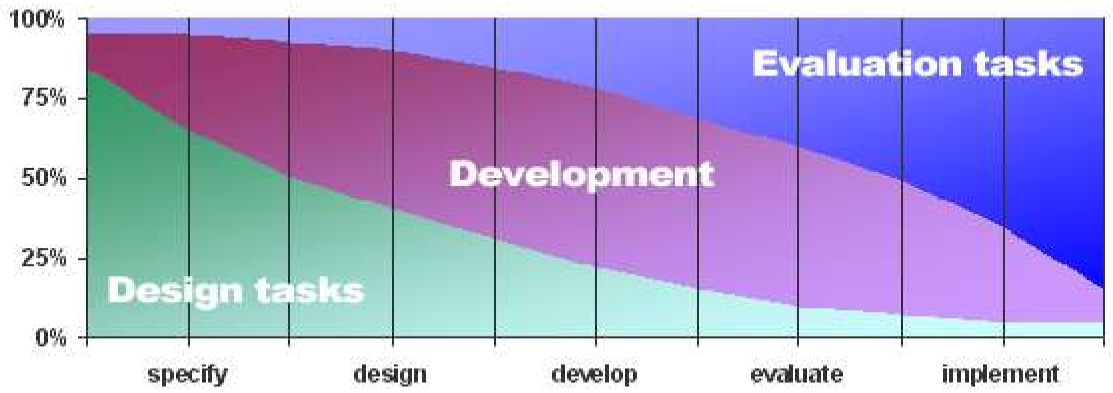
A use case model is based upon the interaction between "actors" and the system. An "actor" is anything which might exchange information with a system and includes end-users. A use case therefore becomes a situation in which an actor "uses" the system. For example an end-user may use a finance system to print an invoice or update a receipt. Each different use is represented by a use case. The collection of all use-cases is the total functionality of the system or product.

By using UML use cases, a project manager or design team can model the behaviour of the system with each type of end-user and each type of interaction. Since this is done visually it more clearly shows the relationship between actors, use cases and the working environment of the system than a static text document. Diagrammatic models, however, can be inflexible and lack detail and may sometimes need to be supported by textual references.

D e s i g n

The design phase is the first of the three, iterative 'production' phases.

In the design phase you harness your creative imagination to build something to satisfy all those stakeholder requirements you gathered in your specification phase. The design phase sets the correct direction for your development effort.



A separate design phase is necessary to help find the best solution. Any moderately complex system or product will have levels of complexity that will defeat a "suck-it-and-see" approach. By roughing out the expected solution, evaluating it against the user's expectations and refining your approach you can arrive at the optimal solution with the least amount of effort. Good design helps avoid pitfalls, blind-alleys, outright mistakes and unnecessary complexity by illuminating every aspect of your proposed solution.

Design is by its very nature iterative. Your first solution will not be your best and it will require repeated cycles of refinement to fine-tune it. Time spent in this manner is well spent and will save you from tears and drama further down the track.

Design Virtues

When designing your project implementation there are many factors you may have to consider, but regardless of the type of project you are undertaking there are some overriding principles you should consider. In fact they are so universal that I call them "virtues" since projects which embody these 'higher' ideals are always a cut above the ordinary.

Simplicity

The first and most fundamental design virtue is simplicity.

Simplicity is embodied in each of the other five virtues. The simpler a solution to a problem the more attractive it is. Programmers know this as the 'elegant' solution to a problem. The simplest, most correct solution, using the least complex code, is considered to be the 'elegant' solution.

The opposite is brute force where the least sophisticated solution is used in a repeated manner to break down the problem. Rather than spend the time to develop a simple way to solve a problem programmer's rely on the raw horsepower of computers to grind away at a problem. The costs of this approach (in terms of complexity and complication) are passed on to the end-user in the form of baffling user interfaces, superfluous functionality and escalating maintenance and support costs.

In the 14th century William of Ockham coined the Latin, "*Pluralitas non est ponenda sine neccesitate*" or "*plurality should not be posited without necessity*". In essence if you are faced with a number of competing theories or concepts you should select the one that is supported by the facts and makes the least number of assumptions.

This is regularly misquoted as "the simplest solution is often the best".

This is a handy principle to keep in mind throughout a project but especially so in design. There are many ways to solve a problem but the simplest way is often the most flexible, the most robust and the most effective. Paradoxically though, the simplest is often not the one that is first selected and a bit of work might have to be done to unearth it.

By asking questions throughout the design phase – how does this work ? do we understand this? – you can eliminate assumptions and, hopefully, complexity.

The modern paraphrase of this is KISS or "Keep It Simple Stupid". .

Consistency

Human beings, despite all the available evidence, do not enjoy conflict. They have an innate distrust of anything which is unnecessarily complex or intrinsically contradictory. This is sometimes misinterpreted as a desire for minimalism but this is inaccurate. For example a flower is an intrinsically complex object yet it is regarded as beautiful because it follows a simple but subtle and internally consistent design. A flower in which every petal garishly contrasted with the next would not be held to be nearly as beautiful as one in which each petal was identical, or a subtle variation in colour of the others. Humans like consistency.

Therefore anything that you produce should be internally consistent. If you are building a house, all of the door handles should operate in a similar manner; doing otherwise will simply annoy and possibly infuriate the occupants of the house. If you are building a software package all of the user interface controls must operate in a consistent manner. Buttons with the same label and same name should perform the same function, etc.

Consistency is a primary driving force in design.

Performance

Consumers are far more educated and 'product literate' than they have ever been before. Typical modern consumers have access to a wide range of information, including sources such as the Web, and will mercilessly evaluate your product against your competitors. In order to compete in the modern market it is no longer sufficient for particular products to be capable, consumers are demanding products that are 'best of breed' or at the very least distinguish themselves..

High-performance products also facilitate other design virtues such as ease of use and utility. In general terms the more powerful a product the more resources it <u>can</u> devote to making the end-user's life easier. A typical example is the modern desktop computer which has sufficient computing power to support the modern range of 'user-friendly' graphical interfaces.

However a caveat applies. Performance does not necessarily engender other virtues, it merely makes them possible. The VCR is a modern example of mis-directed performance. It squanders its technological potential through obscure and arcane interfaces.

Simplicity + performance = elegance.

Ease of use

Ease of use flows from simplicity, consistency and performance. A product which embodies each of the three previous principles should be simple to use. It will not be difficult to use or understand, it will be consistent in its operation and it will respond quickly to the user's requests.

Products which are easy-to-use utilise familiar metaphors and design concepts from the context in which they are set. In this way they extend their consistency to include their immediate environment. For example if the operating system on which your system or product runs has certain metaphors for user interface controls you should adopt them. You could possibly invent your own, possibly more sophisticated, user interface controls, but your project will certainly represent only a small portion of their user 'interface' experiences and they will resent having to learn particular controls just to use your software.

Do not assume you know what ease of use is, either. Most technical professionals are so far removed from the day-to-day work of typical end-users that they forget how simple concepts can confound inexperienced or unsophisticated users. You can utilise user-centric design principles like usability testing and prototyping to design the best interface possible for your software project.

Sometimes this is succinctly stated as "your software should not annoy the user".

Ease of maintenance

Things which are beautiful need not be difficult or costly to maintain. A crystal chandelier might be beautiful but the cost of maintaining one is such that fewer and fewer people are prepared to go to the extent of having one in their home. On the other hand a modern design icon like an Alessi kettle or a Smeg oven is not only beautiful but functional and easier to maintain than its predecessors.

In the same light your software should not only be easy to operate but it should be easy to maintain. Studies have shown repeatedly that software is used over time scales that far exceed its expected operating lifetime. Businesses have a reluctance to needlessly upgrade and tend to retain software even if it only barely meets their current needs. By adopting good documentation practices, good coding practices and providing flexibility in your software to adapt to changing conditions, you can ensure that it has a long and successful life.

Utility

The final and perhaps most obvious virtue is utility. A good system or product must be directly useful to the end-user or operator. The more useful it is the better.

This seems startlingly obvious but consider how many times you have encountered products which are imbued with a plethora of 'features' which only frustrate your attempts to achieve your goal. You might only want to write a letter to your friend, but the word processor will offer to produce a elaborately formatted, grammatically mangled, spelling corrupted fax/presentation/memo/report.

The temptation to include this or that 'additional' feature can be overwhelming in a software project. Each and every additional feature however has the potential to distract from delivering the finite solution that the customer or end-user requires. This destructive aspect of software development is known as feature creep (scope creep by another name) and is often driven by commercial pressure.

Include in your project *only* what is directly relevant to satisfying the requirements.

The Technical Specification

Now that you have determined *what* you are going to do, you need to decide *how* you are going to do it. The traditional method of doing this is by using a technical specification, a document that describes the design of your software.

When using an iterative development method (like the one outlined in this book) the use of prototypes and constant review can mean you won't need a Tech Spec. On some formal projects however the client or the powers-that-be will demand one. Use your own judgement and see what fits your process best.

The advantages that technical specifications offer are :

1. They force logical decomposition of a problem before any kind of solution is attempted
2. They offer another channel for documenting the decision process and disseminating ideas amongst the project team and stakeholders
3. In commercial projects they server as a document-of-record for the process undertaken and the solutions proposed for the project
4. They offer a valuable basis for ongoing documentation of the project

The disadvantages they have are :

1. Software is largely visual and tech specs tend to be textual. Describing what people will see in the finished product can be difficult. Textual description can of course be supplemented with diagrams and/or prototypes.
2. Written documents can leave room for interpretation which can lead to ambiguity or divergent expectations (usually between client and developer)

The "tech spec" usually details in words and possibly diagrams how each of the components within the project will fit together and how it will be constructed to achieve the appropriate solution.

By analogy, the requirements specification might identify the requirement as being "*to cross from the left bank of the river to the right bank*". This is "what needs to be achieved". The technical spec on the other hand will examine the possibilities of a bridge, a balloon, a ferry or a canon and select the one most appropriate to achieving those requirements (presumably ensuring that you arrive at the other side safe, dry and with all your limbs attached).

The technical spec will further document how the particular solution will be achieved. For example if (God forbid!) the canon were selected, the technical spec would specify the calibre, weight and muzzle velocity of the canon. It would also perhaps delve into what propellant is to be used, how the device is to be aimed and what kind of metal is to be used in its construction. If you were lucky the tech spec would also specify how a safe landing would be achieved.

In this way the technical spec is used to document the direction of the project, the design, before it is actually implemented. This saves time and money. If someone were to simply dive in and build a forty-inch howitzer it would probably cost the life of at least one unfortunate tester before the project team realised the solution was less than optimal. By then it would, of course, be extremely costly to scrap the canon and build a simple bridge. Conversely an astute observer might be able to draw the same conclusion from a written technical spec before any resources had been committed and the tester splattered all over the landscape (and yes I feel a certain affinity for this analogy because I use to be a tester / bearer of bad tidings).

The Format of a Technical Specification

Tech specs and requirements specs can be quite similar.

Aside from the special issue of traceability (see below) the fundamental reason for this is to reduce the administrative overhead on the team. By making the two documents relatively similar it is easy for the different team members involved with different phases of the project to compare them.

Take for example the requirements spec for a product scheduling system given previously :

3. Functional Requirements

3.1. **Product list** – *the system should produce a basic list of products available or under development*

3.2. **Current product status** – all parties highlighted the need for a clear indicator of the current status of a product. *There should be a simple flag which indicates at-a-glance whether the product is ready for release.*

3.3. **Dates** – *for each product, dates for each major milestone must be shown*. Some of these dates will remain in the public domain others will be available only to "private" users. *Private users should have the ability to publicise dates as they see fit.*

Etc, etc, etc...

The equivalent section of the technical specification might look like this:

1. Functional Requirements

1.1. **Product list** – the basic framework for the system will be a web page displayed on the company intranet which lists all products available or under development

1.2. **Current product status** – for each product a status flag will be displayed indicating one of the following states:

- **Shipping** – the product is live and available for immediate shipping
- **Concept** – the product is under consideration for development
- **Design –** the product concept has been accepted for development and basic design is underway
- **Development** – product design is complete and development has commenced
- **Testing** – product development is complete and final testing has commenced
- **Withdrawn** – the product is no longer available for sale

1.3. **Dates** – also displayed along with the current status for each product will be a row of four dates indicating the major phases of production. Dates which are not yet finalised will be displayed as "TBA" (To-Be-Announced). Dates which are completed shall be blank**.**

The table will have the following format:

Product	**Status**	**Design**	**Develop**	**Testing**	**Withdrawal**
Standalone v1	Shipping	-	-	-	31-Mar
Standalone v2	Design	31-Jan	15-Feb	5-Mar	TBA
Server v1	Shipping	-	-	-	TBA
Fat client v1	Obsolete	-	-	-	-
Fat client v2	Testing	-	-	20-Jan	TBA
Thin client v1	Concept	15-Mar	-	-	TBA

The tech spec should carry enough detail to allow developers to work without confusion but should avoid unnecessary detail.

Traceability

Given a reasonably complex project with hundreds or perhaps thousands of stakeholder requirements how do you know that you have implemented them all and satisfied the customers demands ? How do your prove during testing or launch that a particular requirement has been satisfied? How do you track the progress of delivery on a particular requirement during development?

This is the problem of traceability. How does a requirement map to an element of design (in the tech spec for example) and how does that map to an item of code which implements it and how does that map test to prove it has been implemented correctly ?

On a simple project it is enough to build a table which maps this out. On a large-scale project the sheer number of requirements overwhelm this kind of traceability. It is also possible that a single requirement may be fulfilled by multiple elements in the design or that a single element in the design satisfies multiple requirements. This make tracking by reference number difficult.

If you need a better solution than a simple table I would suggest an integrated system to track requirements for you. There are off-the-shelf software development tools available which use large databases to track individual requirements as database elements. These are then linked to similar items in specification and testing databases to provide the appropriate traceability. The benefit of a system such as this is that it can automatically produce reports which highlight problems with traceability. Such systems are generally built into or associated with SCM (Software Configuration Management) systems and can be very expensive, depending on the level of functionality.

See also the section on "change management" for a discussion of these problems.

The Problem with Tech Specs

With technical projects clients are generally unfamiliar with not only concepts and terminology but also with the functionality that is being described to them. The pace of technological change in industry is such that experienced programmers have trouble keeping up with it, let alone professionals from another field.

The typical situation that occurs is that all the experts go through a lengthy and detailed design process and deliver a custom built solution, tailor-made to fit the client's requirements.

At this point the client looks at it and says: "That's not what I want!"

And the developer says: "But that's what you asked for!"

There has been a fundamental breakdown in communications and no-one can understand why.

The problem stems from the fact that the project manager is an experienced technical practitioner and the client is not. The project manager is used to interpreting specifications and the client is not (probably). When the project manager sees "a multi-function hierarchical picklist" in a spec they understand what it means. The client sees convincing techno-babble delivered by a competent technical expert and (at first) nods vigorously. Hence the problem.

It doesn't matter how good a spec you write or how much detail you put into your design documentation there will always be a necessary gulf of understanding between you and your client This is called the "expectation gap" (or perhaps the "assumption gap").

Bridging the Expectation Gap

How do you bridge the expectation gap?

Client's are very good at taking something tangible and applying it to their own work environment. No one knows that environment better than they do. They are an invaluable source of information about how your proposed solution will work in the context of their environment.

But how do you provide them with something to look at ?

It is obviously infeasible and impractical to wait until the product is complete to ask them to evaluate it. By then it will be too late to implement any changes and you will be simply seeding dissatisfaction amongst stakeholders and your project team.

You need to develop 'examples' of what the software might look like when finished.. A prototype is a quick-and-dirty rendition of the software which demonstrates of a design concept to and end-user. It is delivered in a fraction of the time and cost that a fully functional system would.

By showing a prototype to the client which embodies their requirements in a tangible form you can much more accurately gauge how close you are to meeting their requirements. By using a visual, tangible form you are speaking a language the client understands.

If you repeat this exercise a number of times you can quickly home in on the clients ideal solution.

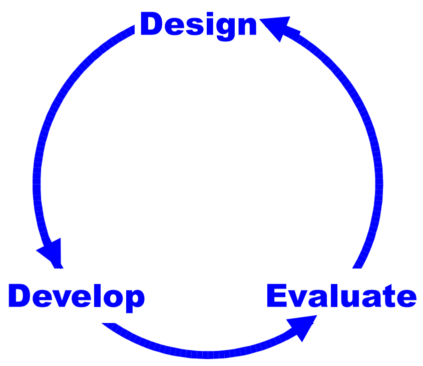
This is known as Rapid Prototyping.

Rapid Prototyping

The essence of rapid prototyping lies in speed. Prototyping can be resource-intensive and if it is to be of real value then the cost in resources must be minimised. Prototypes must be designed, developed and evaluated as rapidly as is feasible. *The strength in prototyping lies not in the prototypes themselves but in the rapid iteration of design ideas and in the feedback from the users.*

There is no time to refine or polish prototypes, they should be turned out in front of users as rapidly as possible. Prototypes can be constructed as actual systems and software products or they can be simulated with screenshots, mock-ups and even paper-based representations. The focus is not to build a final product, but to decide what that product should look like and how it should act.

The Process of Rapid Prototyping



Rapid prototyping is a driving force behind the process I have already described : design-develop-evaluate.

Any issues that arise during evaluation are noted and fed back into the design process to generate new ideas and a new prototype for evaluation. A typical cycle should happen as rapidly as possible, preferably on the order of hours to days and not weeks or months.

Later in the development process you will tend to scale back the prototyping work and focus on developing the solution. This is perfectly normal and reflects the fact that earlier phases have reduced uncertainty in design and you can focus on delivery. Should a new design decision arise however the development team should not hesitate to flop out a new prototype to test their assumptions.

The Benefits of Rapid Prototyping

Responsive development – because the project goes through iterative phases of design, development and evaluation it is possible to adjust the design on-the-fly. By not committing to massive chunks of development without evaluation, the team minimises the chance of misdirected time during coding.

Rapid turn-around – because design, development and evaluation are integrated into a tight control loop the chances of staying on track are much better. Large-scale, linear developments have a horrible tendency to save up trouble till late in the project and blow-out both schedule and budget only during the final phases of testing. By spreading corrections throughout the life-cycle it is possible to head off this problem and to resolve issues sooner rather than later.

Customer-centric design – the use of prototypes allows the inclusion of customers or end-users in the design process. By producing and evaluating prototypes it is possible for the end-users to directly influence the design. Rather than being a black box process, development therefore becomes much more open and flexible. It also begins to shape the expectations of end-users and customers as they see the unfolding design. In contrast, during a traditional development cycle, the first time a user will see the finished product is at the launch — when you might come to understand what is meant by a "big-bang" implementation.

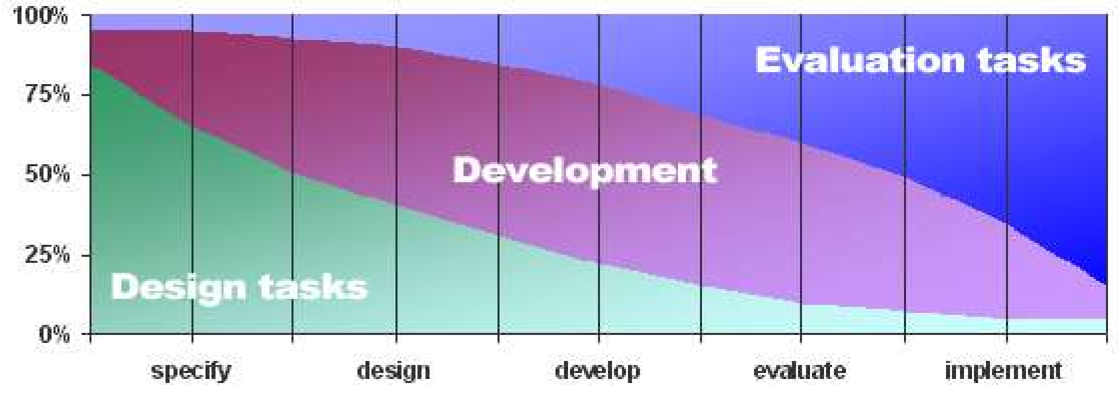
The Pitfalls of Rapid Prototyping

Using the prototype as a basis for production - A prototype is <u>not</u> the product, nor even a precursor to the product. The prototype is a tool by which you make design decisions which are then implemented in the product. Prototypes should be discarded after the correct design has been selected. Prototypes are built with speed in mind and your final product should be built with quality in mind. The danger of building prototypes into final product is that shortcuts and approximations of the prototype are built into the final product along with the correct design elements. This means that what should be an inherently robust, well-engineered product will be riddled with bugs. Prototypes should remain separate from final product.

Losing control of the cycle – because this process is inherently more flexible than a linear model there is the risk of it running amok. Unless careful control is placed on the process it is possible to iterate round the circle of design, develop and evaluate ad infinitum. Without strong time-management discipline this model can become unworkable.

Lack of visibility or version control – because there is a rapid turn around between the three phases things can move pretty fast. If your infrastructure is unable to cope with rapid iterations in design your specifications and prototypes may well become out of synch and it will not be clear which way your product is actually going. This is also the point at which a prototype is in danger of becoming the basic of an actual product.

The development phase is the second of the three, iterative production phases. In early stages, work will be focussed on developing prototypes for evaluation and in the later phases it moves into producing delivery quality code for testing and launch.



In the context of our 'model' you will still be executing cycles of design-develop-evaluate but now the emphasis is on the development portion and not design. Revisions to design and the testing of individual units will also occur, but the brunt of your effort will be devoted to writing and developing the product or system. Prototypes will become less useful, but still may be used to resolve particular design decisions.

Staying on track

The single biggest problem during the production phase of the project is staying on track. Despite all the best planning, having the best team and anticipating all the possible pitfalls projects seem to have a knack of wandering off course.

The Myth of Completion

There is a common myth about tasks which is perpetrated by project management tools such as MS Project. The myth is this: tasks can be partially complete, i.e. a task can be 10% or 20% done.

If a task is thought of as a goal then the lie becomes obvious – either you have achieved your goal or you have not. It's a black-and-white, binary proposition. If your goal is a vague and imprecise statement of intent, like "write some instructional documentation", then it is complete the moment you start. As soon as you put pen to paper you have "written some instructional documentation".

On the other hand if the task is well defined and has a measurable deliverable then the goal is not achieved until it is delivered. For example: "complete a user guide and a technical manual, ready for publishing" is much more useful because it has a clear measure of success. It is only complete when the documents are written, have been reviewed and edited and are ready for publication.

The danger in believing that tasks can be partially completed is that it gives you a false sense of security. Because 50% of a task can be such a hard thing to define, people will tell you they have completed 50% of the task when they are 50% of the way through the time allocated to it. They might only be 10% of the way through it, but 50% sounds much better, if there is 50% of the time left there must be only half of the work left, too!

This misconception is particular entertained by those people who believe that time is elastic. That is that you can cram any amount of work into a particular length of time, it just depends on how hard you work.

These are the people that will tell you one of two things:

1. After working 10 days on a 20-day task they announce that they have only done 10% (i.e. nothing but 10% sounds better) and that they will make up the time. This is despite the fact that they fought tooth-and-nail in the first place to get 20 days allocated to the task. Truth be told they are simply rolling their delays from one task to the next and picking up speed as they rip through the project. By the time they hit a milestone they will be way behind and so will you!
2. The second case is that they report consistent progress all the way through the first 15 days of a 20 day task and then their progress suddenly slows down so that on successive days they go from 95% complete to 96% complete. They will certainly overrun their task and be claiming three weeks later that the last 1% of the task is nearly done. The old adage "the first half of a task takes 90% of the time and the other half of the task takes the remaining 90% of the time" is all too often true.

It is much better to break tasks down into smaller steps and to simply report on their completion. Then if a particular task misses its completion date it automatically triggers an adjustment to the schedule. There is no ambiguity, there is no confusion.

Standards and Process

The basis of quality software in any enterprise is the establishment of minimum standards and processes. "Standards" however is second only to "metrics" in the average programmer's lexicon of dirty words. An individual developing software by themselves can rely on their creative genius to carry them through, for they alone will be accountable for their mistakes. An organisation developing software however must sacrifice individual flair for prudence and establish a base level of operation, a standard.

Standards could be also be extended to include the following areas:

- Coding style and bad practices to be avoided
- Source and version control (see below)
- Compilation and linking of code
- Handover of iterations between development and test teams
- Change control
- Defect Tracking
- User Interface style

Adoption of standards is usually a carrot-and-stick affair. Everyone recognises the need for them but rankles at having to implement 'needless bureaucracy'. In some industries the adherence to standards is mandated by the nature of the work (mission critical systems, military applications etc) for most of us however it's just 'overhead'. Develop a system that works for you, get consensus from the whole team and then punish people who don't follow the agreed standard. Given human nature, it's the only way to get consistency into the process.

Change Management Process

I use the phrase "change management" here to mean the overall management of change withing our software project. Source control is covered in the next section.

The basis of change management is to have a clear process which everyone understands. It need not be bureaucratic or cumbersome but it should be applied universally and without fear of favour. The basic elements of a change process are :

- What is under change control and what is excluded ?
- How are changes requested ?
- Who has the authority to approve or reject changes ?
- How are decisions upon approval or rejection are documented and disseminated ?
- How changes are implemented and their implementation recorded ?

The process should be widely understood and accepted and should be effective without being bureaucratic or prescriptive. It is important for the project team to be seen to be responsive to client needs and nothing can hurt this more than an overly-officious change control process. Change is inevitable in of a project and while you need to control it you do not want to stifle it.

A typical process might be as minimal as the following:

1. Once a project document has been signed-off by stakeholders, a change to it requires a mandatory *change request* to be logged via email. The request will include the nature of the change, the reason for the change and an assessment of the urgency of the change.
2. A "change control board" consisting of the development manager, test lead and a product manager will assess the issue and approve or reject each request for change. Should more information be required a member of the change control board will be assigned to research the request and report back.
3. No change request should be outstanding for more than a week.
4. Each change which is accepted will be discussed at the weekly development meeting and a course of action decided by the group. Members of the development team will then be assigned to implement changes in their respective areas of responsibility.

If you have a flexible change request process team members can be encouraged to use it to seek additional information or clarification where they feel it would be useful to communicate issues to the whole project team.

Tracking Change

To make change management easy you need a simple method of tracking, evaluating and recording changes. This can be a simple database or log but in large projects it has evolved into an customised information system in its own right.

As such the system needs to be able to handle:

- Logging requests for changes against products and documentation
- Recording and managing the priority of a particular change
- Logging the decision of a change management authority
- Recording the method and implementation of change
- Tracking implemented changes against a particular version of the product or document

The more structured a system the more secure the change control process, but obviously the more overhead. A balance must be struck between the enforcement of proper procedure and responsiveness of the system.

Change management systems are useful for managing everyone's expectations too. Often decisions are requested by stakeholders or clients and if proper consultation is not entered into they can sometimes assume they will automatically be included (just because they asked for it).

If the volume of change requests is particularly high (as it often is) communicating what's in and what's out manually can be difficult. A simple, well understood change management system can often be directly used by stakeholders to log, track and review changes and their approval. This is particularly true for projects that span disparate geographical locations where meetings may not be possible.

In many projects the change management system can be linked to (or is part of) a defect tracking system. Since resolution of a defect is, in effect, a request for change both can often be handled by the same system. The change and defect tracking system can also be linked with version control software to form what is commonly known as a Software Configuration Management (SCM) system. This integrated system directly references changes in the software against specific versions of the product or system as contained in its version control system. The direct one-to-one reference cuts down on management overhead and maintenance and allows the enforcement of strict security on allowable changes.

Source and Version Control

A source control system restricts access to source code. In much the same way as a database locking system prevents multiple users from modifying the same record at the same time, a source control system stops multiple developers from modifying the same code at the same time.

A version control system extends this concept by tracking multiple versions of the code over time and identifying the differences between them. This allows developers to "roll back" to a previous version of the code without tedious assembly of source code. A version control system usually contains a "make" or compilation system which automatically extracts the latest versions of relevant modules of code in order to build a complete system.

Version control systems have a number of advantages. They allow you to build and track releases easily, they provide a convenient method of back and roll back and they allow parallel development. Code can be built up from basic components into an integrated system with many developers (or many teams) working on many different areas of the code simultaneously. In an ideal situation each block of code would be entirely independent but in reality this situation is rarely possible. Usually there is some overlap or dependency between areas of the code and changes made in one module have cascading effects in all linked modules.

Without a source control system it would be possible for multiple programmers to modify the same root level code simultaneously, possibly leading to confusion, conflict and errors. A source control system would restrict access to each module of source code to a single developer. When that individual has the source code "checked-out" it is impossible for any other developer to gain access to the code and therefore prevents parallel modifications. When the developer in question has completed their changes they check the code back in and it becomes available.

A version control system extends this concept by allowing multiple versions of the same source code by giving developers the option of "branching" the source code to create a new baseline source with their additional changes. Although branching allows more flexibility it can cause problems when multiple branches must be re-merged to form a single product or executable. The hand merging of such code can be extremely difficult.

The most sophisticated of modern versions of version control systems alleviate this by assisting in the automation of merging of code. Using a comparative source editor (one which highlights the changes in the source code, perhaps by colour) developers checking code back in are forced to re-merge the code at this point, avoiding complicated merging at a later date.

Compilation and build processes go hand-in-hand with version control systems. In order to get the most out of them a version control system must be supported by processes which ensure it is not abused. Developers regularly find ways to subvert the version control system and allow them to work on separate versions of the code. A commonly adopted practice is to have a separate build-and-release team who perform regular builds of the system for release to testing or customers. The developers have the individual responsibility of getting the latest version of their code checked back into the system in time for a release. The build team then simply builds the repository in its current state, incorporating all the latest code as it resides in the version control system. The use of a build-and-release team also ensures standards are adhered to during the build cycle.

White Box testing

The testing of code during development is often known as white-box testing. Testing of completed functional code is known as black-box testing because testers treat the object as a black-box and concern themselves with verifying input against output. White-box, or glass-box testing relies on analysing the code itself and the internal logic of the software. White-box testing is often, but not always, the purview of programmers.

Code Inspection

The basic form of static evaluation is code inspection. As its name implies it relies on inspecting the code, examining the logic and structure and comparing it with accepted best practices. In large organisations or mission-critical applications a formal inspection board can be established to make sure that written software meets the required minimum standards. In other less formal inspections a project manager or quality team can be responsible for evaluating completed code.

Code inspection can sometimes be automated. Many syntax and style checkers exist today which verify that a module of code meets certain defined standards. By running an automated checker across code it is easy to check basic conformance to standards and highlight areas that need human attention.

A variant on code inspection is the use of peer programming as espoused in methodologies like Extreme Programming. In peer programming the programming responsibilities for a module of code are shared between two individuals. While one person will write most of the code the other is responsible for reviewing and evaluating the quality of the code. The reviewer looks for flaws in logic, lapses of coding standards and bad practice. The roles are then swapped.

Advocates assert this is a speedy way to achieve good quality code and opponents retort that its a good way to waste a lot of people's time. As far as I'm concerned the jury is still out.

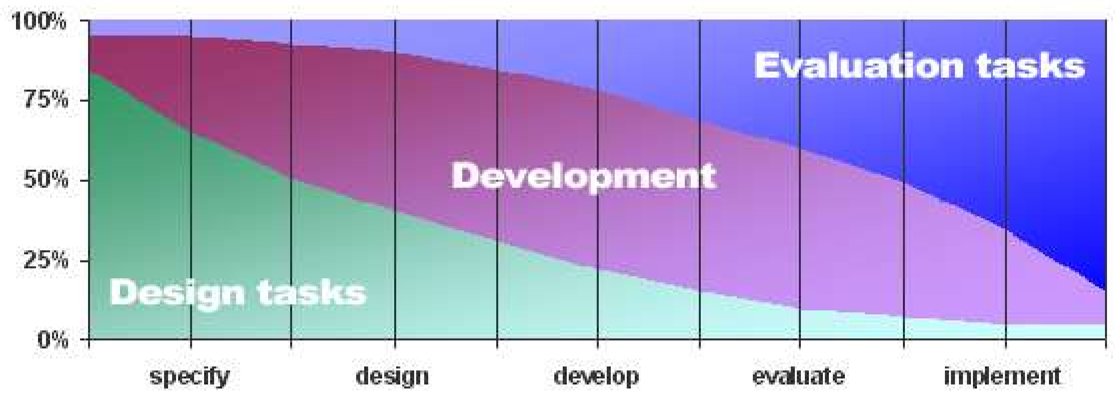
Dynamic Analysis

Dynamic analysis methods rely on analysing the compiled code of a product or system *while it is running.* While the previous kinds of white-box testing looked at source code in its static format, dynamic analysis looks at the compiled/interpreted code running in the appropriate environment. Normally this is an analysis of variable quantities such as memory usage, processor usage or overall performance.

One common form of dynamic analysis used is that of memory analysis. Given that memory and pointer errors form the bulk of the issues encountered with software programs, memory analysis is extremely useful. A typical memory analyser reports on the current memory usage level of a program under test and of the disposition of that memory. The programmer can then ‘tweak’ or optimise the memory usage of the software to ensure the best performance and the most robust memory handling.

Often this is done by ‘instrumenting’ the code. A copy of the source code is passed to the dynamic analysis tool which inserts function calls to its external code libraries. These calls then export run time data on the source program to an analysis tool. The analysis tool can then profile the program while it is running. Often these tools are used in conjunction with other automated tools to simulate realistic conditions for the program under test. By ramping up loading on the program or by running typical input data sets the program’s use of memory and other resources can be accurately profiled under real-world conditions.

The evaluation or testing phase is the last of the three production phases: design, develop, evaluate.



The purpose of evaluation is to reduce risk.

The unknown factors within the development and design of new software can derail it and minor risks can delay the project. By using a cycle of testing and resolution you can reduce uncertainty and eliminate errors. Testing is also the only tool in your arsenal which reduces errors. Any kind of development potentially introduces more errors since it introduces more functionality. Planning and design can help limit risk but they cannot reduce or eliminate risk already present in a project.

Concepts of Testing

The Testing Mindset

There is particular philosophy that accompanies "good testing".

A professional tester approaches a product with the attitude that the product has defects and it is their job to discover them. They assume the product or system they receive is inherently flawed and it is their job to 'illuminate' the flaws. This attitude is necessary but can bring conflict with developers and designers.

This approach really is necessary to testing. Designers and developers approach software with optimism based on the assumption that the product is basically working correctly and just needs to be refined to be complete. This means that they often overlook fundamental issues or fail to recognize them when they see them. By taking a sceptical approach, the tester offers a balance.

Test Early, Test Often

There is an oft-quoted truism of software engineering that states: a bug found at design time costs ten times less to fix than one in coding and a hundred times less than one found after launch. Barry Boehm quotes ratios of 1:6:10:1000 for the costs of fixing bugs in requirements,design, coding and implementation (upper limits).

Nor is a single pass of testing enough. Your first past at testing simply identifies where the issues occur. At the very least, a second pass of (post-fix) testing is required to verify that issues have been correctly resolved. The more passes of testing you conduct the more confident you become and the more you should see your project converge on its delivery date.

Regression vs. Retesting

From the above explanation the need for re-testing is fairly evident. You must re-test fixes to ensure that issues have been resolved satisfactorily before development can progress. Broadly speaking, re-testing is the act of repeating a test to verify that a previously found issue has been correctly fixed. Regression testing on the other hand is the act of repeating other tests in 'parallel' areas to ensure that the applied fix has not introduced other errors or unexpected behaviour.

For example, if an error is detected in a particular file handling routine then it might be corrected by a simple change of code. If that code, however, is utilised in a number of different places throughout the software the effects of such a change could be difficult to anticipate. What appears to be a minor detail in the way a module of code handles a particular process could affect a separate module of code elsewhere in the program. What therefore appears to be a bug fix could in fact be introducing bugs elsewhere in the program. It has been estimated that up to 50% of bug fixes actually introduce additional errors in the code.

Not only that, but the programmers also risk introducing casual errors every time they place their hands on the keyboard. An inadvertent slip of a key that replaces a full stop with a comma might not be detected for weeks but could have serious repercussions. Regression testing attempts to reduce these risks by reassessing the 'area of impact' affected by evaluating the likely impact of the fix and determining if anything has changed (in terms of expected behaviour).

Verification and Validation

Two major types of tasks in testing are verification and validation.

Verification tasks are designed to ensure that the product is internally consistent. They ensure that the product meets the design, which meets the specification, which meets the requirements . . .and so on. The majority of testing tasks fall into the verification category with the final product being checked against known good references to ensure the output is as expected.

Test plans are normally written from the requirements documents and from the specification. This verifies that the software delivers the requirements as laid out in the technical and requirement specifications, it does not however address the 'correctness' of those requirements.

> On a large scale project I worked on as a tester, we complained to the development team that our documentation was out of date and we were having difficulty constructing valid tests. They assured us they would update the specification forthwith and provide us with a new version to plan our tests from.
>
> When I came in the next day I found two programmers sitting by a pair of computer terminals. While one of them ran the latest version of the software, the other would look over their shoulder and then write up the onscreen behaviour of the product as the latest version of the specification.
>
> When we complained to the development manager she said "What do you want? The spec is up to date now, isn't it?". The client, however, was not amused; they now had no way of determining *what the program was supposed* to do as opposed to *what it actually did*.

Validation tasks are just as important as verification, but less common. Validation is the use of external sources of reference to ensure that the internal design is valid, i.e. it meets users expectations. By basing some of their testing on external references (such as the direct involvement of end-users) the test team can validate the team's decisions and ensure the project is heading in the correct direction. Usability testing is a prime example of a useful validation technique.

Test Planning

The Purpose of Test Planning

As part of a project, testing must be planned to ensure it delivers on its expected outcomes. Test planning represents a special challenge, however. Essentially, the aim of test planning is to decide where the bugs in a product or system will be and then to design tests to locate them.

The paradox is of course, that if we knew where the bugs were then we could fix them without having to test for them. Testing is the art of uncovering the unknown and therefore can be difficult to plan.

The usual, naïve retort is that you simply test "all" of the product. Even the simplest program however will defy all efforts to achieve 100% coverage (see the appendices). Even the term coverage itself is misleading since this represents a plethora of possibilities. Code coverage, branch coverage, input/output coverage ? There are so many to choose from and the stark reality for testing is that complete coverage (and complete confidence) of any sort is simply not possible.

There is an answer however.

At the start of testing there are a (relatively) large number of issues with the project and these can be uncovered with little effort. As testing progress the law of diminishing returns applies and more and more effort is required to uncover subsequent issues. At some point the investment to uncover that last 1% of issues is outweighed by the high cost of finding them. The cost of letting the customer or client find them will actually be less than the cost of finding them in testing.

The purpose of test planning therefore is to put together a plan which will deliver the right tests, in the right order, to discover as many of the issues with the software as time/budget allows.

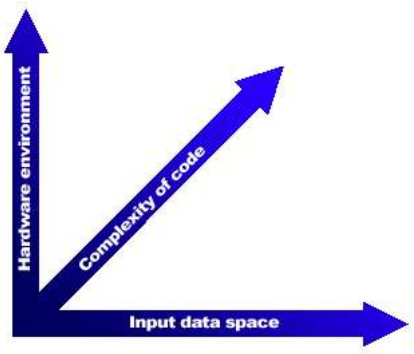
> But how much testing is enough?
>
> Firstly, remember that companies live and die on their reputation for quality. You might be tempted to cut back on testing in the expectation that the chance of failure is small and the number of customers affected will be minimal. This is dangerous territory however; the quality of your software is yet proven.
>
> Also remember that you are not judged in absolute terms but in competition. If your competitors put in that extra effort and are able to demonstrate that your product is inadequate when compared to theirs then, no matter how small the difference, they will be able to take business away from you.
>
> This is a subtle and difficult subject to deal with and opinions within the team will be many and varied. Most experienced software developers I know however, err on the side of *more* testing.

The Process of Test Planning

Given that we don't *a priori* know where issues are going to occur in the product or system we must design a test regime which makes the most efficient effort to uncover issues. This is where our 'model' comes into play again. By using the model of "design-develop-evaluate" we can constantly refine our testing approach to ensure it delivers the best results. By starting with a broad-based test plan we can identify areas of likely risk and focus our efforts on specific areas which will yield the most bugs.

But how to identify those areas of risk?

It is useful to think of software as a multi-dimensional entity with many different axes. For example one axis is the code of the program, which will be broken down into modules and units. Another axis will be the input data and all the possible combinations. Still a third axis might be the hardware that the system can run on, or the other software the system will interface with. Other possible include functionality, code structure and 'performance'.



Testing can then be seen as an attempt to achieve "coverage" of as many of these axes as possible to find bugs. Remember we are no longer seeking the impossible 100% coverage but merely an indication of where issues lie, where the likely areas of risk are. Testing can then be focused on these areas in order to find and eliminate the issues.

Outlining

To start the process of test planning a simple process of 'outlining' can be used.

Take each of the axes and break each one down into its component parts. For example, with the axis of "code complexity" the program would be broken down into the 'physical' organisation of the component parts of code that make up the system or product. Taking the axis of "hardware environment" (or platform) it would be broken down into all possible hardware and software platform combinations that the system or product will be expected to run on.

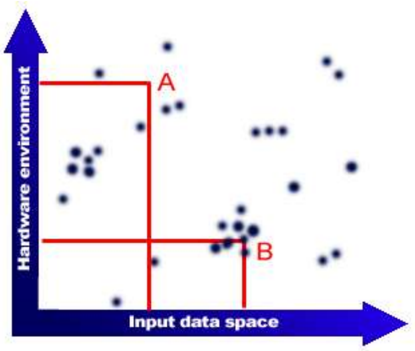
This is a process of deconstructing the software into constituent parts based on different taxonomies. For each axis simply list all of the possible combinations you can think of. Testing will seek to cover as many elements as possible on as many axes as possible and focus efforts wherever issues are found.

Axis / Category	Explanation
Functionality	As derived from the technical specification
Code Structure	The organisation and break down of the source or object code
User interface elements	User interface controls and elements of the program
Internal interfaces	Interface between modules of code (traditionally high risk)
External interfaces	Interfaces between this program and other programs
Input space	All possible inputs
Output space	All possible outputs
Physical components	Physical organisation of software (media, manuals, etc)
Data	Data storage and elements
Platform and environment	Operating system, hardware platform
Configuration elements	Any modifiable configuration elements and their values

Test Case Design

The next step in testing is to design test cases which cover or exercise each of the points detailed on your outline. Note that a single test may in fact validate more than one point on one axis. A test could simultaneously validate functionality, code structure, a UI element and error handling.

In the diagram shown on the next page two tests have been highlighted in red : A and B. Each represents a single test which has verified the behaviour of the software on the two particular axes of input data space and hardware environment.



The dark splotches in the diagram denote bugs in the software. You will note that bugs tend to cluster around one or more areas with in one or more axes. These define the areas of risk in the product. Perhaps this section of the code was completed in a hurry or perhaps this section of input space was particularly difficult to deal with. Whatever the reason, these areas are inherently more risky, more likely to fail than others.

You can also note that test A has not uncovered a bug it has simply verified the behaviour of the program at that point. Test B on the other hand has identified a single bug in a cluster of similar bugs. Focussing testing efforts around area B will be the most productive use of effort since there are more likely to be more bugs to be uncovered here. Focussing around A will probably not produce any significant results.

Your aim should be to provide a broad coverage for the majority of your outline 'axes' and deep coverage for the most risky areas discovered. Broad coverage implies that an element in the outline is evaluated in a cursory or elementary fashion while deep coverage implies a number of repetitive, overlapping test cases which exercise every variation in the element under test.

The aim of broad coverage is to identify risk areas and focus the deeper coverage of those areas to eliminate the bulk of issues. It is a tricky balancing act between trying to cover everything and focusing your efforts on the areas that require most attention.

Use your intuition too, risky code can be identified through 'symptoms' like unnecessary complexity, historical occurrences of issues, stress under load and reuse of code. Use your instincts to hone your focus for testing and ask the opinions of others. Developers often know where bugs lurk in their code.

As you progress through each cycle of evaluation you can further refine your test plan. As each cycle of testing uncovers more issues you can shift testing to the more risky areas of your product or system. During the early stages of testing not many issues are found. As testing hits it stride issues start coming faster and faster until the development team gets on top of the problem and the curve begins to flatten out again.

This is the point where your risk/reward ratio begins to flatten out and it may be that you have reached the limits of effectiveness with this particular form of testing. If you have more testing planned or more time available, now is the time to switch the focus of testing to a different point in your outline strategy.

Cem Kaner said it best when he said "The best test cases are the ones that find bugs." A test case which finds no issues is not necessarily worthless but it obviously is worth much less than a test case which *does* find an issue. Your efforts must focus on those test case that find issues. The cycle of refinement should be geared towards discarding pointless or inefficient tests and diverting attention to more fertile areas for evaluation.

Also, referring each time to your original outline will help you avoid losing sight of the wood for the trees. While finding issues is important you can never be sure where you'll find them so you can't assume the issues that you are finding are the only ones that exist. You must keep a continuous level of *broad coverage* testing active to give you an overview of the product or system while you focus the *deep coverage* testing on the trouble spots.

Testing the design

Design documents and specifications can be tested in their own right and not only as part of functional testing of a delivered system or product. While all design documentation can be evaluated requirements and technical specifications have a pivotal importance in the life-cycle.

The purpose of evaluating a specification is threefold:

- Testing the spec itself to make sure it is accurate, clear and internally consistent
- Evaluating how well it actually reflects the reality and what the end-user expects
- Making sure it is consistent with all previous and subsequent phases of the project

The specification is an embodiment of the requirements which should then flow through to subsequent phases like production and testing. If the requirements are poorly specified then not only will the product be inadequate but it would also be incredibly difficult to verify its status accurately. If the technical specification is out of synch with the requirements specification then it is likely that the development team will be well on its way to producing the wrong product.

Testing a requirements specification consists of reviewing each requirement from both a validation and a verification point of view. Each requirement should be reviewed against a list of desirable attributes:

- Specific – an ambiguous requirement sows uncertainty in the whole process. If you can't be specific about the target at the start of development then you can't expect the product to be on target nine months later. *Above all else, a specification must be specific!*
- Measurable – a requirement which specifies a quantitative or qualitative improvement must do so with a specific value for that improvement (100% faster or 20% more accurate, etc)
- Testable - given all of the above, is the requirement testable? If not, it should be re-written so that it can be tested. A requirement which is not testable is ultimately not 'provable' and cannot therefore be confirmed either positively or negatively.
- Consistent - if one requirement contradicts another, the contradiction must be resolved
- Clear - requirements must be simple, clear and concise.

Further, a commercial specification should not talk about "how" to do something and a technical spec should not talk about "why" to do things. Problems in the tech spec or requirements spec can usually be spotted by the following symptoms:

- Ambiguity: words like "probably", "might" or "if" indicate indecision on the part of the author and hence ambiguity. Requirements including these words should be either eliminated or re-written to provide some kind of surety as to the desired outcome.
- Consistency: check for inconsistency between versions of the same requirement, eliminate the ambiguous or combine them all into a single requirement
- Clarity: requirements should be stated separately. Requirements composed of long-winded multiple sentences or of a single sentence with multiple clauses imply multiple possible outcomes and so lacks clarity. Split them up into single statements.
- Measurability: comparative words like "better" or "improved" or "greater" should be accompanied by a quantitative value. *How much* better? *How much* improved?
- Inclusion and exclusion: specs should not only state what will be done, but explicitly what will *not* be done. Leaving something un-specified does not necessarily imply this.

Functional Testing

If the aim of a project is to "deliver widget X to do task Y" then the aim of "functional testing" will be to prove that widget X actually does task Y. Simple ? Well, not really.

As discussed, any reasonably complex deliverable will have more inputs, states and outputs than you can reasonably afford to test given the constraints imposed upon your project. Given an infinite amount of time and resources you could cover all of the possibilities but given a real world situation you must decided what is an acceptable level of testing.

Alpha and Beta Testing

As has already been explained it is important to test early and often. While embracing a rapid prototyping model will enable most production efforts to include a large number of test cycles there are some commonly recognised milestones in the testing lifecycle.

Typically these milestones are known as "alpha" and "beta" tests. There is no precise definition for what constitutes alpha and beta test but the following are offered as common examples of what is meant by these terms :

- Alpha – enough functionality has been reasonably completed to enable the first round of system testing to commence. At this point the interface is usually not complete and many of the bugs in the system have not been eliminated.
- Beta – the bulk of functionality has been completed and remaining work is aimed at improving performance, eliminating defects and completing cosmetic work. At this point the UI is usually 'all there' but obvious bugs still remain.

In the software industry Beta testing is often associated with the first end-user tests. The product is sent out to prospective customers who have registered their interest in participating in trials of the software. Beta testing, however, needs to be well organised and controlled otherwise feedback will be fragmentary and inconclusive. Care must also be taken to ensure that a properly prepared prototype is delivered to end-users, otherwise users will be disappointed and time wasted.

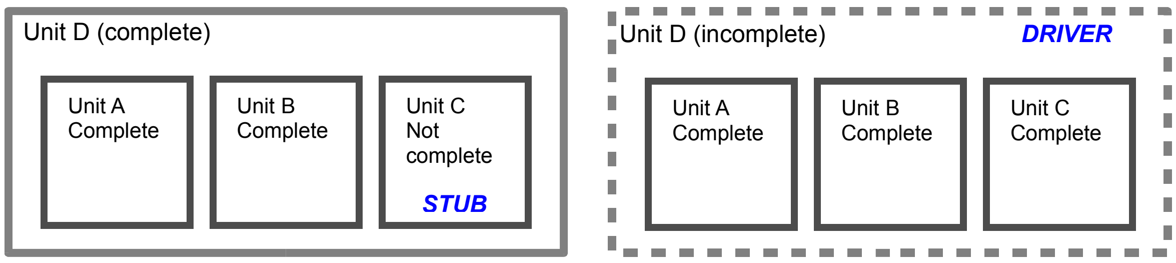
Unit, Integration and System testing

The first type of testing that can be typically conducted in any development cycle is **unit testing**. In this, discrete components of the final product are tested independently before being assembled into larger units. Units are typically tested through the use of 'test harnesses' which simulate the context into which the unit will be integrated. The test harness provides a number of known inputs and measures the outputs of the unit under test, which are then compared with expected values to determine if any issues exist.

An important concept related to unit testing is 'smoke' testing. Smoke tests are quick-and-dirty tests which are performed rapidly to validate the basic quality of a piece of the project. By subjecting it to some basic tests the team can determine whether it is ready to be integrated into larger units. This saves time and resources if the unit has major flaws.

> The term 'smoke testing' comes from the electronics industry. If an engineer designs a circuit or an electrical system the first test to be conducted is to plug it into the power briefly and switch it on. If smoke comes out then you switch it off and it's back to the drawing board (or soldering iron). If no smoke appears then you it's basically okay and you can try some more tests on it to see if it is working properly. The same principle can be applied in software development.

In **integration testing** smaller units are integrated into larger units and larger units into the overall system. This differs from unit testing in that units are no longer tested independently but in groups, with the focus shifting from the performance of individual units to the interaction between them. At this point "stubs" and "drivers" take over from test harnesses. A stub is a simulation of a particular sub-unit which can be used to simulate that unit in a larger assembly. For example if units A, B and C constitute the major parts of unit D then the overall assembly could be tested by assembling units A and B and a simulation of C, if C were not complete. Similarly if unit D itself was not complete it could be represented by a "driver" or a simulation of the super-unit.



As successive areas of functionality are completed they can be evaluated and once complete are integrated into the overall project. Without integration testing you are limited to testing a completely assembled product or system and, as discussed previously, this is inefficient and error prone. Much better to test the building blocks on the fly and build your project from the ground up in a series of controlled steps.

System testing represents an overall test on an assembled product. Systems testing is particularly important because it is only at this stage that the full complexity of the product is present. The focus in systems testing is typically to ensure that the product responds correctly to common input conditions and (importantly) the product handles exceptions in a controlled and acceptable fashion. System testing is often the most formal stage of testing and the most structured (an even more structured variant – acceptance testing is discussed below).

Acceptance Testing

Some formal software projects have a special, final phase of testing called "Acceptance Testing". Acceptance testing forms an important and distinctly separate phase from previous testing efforts and its purpose is to ensure that the product meets minimum defined standards of quality prior to release. This is normally where the client signs the cheques. Often client or end-user representatives will conduct the testing to verify the software has been implemented to their satisfaction (User Acceptance Testing or UAT).

While other forms of testing can be more 'free form', the acceptance test phase should represent a planned series of tests and release procedures to ensure the output from the production phase reaches the end-user in an optimal state, as free of defects as is humanly possible.

Non-functional Testing

Usability Testing

Usability testing is the process of observing users' reactions to a product and adjusting the design to suit their needs. Marketing knows usability testing as "focus groups" and while the two differ in intent many of the principles and processes are the same.

In usability testing a basic model or prototype of the product is put in front of evaluators who are representative of typical end-users. They are then set a number of standard tasks which they must complete using the product. Any difficulty or obstructions they encounter are then noted by a host or observers and design changes are made to the product to correct these. The process is then repeated with the new design to evaluate those changes.

There are some fairly important tenets of usability testing that must be understood :

- *Users are not testers, engineers or designers* – you are not asking the users to make design decisions about the software. Users will not have a sufficiently broad technical knowledge to make decisions which are right for everyone. However, by seeking their opinion the development team can select the best of several solutions.
- *You are testing the product and not the users* – all too often developers believe that it's a 'user' problem when there is trouble with an interface or design element. Users should be able to 'learn' how to use the software if only they are taught right! Maybe if the software is design properly, maybe they won't have to 'learn' it at all ?
- *Selection of end-user evaluators is critical* –You must select evaluators who are directly representative of your end-users. Don't pick just anyone of the street, don't use management and don't use technical people unless they are your target audience.
- *Usability testing is a design tool* – Usability testing should be conducted early in the life-cycle when it is easy to implement changes that are suggested by the testing. Leaving it till later will mean changes will be difficult to implement.

One issue that people often posit about usability studies is the misconception that a large number of evaluators are required to undertake a study. Research has shown that no more than four or five evaluators might be required. Beyond that number the amount of new information discovered diminishes rapidly and each extra evaluator offers little or nothing new. Consider this, if all five evaluators have the same problem with the software is it likely the problem lies with them or with the software ? With one or two evaluators it could be put down to personal quirks.

The proper way to select evaluators is to profile a typical end-user and then solicit the services of individuals who closely fit that profile. A profile should consist of factors such as age, experience, gender, education, prior training and technical expertise.

I love watching developers who take part as observers in usability studies. As a former developer myself I know the hubris that goes along with designing software. In the throes of creation it is difficult for you to conceive that someone else, let alone a user (!), could offer better input to the design process than your highly paid, highly educated self.

Typically developers sit through the performance of the first evaluator and quietly snigger to themselves, attributing the issues to 'finger trouble' or user ineptitude. After the second evaluator finds the same problems the comments become less frequent and when the third user stumbles in the same position they go quiet. By the fourth user they've got a very worried look on their faces and during the fifth pass they're scratching at the glass trying to get into to talk to the user to "find out how to fix the problem".

Other issues that must be considered when conducting a usability study include the ethical nature of the process. Since your are dealing with human subjects in what is essentially a scientific study you need to consider carefully how they are treated. The host must take pains to put them at ease, both to help them remain objective and to eliminate any stress the artificial environment of a usability study might create. You might not realise how traumatic using some software can be for your average user!

Separating them from the observers is a good idea since no-one performs well with a crowd looking over their shoulder. This can be done with a one-way mirror or by putting the users in another room at the end of a video monitor. You should also consider their legal rights and make sure you have their permission to use any materials gathered during the study in further presentations or reports. Finally, confidentiality is usual important in these situations and it is common to ask individuals to sign a Non-Disclosure-Agreement (NDA).

Performance Testing

One important aspect of modern software deployment is its performance in multi-user or multi-tier environments. To test the performance of the software you need to simulate its deployment environment and simulate the traffic that it will receive when it is in use but this can be difficult.

The most obvious solution to the problem of accurately simulating the deployment environment is to simply use the live environment to test the system. This can be costly and potentially risky but it provides the best possible confidence in the system. It may be impossible in situations where the deployment system is constantly in use or is mission critical to other business applications.

If possible however, live system testing provides a level of confidence not possible in other approaches. Testing on the live system takes into account all of the idiosyncrasies of such a system without the need to attempt to replicate them on a test system.

The other common solution to this problem is to use capture-and-playback tools, or automated testing. A capture tool is used to record the actions of a typical user performing a typical task on the system. A playback tool is then used to reproduce the action of that user multiple times simultaneously on multiple machines. The collective, multi-user playback therefore provides an accurate simulation of the stress the real-world system will be placed under.

The use of capture and playback tools must be used with caution, however. Simply repeating the exact same series of actions on the system may not constitute a proper test. Significant amounts of randomisation and variation should be introduced to correctly simulate real-world use.

Note that automated tools are not a panacea for testing. Such tools must be used within a proper, structured testing framework to provide benefit. It is entirely possible to construct tests with automated tools that do not accurately reflect the intended behaviour of your software. By implementing such a tool you may be fooling yourself into a false sense of security since your (erroneously designed) tests will be producing no issues. Simply buying a tool and assuming that your product is perfect because a tool reports your program to be bug-free is stupid.

I know of one instance where a very large multi-national television company launched their "interactive media" services with a live-video web version of their most popular day time soap-opera. Their fit-out was impressive, rack upon rack of dedicated servers and load balancing routers, all to cope with the expected demand. Extensive performance testing of the site indicated that the servers could comfortably handle loads on the order of tens of thousands of simultaneous hits.

On the first night, amongst massive publicity, the site went live and lasted a total of 15 minutes. The peak load was measured at about 65,000 simultaneous users before their site went down and took some telecommunications routers with it. For each of the 65,000 users that got through to the site another five were refused connections. The site and indeed the company never recovered from the embarrassment.

Release Control

When you release a product 'into the wild' you will need to know what you released, to whom and when. This may seem trivial but in a large development environment it can be quite tricky. Take for example a system where ten developers are working in parallel on a piece of code. When you go to build and release a version of the system how do you know what changes have been checked in ? When you release multiple versions of the same system to multiple customers and they start reporting bugs six months later, how do you know which version the error occurs in ? How do you know which version to upgrade them to, to resolve the issue?

Hopefully your version control system will have labelled each component of your code with a unique identifier and when a customer contacts you, you can backtrack to that version of code by identifying which version of the system they have. Further your change control system will let you identify the changes made to that version of the system so that you can identify what caused the problem and rectify it. Your change control system should also record the version of the product that contains the fix and therefore you can upgrade all customers to the correct version.

As usual there are many vendors who will sell you either complete SCM systems or even release control systems that will do all this for you. The simplest solution however could be a spreadsheet or database. This will typically track the following items for each release :

- The version number of the release
- The date of the release
- The purpose of the release (maintenance, bug fix, testing etc)
- For each component within the release
    - the name and version number of the component
    - the date it was last modified
    - some kind of checksum or 'hash' which can be used to confirm the integrity of each module in the release

Sample version control database :

Version	Date	Type	Components			
			Name	Version	Last Mod	Hash
3.0.5.28	27-05-2004	Internal	Kernel	2.3.0.25	25-05-2004	#12AF2363
			GUI	1.0.0.12	27-05-2004	#3BC32355
			Data Model	2.3.0.25	23-05-2004	#AB12001F
			I/O Driver	2.3.0.01	01-04-2004	#B321FFA
3.0.5.29	05-06-2004	Internal	Kernel	2.3.0.28	03-06-2004	#2C44C5A
			GUI	1.0.0.15	01-06-2004	#32464F4
			Data Model	2.3.0.25	23-05-2004	#AB12001F
			I/O Driver	2.3.0.01	01-04-2004	#B321FFA
3.0.5.30	etc…	…	…	…	…	…

(NB: A hash simply a mathematical function that takes the file as input and produces a unique number to identify it. If even a tiny portion of the original changes the hash value will no longer be the same. The hashes in the table above are represented as MD5 hashes in hexadecimal).

Release controls should apply within the team as well as to external releases. When the development team releases a product to the test team, it should follow a controlled process. This allows you to ensure that your dev and test teams are working on the same builds of the product, that defects are being tracked and resolved against a particular build and that there is a minimal lag in such a handover.

Verification and Smoke Testing

An important part of the release control process is the verification of the basic functionality of a particular release. Often errors in the build, assembly or compilation process can result in faults in the delivered release. Spending time identifying and isolating these faults is frustrating and pointless since they are simply artefacts of a bad build.

A release should be verified for basic functionality through the use of 'smoke tests'. That is, before it is passed to any other group the team responsible for building a release should run a series of simple tests to determine that the release is working properly. This catches gross functional errors quickly and prevents any lag in releases. These test can be automated as part of the build/release process and the results simply checked before the release is shipped.

A release should be verified for completeness as well. Often releases are built lacking a particular data or configuration file which is required. Part of the release process should be a simple test which verifies that all of the expected components are present in a release. This can be done by maintaining a separate list of necessary components which can then be automatically compared to a particular build to determine if anything is missing.

Release Notes

One extremely valuable piece of documentation that is often neglected in projects is a set of release notes. Accompanying each release should be a brief set of notes that details (in plain English) the changes made to the system or product with this release.

The major reason for including release notes is to help set expectations with end users. By including a list of 'known issues' with a particular release you can focus attention on important areas of the release and avoid having the same issue reported over and over again.

If the release is a Beta release to tester/customers the notes may also include some notes on how end-users can be expected to log feedback and what kind of information they should provide. If the release is a normal production release then the notes should be more formal and contain legal and license information and information on how to contact support.

Sample Release Notes : ABR SDK v3.0.5.29

Delivery Items

This release image contains three separate archive files:

> **Documentation archive** - this archive contains all of the necessary architectural and installation documentation for the SDK.
>
> **SDK Installation Archive** - this archive is intended to be installed by users who require a desktop version of the application.
>
> **SDK Require Binaries** - an archive containing only the 'executable' portions of the SDK This archive is intended for individuals who intend to integrate the SDK into an application.

Known Issues

This section represents any known issues with the current release of product.

1. Threadpool variation of pool sizes via registry is not enabled

An interim multi-threading model used a number of settings in a registry key that allowed changes to be made to the number of threads available to the SDK. These settings are no longer used and the configuration of the threading model is fixed within the SDK.

2. Removal of ABR.DLL

Part of the pre-processor functionality was contained within a separate DLL for historical reasons – the ABE DLL. Use of this DLL is no longer necessary and builds 3.0.5.28 and later will no longer contain an ABR.dll as a separate file.

3. Removal of status message

As of build 3.0.5.20 the KeepAlive message has been removed. This has been replaced with a progress message upon completion of file processing….

Constructive Cost Model

Numerical models can be used to estimate costs and model the budget and resource requirements for a project. Typically these models revolve around the use of key indicators which are used to drive a simple model and produce an estimate of the quantity required.

An example of a numerical model used in software development is the Constructive Cost Model (COCOMO) developed originally by Barry Boehm in 1981. COCOMO is an empirical model constructed by analysing a large number of existing software projects and developing a formula to fit the data.

The simple COCOMO model or basic COCOMO 81 is shown below:

Complexity	Formula	Description
Generic	$PM = 2.4\ (KDSI)^{B} \times M$	Generic model
Simple	$PM=2.4\ (KDSI)^{1.05} \times M$	Well understood applications developed by small teams
Moderate	$PM=2.4\ (KDSI)^{1.12} \times M$	More complex projects or teams have limited experience
Complex	$PM=2.4\ (KDSI)^{1.20} \times M$	Complex projects with many inter dependencies

Where: PM = Person Months; KDSI = '000s of Delivered Source Instructions; M = multiplier

The basis of the COCOMO model is that the amount of effort involved in a programming project is exponentially proportional to the size of the project (represented by KDSI) and directly proportional to the type of process in use and experience of the team (represented by M).

Note that KDSI has replaced the older thousands-of-lines-of-code (KLOC) measurement. The reason for this is that with the development of newer languages including the 4GL family of software development languages the term "line of code" became imprecise. A single line of code in machine language represents an entirely different level of complexity from that of a single line of code in a third generation language like C or Pascal or a fourth generation language like SQL. KDSI refers to the number of instructions rather than lines of code.

It is important to recognise that COCOMO must be tuned for local variances. By using it in an experimental fashion the user can refine the model and find appropriate values of M to apply. In this way the model becomes self-correcting. It is also important to note that the COCOMO 81 model is based on the standard waterfall model of software development and is not applicable to other models such as RAD.

COCOMO 81 has been largely supplanted by the upgraded COCOMO II model and a host of specialised COCOMO models. COCOMO II uses differing levels of estimation model based on more factors than the size of the project and the relative experience of the team.

PRINCE2

http://www.ogc.gov.uk/prince2/

PRINCE2 is a process heavy project management methodology developed by the Central Computer and Telecommunications Agency (CCTA) now part of the Office of Government Commerce (OGC) in 1989 as a UK Government standard for IT project management. PRINCE2 is suited to large multiple-project programmes in a formal environment.

The model is a highly structured breakdown of project activities and can be more than a little confusing. Given its bureaucratic origins it is no wonder that it is both complex and process heavy. It is best suited to large multi-faceted programs which multiple layers of control. When dealing with government and quasi-government organisations in the UK, PRINCE2 can be considered mandatory but has limited application outside of this sphere.

Dynamic Systems Development Methodology (DSDM)

http://www.dsdm.org/

DSDM is a formalised, structured RAD methodology for software development which was designed by a consortium of 16 companies in the mid 90s. DSDM is based on the concept that most traditional methodologies use a model of the Scope/Quality triangle (see) in which the Quality or Functionality delivered is fixed and it is the Time and Resources available that vary to meet this. DSDM instead assumes that resources and time are fixed (which is a fairly realistic assumption) and that delivery of requirements (functionality) will be defined by these two factors.

DSDM also embodies a number of enlightened principles such as:

- **Active user involvement is imperative in DSDM**. The methodology encourages users to be active participants in the development process. The DSDM model also emphasises that the team, consisting of both users and developers must consult with end-users on design decisions. The focus is on making the team self sufficient and responsible in delivering the project without frequent input from higher management.
- **Iterative and incremental development are essential to** DSDM to allow systems to grow incrementally. This allows developers to make full use of feedback from the users by enabling them to incorporate changes into the growing system. A side effect is that partial solutions can be delivered to satisfy interim client needs.
- **Requirements are baselined at a high level** which means "freezing" and agreeing the purpose and scope of the system at a level which allows for detailed investigation of what the requirements imply. Further, more detailed baselines can be established later in the development, although the scope should not change significantly.
- **Testing is integrated throughout the life-cycle** and is not treated as a separate activity. As the system is developed incrementally, it is also tested and reviewed by both developers and users incrementally to ensure that the development is moving forward not only in the right direction but is technically sound.

Rational Unified Process (RUP)

http://www-306.ibm.com/software/awdtools/rup/

The Rational Unified Process is a software engineering model developed by Rational Software, a software development tools vendor now owned by IBM. RUP is based heavily on the work of the three Rational object-oriented development gurus (Grady Booch; Ivar Jacobsen and James Rumbagh) and includes their work on the Universal Modelling Language.

RUP takes an iterative approach to software development and proposes that development be organised around two axes, content or types of task and time or phases of development. At each phase of development a different mix of tasks is undertaken by the project team and for each phase, different iterations of the content are delivered.

The phases of development proposed in RUP are Inception, Elaboration, Construction and Transition and parallel the five phases of the Iterative model. The tasks undertaken in each phase include business modelling, requirements, analysis and design, implementation, test and deployment. Underlying these primary 'development' tasks are a set of 'management tasks' involving configuration management, project management and environment management.

In each of the phases of development multiple iterations of the software are delivered and the phase concludes with one or more milestones which trigger the transition to the next phase. For example in the inception phase the outcomes include a vision document, a partially complete use-case model, a project plan and one or more prototypes. As work progresses through the development lifecycle the design is revised

Best practices espoused by the RUP are:

- Develop software iteratively
- Manage requirements
- Use component-based architectures
- Visually model software
- Verify software quality
- Control changes to software

Like many software development methodologies RUP has a strong basis in fact that has been obscured by successive levels of detail. The strengths in RUP lie in the detailed design phase, the attention to many fundamentals like change control, testing and requirements and in the comprehensive nature of the model.

The obvious weaknesses in RUP are, in my opinion, a heavy emphasis on business modelling, token support for the iterative nature of development and an obvious but unnecessary emphasis on tools. RUP is also built heavily around the use of component architectures and the reuse of software. While laudable in itself component architectures are not universal and some of the underlying premises of RUP seem to rely on the use of components.

Capability Maturity Model (CMMi)

http://www.sei.cmu.edu/cmmi/

The Capability Maturity Model was developed by the Software Engineering Institute at the behest of the US Department of Defence. The model outlines a series of steps in the growth of a software development organisation as it matures. The model also outlines steps that can be taken to move from one level to another, helping organisations progress. In 2000 the model was upgrade to the CMMi (Integration) model which added layers for new software techniques and other processes.

CMMi outlines the characteristics of successful and stable organisations and the behaviours and processes that embody them. The CMM is frequently invoked by large commercial and government organisations in the search for vendors of developed software and certification in the various levels of the CMM is possible.

The CMM defines five levels of maturity ::

1. Initial – Process is unpredictable, poorly controlled and reactive – there is no process or it isn't followed, development depends on 'heroic' efforts of individuals to succeed.
2. Managed – Process is characterised for projects and is often reactive – basic process is used but often different process for each project. At this level success is achieved through knowledge gained from previous development projects of a similar nature.
3. Defined – Process characterised for the organisation and is proactive – the organisation as a whole has standards and processes that are followed. The process is supported by training and a dedicated team who maintain, enforce and disseminate the process.
4. Quantatively Managed – Process measured and controlled - the process is supported by measurement techniques which are used to refine and direct software development.
5. Optimising – Focus on continuous improvement. Using all of the information gleaned from the previous levels the organisation will implement and evaluate new processes

CMM evokes bipolar opinions in software development circles. My own observations are that the first three levels are reasonably accurate and a good guide to a path for the improvement of a software development organisation but the last two levels are less clear.

Most organisations find themselves at level one or two (or somewhere between) and have extreme difficulty in transitioning to level three. Grasping the nettle and moving from a creative, individual pursuit to a disciplined software engineering process requires widespread culture change.

It has been long acknowledged that the software development industry is, if not chaotic, inconsistent. The industry has long been littered with expensive project failures and disappointment. So the saying goes, "if engineers built buildings the way programmers write programmes, the first woodpecker to come along would destroy civilisation". The CMM may not represent the ultimate truth in software development but I think it illuminates some underlying weaknesses present in many software development organisations.

Sad as it may seem, our clients are not often looking for creative flair and excellence. They would be happy with an incremental but predictable improvement at a known cost. Prudence and consistency are the bywords of modern businesses and CMM is an attempt to deliver that to the software industry.

Agile Development and Extreme Programming (XP)

http://www.extremeprogramming.org/

eXtreme Programming is one of the new breed of "agile" development methods.

In response to the increasingly complicated and process-heavy development methodologies a number of groups started exploring the concept of "agile programming". Agile programming is a shift away from the heavy process formalisms of so called "heavy" methodologies towards a more "people focussed" approach to software development. The intention of agile methodologies is to remove the emphasis on process and speed up the delivery of software by handing creativity back to the developers.

XP is the leading example of agile programming to date. Supporters say XP allows rapid, accurate development which provides software with a minimum of overhead and a maximum of responsiveness. Critics say the lack of structure in XP leads to inefficient or error prone development although concrete proof, either way, is hard to come by.

The iterative development method outlined in this book could be considered an "agile" methodology but some would argue that it contains formalisms from earlier methodologies that would exclude it from this class. My own view is that while XP embodies agile concepts it is loosely structured and *as a methodology* is not agile enough in itself. It suits a particular class of software projects but does not scale well nor is it generic enough for all applications.

Like most agile methodologies XP emphasises customer or user focus as a key to success. By targeting user satisfaction XP aims to focus development efforts on delivering meaningful functionality to the end-user. XP also emphasises other important and valuable concepts such as rapid iteration, unit testing, acceptance testing and re-factoring in design.

Initially, User Stories are captured and requirements are fed into the planning for the first iteration. At the same time initial architectural decisions are made on the technical solution which are fed into the planning process through a "system metaphor". An important XP concept here is the "spike". A spike is in essence a prototype or example in code used to solve a particular user or technical problem. Like prototypes spikes are destined to be thrown away once the design issue is resolved. The planning for the iteration is therefore informed by spikes where "uncertain estimates" for the system are refined via spikes and fed back as "confident estimates".

An iteration is then delivered after unit testing and undergoes acceptance testing. This informs the development of the next iteration and successive versions are delivered. Once a candidate iteration is discovered it is delivered for customer approval and implemented in an incremental fashion.

Other agile techniques include SCRUM (Ken Schwaber and Mike Beedle), Crystal (Alistair Coburn) and Adaptive (Peter Coad, Eric Lefebvre and Jeff De Luca). The Dynamic Systems Development Methodology (DSDM) is also an agile methodology which was introduced earlier.

Cleanroom Software Engineering

Cleanroom software engineering is an incremental software engineering methodology developed at IBM to provide very high standards of verifiable quality. Cleanroom engineering uses formal methodologies (especially statistical methods) and a top-down method to build software products. The term cleanroom refers to the environment in which hardware (integrated circuits) are constructed to minimise defects.

In the basic process for cleanroom engineering small teams design software components using a particular specification methodology known as box structures. Using black-boxes the external, visible behaviour of the software is specified while state boxes identify the internal states and transitions required to achieve black box behaviour and clear boxes define the procedures necessary for the state box transitions. The "correctness" of these various box states are then verified by the development team as a group with unanimous consent being required before progressing to the next module. This technique minimises possible errors in specification and design.

The development team then develops these specifications into actual code and delivers them to a separate certification team. This team repeatedly executes the code under test conditions derived from the original specification. The tests are automated and randomised to cover the input space of the software design and any discrepancies or errors as compared with the specification are noted. The time to failure of the code is recorded and over many successive tests and a Mean Time To Failure (MTTF) can be calculated. MTTF therefore becomes a statistical indicator of the quality of the software release*. Under cleanroom engineering techniques a predefined limit for MTTF is set and software must be below this limit or it must be redesigned or rewritten.

The standard caveat on testing applies however; the quality of your product is only as good as the quality of your testing. It is entirely possible to fool yourself into a false sense of security by performing an inadequate or badly designed set of tests and convince yourself you have a product of quality by a failure to detect bugs. Similarly poorly designed tests in cleanroom engineering will fail to produce an accurate MTTF and hence give a false indicator of quality. This is minimised by the special techniques used in specification and design.

The highly specialised nature of cleanroom engineering produces high quality software but is normally beyond the reach of most commercial software organisations. The principles however can be applied particular in the areas of verifying specification and of the statistical measurement of failure data.

P u r e   T h e o r y

The following is intended as a light hearted examination of some of the more interesting philosophical thoughts that underlie the discussions in this book. These idle thoughts kept me going through more than one endless change management meeting or all night coding sessions. They might make you laugh, they might even make you think.

Chaos theory

Chaos theory was once very much in fashion but has since fallen somewhat from favour. Never-the-less it has a number of interesting things to tell us about complex system.

Chaos theory derives from the work of individuals such as Edward Lorenz in complex mathematical systems. Lorenz discovered in a series of meteorological experiments that surprisingly small changes to the inputs for his system caused significantly large changes in his output. In fact rounding off the 4th decimal place from digits in his input stream (i.e. rounding 0.6541 to 0.654) effectively changed a sunny day into a snowstorm. His output predictions wildly differed after an input difference of only a fraction of one percent.

This is sometimes referred to as the Butterfly Effect in which the flapping of a butterfly's wings in Madagascar could produce a hurricane in Japan. This is somewhat inaccurate as a chaos theoretician would say the butterfly's wings might tip a pre-existing, unstable probability system across the line from non-hurricane to hurricane.

The theory was later expanded to encompass the assertion that, in a chaotic system, *arbitrarily* small changes can have *arbitrarily* large effects. This differs from traditional understanding of a mathematical system in which determining the major inputs of the system is sufficient to predict its behaviour (e.g. Newtonian physics).

The stock exchange is oft cited as an example of a complex system which is chaotic. Although stock price fluctuations are affected by a large number of input conditions the most significant of these could theoretically be determined. As these major input factor stock prices could therefore be accurately predicted, it's just a question of having a sufficiently accurate model and a sufficiently fast computer. Needless to say this has been tried and does not work. Chaos theory asserts that this is because tiny factors below your threshold of observation can ultimately influence the outcome of the stock price. Predicting stock price (consistently) is therefore impossible.

The upshot for those that must manage a complex systems is intriguing. Chaos theory affirms that if a complex system becomes chaotic it becomes inherently unpredictable*. If a system is unpredictable it is uncontrollable. Your job therefore becomes one of holding a project back from the brink of chaos, a situation that will be familiar to many of you. Should a project become chaotic it will be beyond control and the outcome will no longer be predictable.

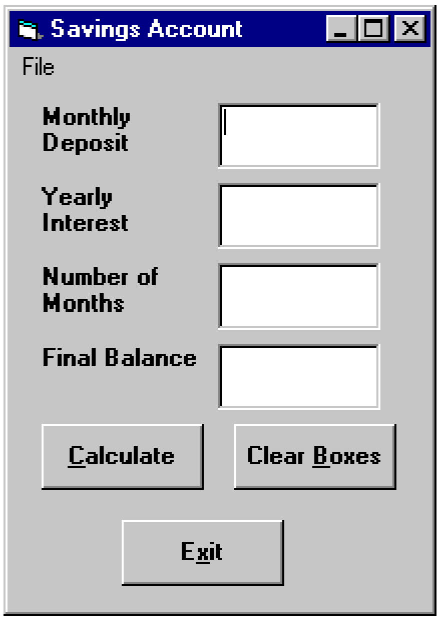
Another interesting corollary might be drawn by the astute reader. A project manager maintains control of a complex project by monitoring certain indicators such as a project schedule and budget. However, since factors below the threshold of a project manager's attention can affect the outcome of the project the question must be asked what value those documents have ? The answer, on the basis of chaos theory, is that there is little value in diving into the fine detail of these indicators since there will always be a level of detail below this which could still affect the outcome of the project. You can't track every second of schedule or every cent in the budget.

Funnily enough this reflects the practical experience of many project managers. Project managers must maintain an adequate distance from the project they are controlling in order to keep their eye on the final goal (not losing sight of the wood for the trees). Project managers who micro-manage their projects tend to lose sight of their goals and let their projects stray off track. (note, the project manager who attributes the failure of his project to a butterfly flapping its wings in Madagascar is probably in for a rude shock and a change of vocation).

*Chaotic systems are not the same as a random system. A chaotic system will be unpredictable, but given the exact same initial conditions, it will produce the same output. In a random system it is extremely unlikely that it will ever produce the same output twice.

Complexity in Software

Many people underestimate the complexity of software. Software, they argue, is simply a set of instructions which a computer must follow to carry out a task. Software however can be unbelievably complex and as, Bruce Sterling puts it, 'protean' or changeable.



For example, the "Savings Account" application pictured at right is a simple application written in Visual Basic which performs simple calculations for an investment. To use the application the user simply fills in three of four of the possible fields and clicks "Calculate". The program then works out the fourth variable.

If we look at the input space alone we can work out its size and hence the number of tests required to achieve "complete" coverage or confidence it works.

Each field can accept a ten digit number. This means that for each field there are 10^{10} possible combinations of integer (without considering negative numbers). Since there are four input fields this means the total number of possible combinations is 10^{40}.

If we were to automate our testing such that we could execute 1000 tests every second it would take approximately 3.17×10^{29} years to complete testing or about ten billion, billion times the life of the universe. (Note I say "four" input fields because we can't assume the product's error handling works either. We must test *all* cases).

An alternative would be to seek 100% confidence through 100% coverage of the code. This would mean making sue we execute each branch or line of code during testing. While this gives a much higher confidence for much less investment it does not and will never provide a 100% confidence. Exercising a single pass of all the lines of code is inadequate since the code will often fail only for certain values of input or output. We need to therefore cover all possible inputs and once again we are back to the total input space and a project schedule which spans billions of years.

There are, of course, alternatives. When considering the input space we can use "equivalence partitioning". This is a logical process by which we can group "like" values and assume that by testing one we have tested them all. For example in a numerical input I could group all positive numbers together and all negative numbers separately and reasonably assume that the software would treat them the same way. I would probably extend my classes to include large numbers, small numbers, fractional numbers and so on but at least I am vastly reducing the input set.

Note however that these decisions are made on the basis of our assumed knowledge of the software, hardware and environment and can be just as flawed as the decision a programmer makes when implementing the code. Woe betide the tester who assumes that $2^{64}-1$ is the same as 2^{64} and only makes one test against them!

The Paradox of Abstraction

Abstraction is, broadly speaking,, the summarising of detail into a simpler form. We do this every day and we do it in project management. At some point you will be faced with a wealth of information which will beyond your capacity to encompass at any single point in time. In order to usefully apply that information you must therefore abstract it down to a simpler form so it becomes useful.

A good example of this is that of a project schedule. If you were to monitor and plot tasks on an hourly basis you would end up with an extremely accurate schedule (errors aside). However your ability to utilise this information would be extremely limited because you simple could not comprehend it all at the same point in time. You could deal with individual parts of the overall schedule at different points in time, but this more or less defeats the point of having a single schedule. You need a summary or an abstract which allows you to understand what's going on without a level of detail that defeats comprehension.

The opposite of abstraction is therefore "holism". In the ideal world you would not only be able to understand the detail of a particular entity but you would also be able to comprehend it as a complete unit without subdivision. In essence you would understand its totality both at a micro level and at the macro level. "Grokking" it, perhaps, in the vernacular of Robert Heinlein. Holism, while a nice idea, is rarely practical for anything complicated.

So we are left with abstraction.

The paradox lies in the fact that while abstraction allows you to gain mental 'traction' with a large or complex topic it forces you into assumptions. By approximating complicated relationships or structures you are assuming that your approximation has the same effect in your model as the original. This is rarely the case. This means that while abstraction offers you the chance to get to grips with a problem, your actual level of control is diminished.

In project management you can think of it in terms of a schedule. It would be ideal to track every task of every hour of every day, in fact every minute of every day would be better. But this is not practical. Instead we deal with schedules that utilise units of days or weeks and are able to therefore grasp the relationships between these gross units that make up the whole schedule. But the cost of doing so is that we lose control of the finer detail. A project task might slip by only an hour but delay the whole project for a month or more.

If you think this is fanciful consider the situation where a major system build slips an hour on a Friday night. The test team who have been waiting for the build go home early and take the weekend off instead of discovering that fatal error in the core code. When they come back on Monday it takes them a couple of hours to warm up and they finally find it at about 3pm that afternoon. By 5pm the impact of the fault finally sinks in and the project manager calls a crisis meeting only to discover the developer responsible for that particular piece of code left for holiday at lunchtime. Throwing extra people at it and rebuilding the system from scratch for a new round of testing solves the problem but chews up a man/week of valuable time that should have been put to better uses. For want of a shoe the battle was lost, etc.

You need to balance your high level view of a project with what is happening on the ground. While you deal with the bigger picture of delivery, milestones and deadlines, you need to have a feel for what is happening at a grass roots level. How are the programmers doing? Have the testers found anything important? Are the stakeholders all happy? Time to cruise the water cooler to find out. As a project manager, you need to deal with abstraction, but you also need to dabble in the detail whenever you can.

Glossary

Acceptance Test	Final functional testing used to evaluate the state of a product and determine its readiness for the end-user. A 'gateway' or 'milestone' which must be passed.
Acceptance Criteria	The criteria by which a product or system is judged at Acceptance . Usually derived from commercial or other requirements.
Alpha	The first version of product where all of the intended functionality has been implemented but interface has not been completed and bugs have not been fixed.
API	Application Program Interface – the elements of a software code library that interacts with other programs.
Beta	The first version of a product where all of the functionality has been implemented and the interface is complete but the product still has problems or defects.
Big-Bang	The implementation of a new system "all at once", differs from incremental in that the transition from old to new is (effectively) instantaneous
Black Box Testing	Testing a product without knowledge of its internal working. Performance is then compared to expected results to verify the operation of the product.
Bottom Up	Building or designing software from elementary building blocks, starting with the smaller elements and evolving into a lager structure. See "Top Down" for contrast.
Checksum	A mathematical function that can be used to determine the corruption of a particular datum. If the datum changes the checksum will be incorrect. Common checksums include odd/even parity and Cyclic Redundancy Check (CRC).
CLI	Command Line Interface – a type of User Interface characterised by the input of commands to a program via the keyboard. Contrast with GUI.
CMM	The Capability Maturity Model – a model for formal description of the five levels of maturity that an organisation can achieve.
Critical Path	The minimum number of tasks which must be completed to successfully conclude a phase or a project
Deliverable	A tangible, physical thing which must be "delivered" or completed at a milestone. The term is used to imply a tactile end-product amongst all the smoke and noise.
DSDM	Dynamic Systems Development Methodology – an agile development methodology developed by a consortium in the UK.
Dynamic Analysis	White box testing techniques which analyse the running, compiled code as it executes. Usually used for memory and performance analysis.
End-user	The poor sap that gets your product when you're finished with it! The people that will actually use your product once it has been developed and implemented.
Feature creep	The development or a product in a piece-by-piece fashion, allowing a gradual implementation of functionality without having the whole thing finished..
Glass Box Testing	Testing with a knowledge of the logic and structure of the code as opposed to "Black Box Testing". Also known as "White Box Testing".
Gold Master	The first version of the software to be considered complete and free of major bugs. Also known as "Release Candidate".
GUI	Graphical User Interface – a type of User Interface which features graphics and icons instead of a keyboard driven Command Line Interface (CLI qqv). Originally known as a WIMP (Windows-Icon-Mouse-Pointer) interface and invented at Xerox PARC / Apple / IBM etc depending on who you believe.
Heuristic	A method of solving a problem which proceeds by trial and error. Used in Usability Engineering to define problems to be attempted by the end-user.
HCI	Human Computer Interaction – the study of the human computer interface and how to make computers more "user friendly".

Incremental	The implementation of a system in a piece-by-piece fashion. Differs from a big-bang approach in that implementation is in parts allowing a transition from old to new.
Kernel	The part of software product that does the internal processing. Usually this part has no interaction with the outside world but relies on other 'parts' like the API and UI.
Milestone	A significant point in a project schedule which denotes the delivery of a significant portion of the project. Normally associated with a particular "deliverable".
MTTF	Mean Time To Failure – the mean time between errors. Used in engineering to measure the reliability of a product. Not useful for predicting individual failures.
Open Source	A development philosophy which promotes the distribution of source code to enhance functionality through the contributions of many independent developers.
Prototype	A model of software which is used to resolve a design decision in the project.
QA	Quality Assurance – the process of preventing defects from entering software through 'best practices'. Not be confused with testing!
Release Candidate	The first version of the software considered fit to be released (pre final testing).
Requirement	A statement of need from a stakeholder identifying a desire to be fulfilled
ROI	"Return On Investment" – a ratio which compares the monetary outlay for a project to the monetary benefit. Typically used to show success of a project.
RUP	Rational Unified Process – a software development methodology focussed on object-oriented development. Developed by the big three at IBM-Rational Corporation.
Scope creep	The relentless tendency of a project to self-inflate and take on more features or functionality than was originally intended. Also known as 'feature creep'.
Shrink wrapped	Software that is designed to be sold "off-the-shelf" and not customised for one user
Show Stopper	A defect that is so serious it literally stops everything. Normally given priority attention until it is resolved. Also known as "critical" issues.
Static analysis	White Box testing techniques which rely on analysing the uncompiled, static source code. Usually involves manual and automated code inspection.
Stakeholder	A representative from the client business or end-user base who has a vested interest in the success of the project and its design
Testing	The process of critically evaluating software to find flaws and fix them and to determine its current state of readiness for release
Top Down	Building or designing software by constructing a high level structure and then filling in gaps in that structure. See "Bottom Up" for contrast
UAT	User Acceptance Test(ing) – using typical end-users in Acceptance Testing (qv). Often a test as to whether a client will accept or reject a 'release candidate' and pay up.
UML	Universal Modelling Language – part of RUP, a diagrammatic process modelling language
Usability	The intrinsic quality of a piece of software which makes users like it. Often described as the quality present in software which *does not annoy* the user.
Usability Testing	User centric testing method used to evaluate design decisions in software by observing typical user reactions to prototype design elements
User Interface	The top 10% of the iceberg. The bit of software that a user actually sees. See CLI and GUI, ifferent from the Kernel or API.
Verification	The process of checking that software does what it was intended to do as per its design. See "Validation". Sometimes posited as "are we making the product right?"
Validation	Checking that the design of a software system or product matches the expectations of users. See "Verification". Sometimes posited as :"are we making the right product?"
White Box Testing	Testing the program with knowledge and understanding of the source code. Usually performed by programmers, see Black Box Testing for contrast.
XP	eXtreme Programming – A form of agile programming methodology