Software Engineering Processes

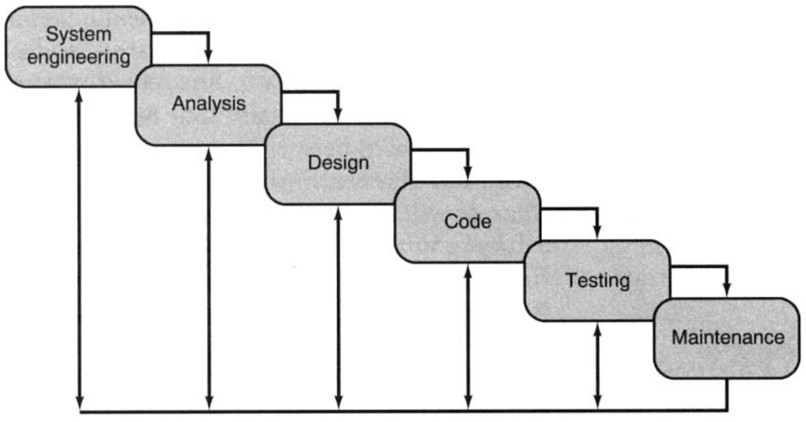
A software engineering *process* is the model chosen for managing the creation of software from initial customer inception to the release of the finished product software. The chosen process usually involves the techniques such as

- Analysis,
- Design,
- Coding,
- Testing and
- Maintenance

Several different process models exist and vary mainly in the frequency, application and implementation of the above techniques, for example, different process models use different analysis techniques, other models attempt to implement the solution to a problem in one big-bang approach, while others adopt an iterative approach whereby successively larger and more complete versions of the software are built with each iteration of the process model.

The Software Engineering Process - The Software Life Cycle

The illustration below highlights the various phases of what is probably the oldest software development process in existence, namely the classic *life-cycle* paradigm, sometimes called the "*waterfall model*". This paradigm implies a systematic, sequential approach (*rarely achieved in practice*) to software development that begins at the system level and progresses through analysis, design, coding, testing and maintenance.



Modelled after the conventional engineering cycle, the life-cycle paradigm encompasses the following activities. Let’s take a look at each of these phases in turn and explain what is involved.

System Engineering and Analysis.

Because software almost always forms part of a much larger system, work begins by establishing not only the role played by the software but, more importantly, its interface and interaction with the outside world.

This '*system view*' is essential when software must interface with other elements such as hardware, **people**, databases and computers outside of, and beyond the control of the system designer. In essence Systems engineering involves exploring some of the following issues:

1. Where does the software solution fit into the overall picture? The software being proposed may be one small cog in a very large wheel. Maybe the software is a calculating employee pay or tax, or perhaps has to co-ordinate the activities of several distributed systems controlling a production/manufacturing plant or warehouse distribution system. Until the overall picture is clear, no analysis of the software can begin.

2. What does the system do? Is it required to control or monitor some process or activity or it simple performing analysis of data?

3. What environment will the system be place in?

    - Hostile, such as a lift controller subject to vibration and dust, or friendly, such as a cosy air-conditioned office?
    - Will the system be '*real time*', 'on-line', '*batch*', '*safety critical*', '*fault tolerant*?'
    - Is it required to be *mobile* (e.g. a cell phone app) or does it require an electricity main outlet?
    - Is it *embedded?*
    - *Does it require a man-machine interface?*
    - *Who* or *what* does it interact with

    All of these factors could severely influence, restrict and or dictate the form of the solution.

4. What inputs and outputs are required/produced and what is the form of that data: Paper, Database, Disk file, Graphics, Network and Analogue to digital converter outputs?

In answering the above question, a list of external devices is uncovered and can be explored further, e.g. what size of paper, what resolution graphics, what format of data is acceptable, what will the output look like, what capacity of disk, what resolution of Analogue to digital converter etc.

System Engineering and Analysis…(*cont*.)

5. Is the system a completely new product, or is it designed to replace a mechanical/human activity? Once this is established, the designer can assess the suitability or otherwise of a software solution to the proposed problem.

6. What user interface is needed and how is it used. For example, mouse, keyboard, buttons on control panel, touch-screen, graphics etc?

7. Does the system impose performance requirements? For example real-time systems often specify maximum response times to events under their control, batch system do not. A database application might require the ability to process 1000 searches/updates per min. Google might impose a 1 sec delay to a search request with 100,000 simultaneous connections per sec - imagine the impact of that on hardware.

8. Does the system interact with other computers and if so, what is the relationship between them in terms of what does each expect of the other?

9. What operating system and or programming languages might be required/imposed?

10. What time schedule has been proposed and how critical is it. What budgets does the customer have and are they realistic. What are the cost/benefits tradeoffs to the user in automating some manual procedure?

Of course once these questions have been answered, the developer is in a good position to assess the risk involved in implementing the system. For example,

- Does the developer have the necessary experience and skills to implement the system or will he/she have to learn a lot of new skills?
- Can the development be carried out with existing staff or will contractors/new staff have to be hired?
- Can the project be completed on time and within budget?

Once the systems engineering and analysis phase has been completed, and a picture of the role the software plays in the overall system has been established, the analysis can then focus specifically on the software and its requirements.

Software Requirements Analysis

Requirements Analysis is the 1st essential step towards creating a specification and a design. Attempting to design a solution to a (perceived) problem without fully understanding the nature and needs of the user and stakeholders will surely end in tears. It has been shown that more than 90% of all software that is rejected by the customer after delivery is done so because the software does not meet the customer *needs*, *expectation*s or *requirements* so it is important to understand these fully. Furthermore, 90% of all money spent on a project relates to the maintenance of it **after** it has been delivered. So what is requirements analysis?

- Requirements analysis is an iterative process conducted jointly by an *analyst* and the *customer* (which includes *domain experts*, *users* and other *stakeholders*) and represents an attempt, to uncover the customer's needs, whilst at the same time assessing the viability of a software solution.

- Analysis provides the designer with a representation of the information processed by the system and the nature of the processing, that is, what does the system do with/to the information it processes. After all a computer can be thought of as nothing more than a system that takes in data, transforms or processes it and produces output.

- Analysis enables the designer to pinpoint the software's function and performance. For example how is analogue data gathered from an A/D converted used to control a manufacturing process? What range of data is acceptable, how fast should the system respond?

- Where the customer is not too sure how the system will eventually behave, the analyst may explore the concept of a *prototype*. This is a part functional model of the software for the customer to assess.

- Where safety critical software is being designed, a more formal specification may be required in terms of mathematical notation such as 'Z' or VDM, so that the resultant code can be proved to comply with the agreed specification.

- The analyst may, where appropriate, require the customer to produce '*verification'* data. That is, data which can be used to test program. The customer would have to provide test inputs and corresponding results, which could be used to assess the correctness of the software.

- Analysis is often complicated by the fact that the customer may only be able to express the problem in terms that he/she understands. That is, they can only express the problem from a '*user's point of view*'. Indeed, they may have little or no understanding of computers or software and may not even have a complete picture of how the system will behave at the initial stages of analysis.

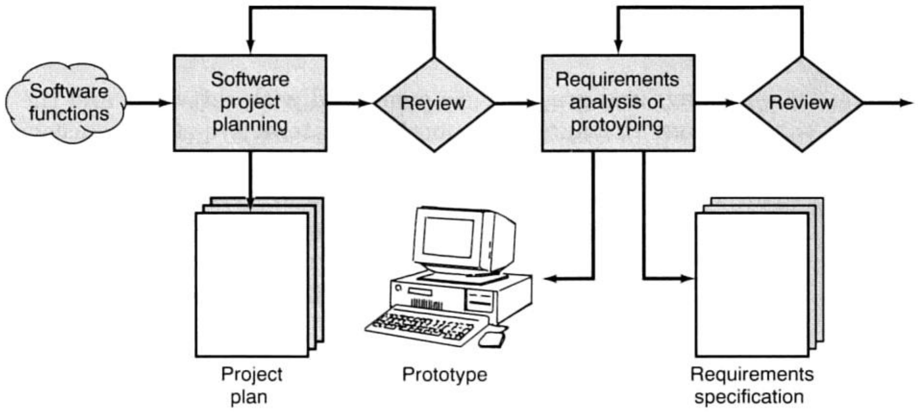
Once the analysis is complete, a *project plan* can be drawn up include, estimates of cost, manpower, resources, time scales etc.

Analysis Summary

The objective of requirements analysis is to create *a "requirements specification document*" that describes in as much detail as possible, exactly what the product should do. This requirements document will then form the basis of the subsequent design phase. The requirements document may well contain

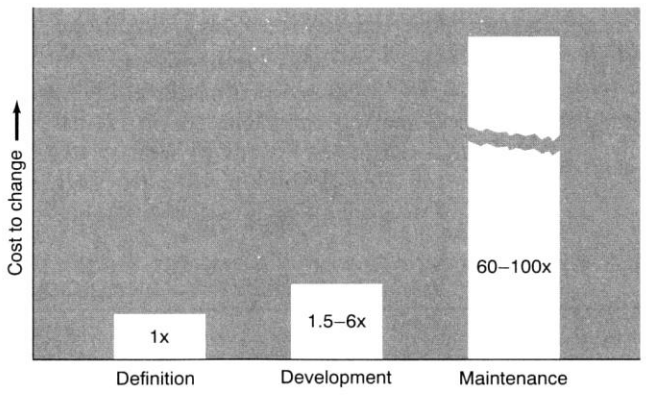
- A Project plan including details of delivery times and cost estimates.
- A model of the software's functionality and behaviour in the form of '*data flow diagrams'* and, where appropriate, *state charts*, any performance and data considerations, any input/output details etc.
- The results of any prototyping so that the appearance of the software and how it should behave can be shown to the designer. This may include a *user's manual*.
- Any formal Z/VDM specifications.
- Any verification test data that may be used to determine that the finished product conforms to the agreed specification.

Analysis then can be summed by the activities in the following diagram


Software functions
Software project planning
Review
Requirements analysis or protoyping
Review
Project plan
Prototype
Requirements specification

The Importance of Correct Analysis

The effects of incorrect or inaccurate analysis can have far reaching/devastating effects on the software time scale, usefulness and cost. This can be seen below, where the cost of maintaining or fixing incorrect software can be 100 times greater than the original cost of getting it right.

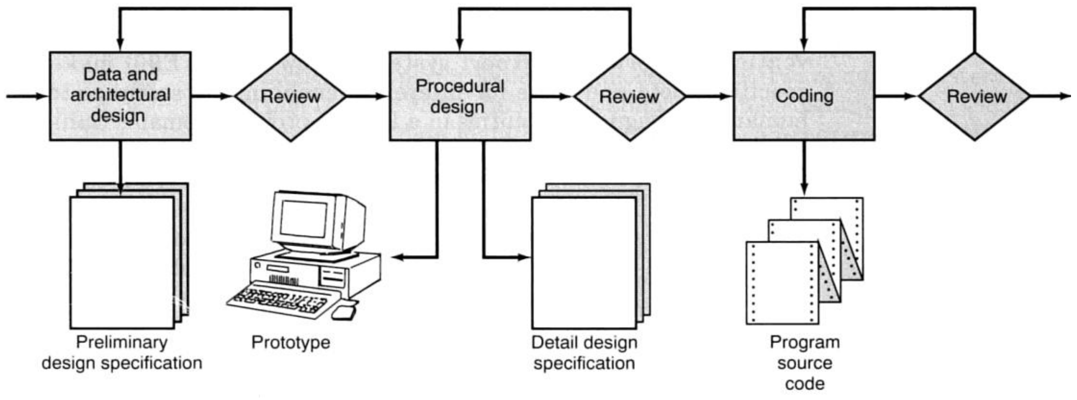


Design and Coding

Once the analysis of the system has been completed, design and development can begin. This is an attempt to translate a set of requirements and program/data models that were laid down in the "*requirements document*" into a well designed and engineering software solution. Design is best summarised by the following sequence of steps

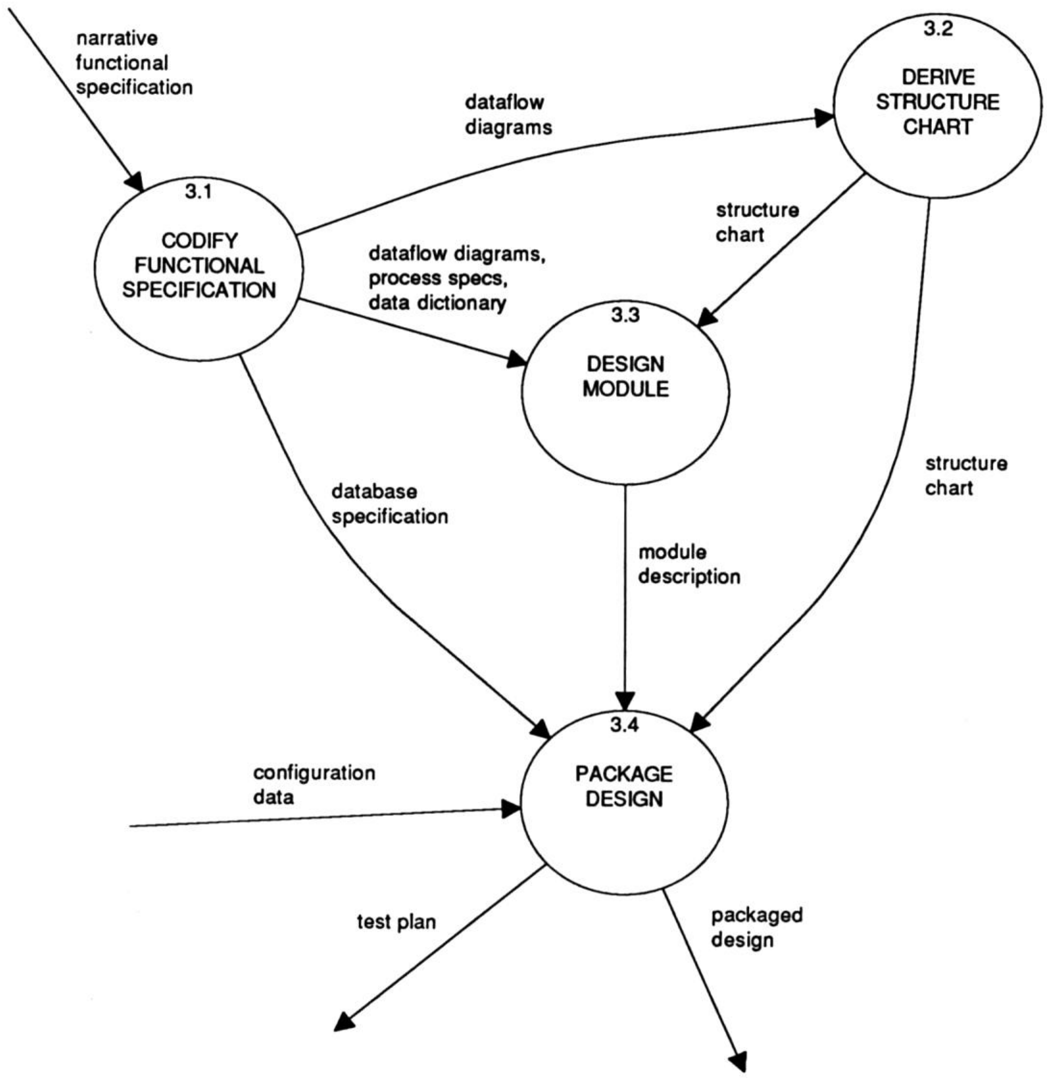
- The data processed by the system is converted into a suitable hierarchical, modular program structure which includes the algorithms used to process that data.
- Each program module is converted into an appropriate cohesive function or subroutine, that is, one designed to perform a single well-defined task.
- Design then focuses on the implementation of each module. The sometimes loose and vague, perhaps English like, description of the modules role/function within the program is expanded and translated into an *algorithm*, which describes in detail exactly what, when and how the modules carries out its task. The module's interface and its interaction with other modules is also considered and assessed for good design (see coupling and cohesion in future lectures).
- The modules algorithms, architecture and data models can then be translated into a chosen *programming language* and the various modules entered, compiled, integrated into a system ready for testing.

At each stage, the process is documented so that if changes are required in the future, the design pertinent to each stage is available for consultation and discussion. The end result of *design and coding* is most definitely **not** just the program listing. It includes the hierarchical program structure, any state charts, flowcharts, pseudocode and all program listings, as shown below.


Data and architectural design
Review
Procedural design
Review
Coding
Review
Preliminary design specification
Prototype
Detail design specification
Program source code

A structured illustration of the design process is shown below. The results of analysis, that is, the target document or specification, feed into the design stage where an attempt is made to '*model*' the system with the aid of data flow diagrams, process specifications and data dictionary entries (all stored in data bases for easy access and maintenance).

From this, a structure chart can be produced to represent the modules (i.e. subroutines/functions) that will be required in the system, which leads to the design of individual modules which can be packaged or integrated to produce a packaged product and, in conjunction with configuration data acquired during analysis, a test plan.



Software Testing and Debugging

Once the constituent software components/modules have been written, testing can begin. Testing involves the following techniques (amongst others)

- *Verification* and *Validation*. That is, checking that the software meets the agreed specification and checking that the software is correct in its operation.
- *Black* and *white* box testing techniques. That is, testing the insides of the modules for correct operation and testing the interfaces to the module.
- *Integration Testing*: Testing that the modules all work together.
- *Acceptance Testing*: Letting the customer test the product.
- *Debugging*: The 'art' of identifying the cause of failure in a piece of software and correcting it.

Software Maintenance

Software maintenance reapplies each of the preceding life cycle steps to an *existing* program rather than a new one in order to correct or add new functionality.

Life Cycle Summary

The classic life cycle model of software development is perhaps the oldest and the most widely used technique for software engineering. However, it has a number of drawbacks:

1. Real projects rarely follow the sequential flow that the classic life cycle model proposes. Iteration always seems to occur and creates problems in the application of the technique. Primarily, because development takes a finite amount of time and the customer needs change during that time. For example, the laws relating to tax are subject to change with every budget, while new competitors and even the company itself may change the way things are done.

2. It is often difficult for the customer to state all requirements explicitly. The classic life cycle requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects.

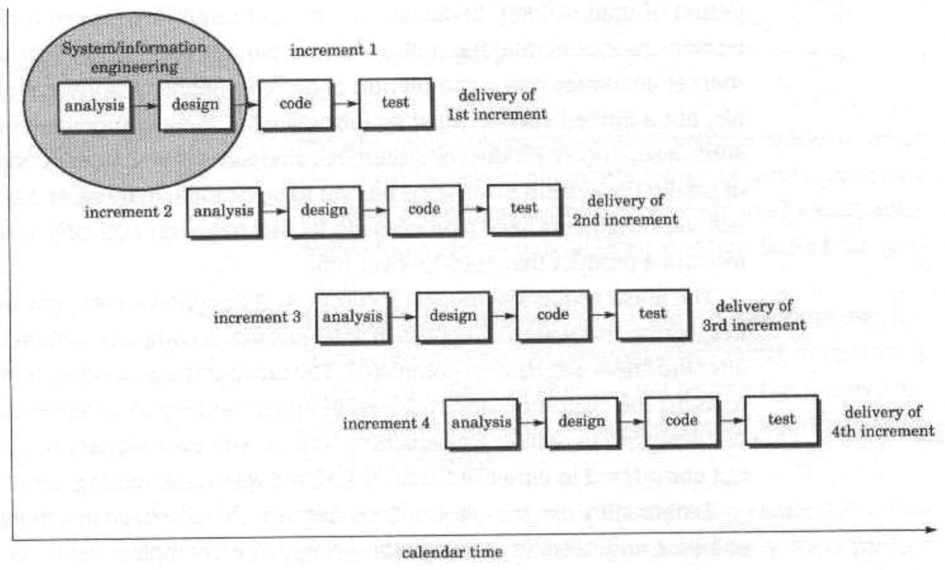
3. The customer must have patience. A working version of the program will not be available until late in the project time span. A major blunder, if undetected until the working program is reviewed, can be disastrous.

However, the classic life-cycle technique has a definite and important place in software engineering work. It provides a template into which methods for analysis, design, coding, testing, and maintenance can be placed and remains the most widely used procedural model for software engineering. While it does have weaknesses, it is significantly better than a haphazard approach to software development. *(See chapter 5, Modern Structured Analysis by Yourdan for more details)*

Other Process Models – The Iterative/Evolutionary Model

There is growing recognition that software doesn't just happen once with the release of a full and finished product, rather it *evolves* over a period of time. All too often this happens during the development of the solution itself where the requirements can change such as the business rules of the company, or the window of opportunity to deploy a specific solution may be small if a company is to open up a lead over competitive products. For this reason, the *big bang* approach to software development proposed by the Software life cycle or waterfall model is probably unrealistic to today's applications

Evolutionary models unlike the classic waterfall model are **iterative** in nature. They are characterised by a process that attempts to engineer software as a series of smaller builds with each build adding progressively more and more functionality to the system. In essence the process is an iterative application of the software life cycle model. The illustration below demonstrates the process.



When an incremental approach is adopted, the 1st increment often concentrates on core feature, that is, the essential workings of the solution, with many fancy features and extras often omitted. For example, the 1st release of a library booking system might concentrate simply on loaning and returning books, while subsequent iterations of the software might for example add facilities to issue fines, reserve a book, produce a profile of a particular persons loan history, search for other libraries with that book etc.

One of the hardest aspects of the iterative model is getting the customer to *prioritise* the functionality so that it can be released iteratively. In other words what do they want first, and what can wait until later? That is can they generate a *must have*, *should have* and *would like to have* list of features.

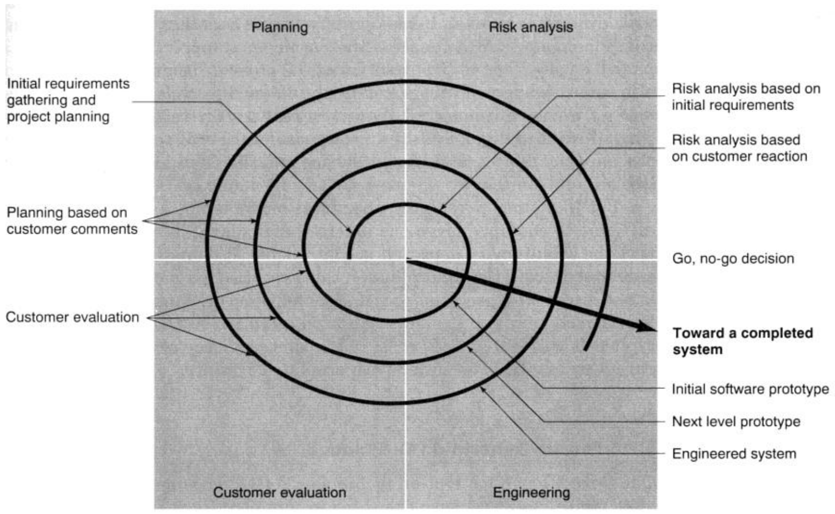
The benefit of the iterative model is the customer gets a core set of functions delivered early and thus can work with them quicker than if he/she waits for delivery of the whole system. Furthermore, software can be customer tested more quickly and thus any obvious errors in say the business models are detected early. Perhaps the best know iterative development processes are typified by the Extreme programming paradigm and the Rational Unified Process (see later)

Iterative/Evolutionary Models – The Spiral Model

The spiral model for software engineering has been developed to encompass the best features of the iterative classic life cycle, while at the same time adding a new element, *risk analysis*. The model, represented by the spiral below, defines four major activities represented by the four quadrants:

1. Planning-- determination of objectives, alternatives and constraints
2. Risk analysis--analysis of alternatives and identification/resolution of risks
3. Engineering-- development of the "next-iteration/level" of the product
4. Customer evaluation -- assessment of the results of engineering

With each iteration around the spiral (beginning at the centre and working outward), progressively more **complete versions** of the software are built. In other words, the product is delivered **not** as one complete monolithic monster, but as a series of iterative development each of which delivers to the customer an executable program comprising progressively more functionality than the previous iteration.


Planning
Risk analysis
Initial requirements gathering and project planning
Planning based on customer comments
Customer evaluation
Risk analysis based on initial requirements
Risk analysis based on customer reaction
Go, no-go decision
Toward a completed system
Initial software prototype
Next level prototype
Engineered system
Customer evaluation
Engineering

During the first circuit around the spiral, objectives, alternatives, and constraints are defined and risks are identified and analysed. If risk analysis indicates that there is uncertainty in requirements, *prototyping* may be used in the engineering quadrant to assist both the developer and the customer. Simulations and other models may be used to further define the problem and refine requirements.

The customer evaluates the engineering work (*the customer evaluation quadrant*) and makes suggestions for modifications. Based on customer input, the next phase of planning and risk analysis occurs. At each loop around the spiral, the culmination of risk analysis results in a "*go, no-go*" decision. If risks are too great, the project can be terminated or re-thought. The spiral model paradigm for software engineering is currently the most realistic approach to the development **for large-scale systems** and software. It uses an *evolutionary* approach to software engineering, enabling the developer and customer to understand and react to risks at each evolutionary level.

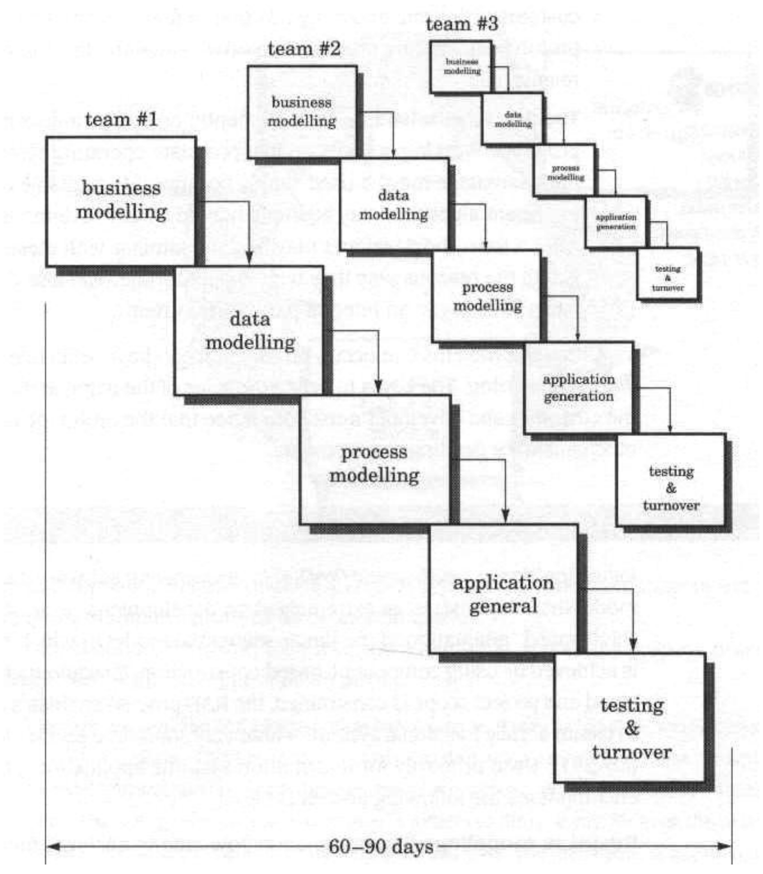
Iterative/Evolutionary Models - The Rapid Application Development (RAD) Model

This particular model emphasizes an extremely short development cycle and is a high speed iterative model. It is particularly (only?) applicable when requirements are well understood by the customer and developer, particularly in the field of Information Systems, for example databases and management information systems.

The techniques rely heavily on 4th generation development tools that allow the developer to capture the system requirements and data model graphically and generate highly specific code targeted at the application area.

A good example of this technique is Microsoft Access database which provides a graphical front end to model data relationships and a code generation tool to produce the Visual Basic or SQL code to run the application.

In essence the developer concentrates only on the code that differentiates one application from another, as much as 90% of the code is common or re-used between applications. For example, the development of a database to record a library booking system might only differ from a database used to record Patient Heart Monitoring services by as little as 10 %.



Agile development

There has long been recognition that some of the older "heavy weight" development methods from the 90's with their emphasis on comprehensive documentation and contract negotiation and rigid systematic approaches to developing software are not always appropriate to modern software development.

For smaller scale projects, where

- Project development times are short
- The development team is small
- The customer is not always sure what they want
- The software is not safety critical

There is an increasing trend to adopt "light weight" development methods commonly referred to as "Agile development"

Agile development represents a group of software development methods in which software requirements and solutions evolve through collaboration between self-organizing teams.

Agile development promotes adaptive planning, evolutionary development, early delivery, continuous improvement, and encourages rapid and flexible response to change.

The Agile Manifesto

In 2001, the agile alliance produced a *Manifesto for Agile Software Development* *http://www.agilealliance.org/the-alliance/the-agile-manifesto/*

The Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

The Agile Manifesto is based on 12 principles:[13]

1. Customer satisfaction by early and continuous delivery of useful software
2. Welcome changing requirements, even late in development
3. Working software is delivered frequently (weeks rather than months)
4. Close, daily cooperation between customers and developers
5. Projects are built around motivated individuals, who should be trusted
6. Face-to-face conversation is the best form of communication (co-location)
7. Working software is the principal measure of progress
8. Sustainable development, able to maintain a constant pace
9. Continuous attention to technical excellence and good design
10. Simplicity—the art of maximizing the amount of work not done—is essential
11. Self-organizing teams
12. Regular adaptation to changing circumstance

Agile Approaches: Extreme Programming (XP)

XP is a fairly recent process model for software development and is summarised in Kent Beck's book. Some of the important (*some might say novel*) aspects of XP are

- It encourages very fast iterative development cycles leading to frequent releases of the code, typically in less than 1 – 2 weeks.
- It attempts to instil into the mind of the programmer the idea that change is not something to be feared. This is based on the idea that object oriented technology has given rise to libraries of components that can be substituted and changed to meet different requirements with minimal knock on effects within the rest of the system. This is analogous to changing a light bulb in your house from 60 watt to 100 watt, the effect is minimal and the effects are limited to the lamp so the change should not be feared.
- It attempts to instil in the programmer the idea that code should not be written until you have a test procedure to validate it against, in other words create your test cases first.
- It encourages frequent rebuilding of the developing code base, sometimes maybe a dozen times a day. The purpose here is to perform almost continuous validation of the developed code against the previously written test suite. This gives encouragement to the programmer that they are doing it right and quickly picks up situations when they are doing it wrong.
- It encourages an approach of "*program for the here and know – not for the future*". In other words, if faced with a problem today, solve it today using the quickest approach you can rather than research three different approaches before selecting

the best. The success of this approach rests totally on the idea that the code can always be restructured and changed quickly (because of object based technology) if the algorithm/approach chosen was not sufficient for the task.

- It encourages the placement of a *customer* on site during development to sit alongside the development team. The idea here is that the customer is a "domain expert", i.e. someone who knows a lot about the business for which we are providing a solution and can act as a sounding board for developers.
- Perhaps the most controversial aspect of the process is that it encourages *pair-programming*, i.e. developers working together rather than individually. Typically one types in the code and is thus focused on syntax, grammar, typing etc, while the other acts on strategy, algorithms and approach. He/she can also observe the other programmer as they type. This approach promotes several things

    - Communication and consideration of alternative strategies.
    - Confidence and trust to try alternatives that they might not feel justified in doing if working alone.
    - Pair learning and debugging, i.e. each can learn from and assist the other

This should all lead to a better solution than if one person alone performed the task (well that's the theory, some cynics say it's paying two people to do one persons job).