CSC 7003 : Basics of Software Engineering

J Paul Gibson, D311

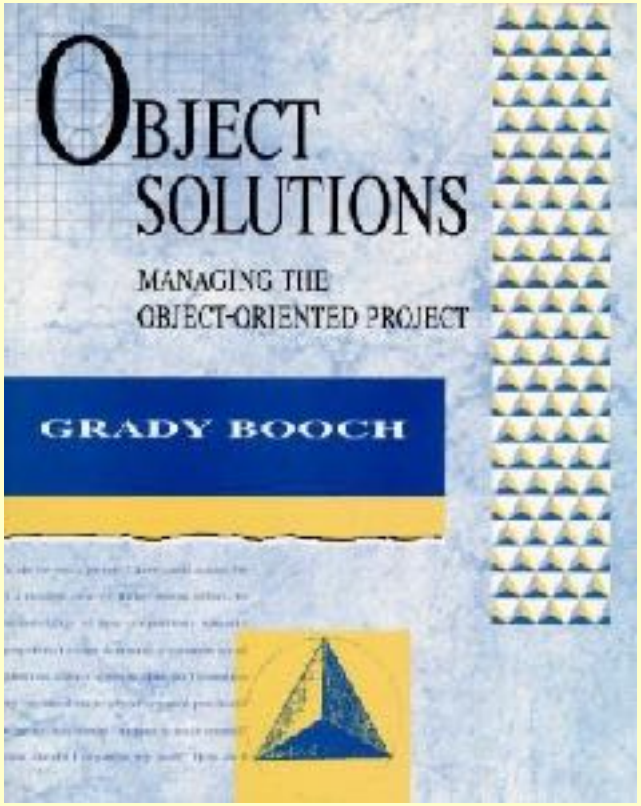
paul.gibson@telecom-sudparis.eu

http://www-public.telecom-sudparis.eu/~gibson/Teaching/CSC7003/

Software Process Life Cycle

~gibson/Teaching/CSC7003/L10-Process.pdf

Recommended text



***Object Solutions: Managing the Object-Oriented Project*, Grady Booch**

Many of our initial recommendations are taken from this text.

Why Do Software Projects Fail (Often) ?

Most often it is because of:

- A failure to properly manage the risks
- Building the wrong thing
- Being blinded by technology

We will concentrate on **software engineering process life cycle**:

Adopting a good *software process life cycle* will help us address these failure modes.

Adopting a good *software process life cycle* does not guarantee success.

We can never have a completely rational development process

Failure to Manage Properly

As projects progress, they often seem to lose their way:

- Unrealistic *schedules* and *plans* are drawn up
- No-one has the nerve to stand up and acknowledge *reality*
- Many problems are viewed as 'a simple *programming* matter', even when they are *process* or *architecture* concerns
- Project *direction* is set by the most *'stubborn'* participants because it is easier for *management* to let these people have their way.
- **Free fall** --- No one takes *responsibility* and everyone waits for the impact.
- Petty empires form … issues become *political*

KEY: Management must actively attack a project's risks

But don't over-manage (the risks)

Failure from Building the Wrong Thing

Projects can also lose their way because they go adrift in completely uncharted territory:

- There is no shared vision of the *problem* being *solved*.
- The (*development*) team is clueless as to the final destination
- No-one takes time to *validate* what is being built with *end-users* or *domain experts*
- Analysts *understand* the real *requirements*, but for a number of political/ social reasons, this understanding never reaches the *designers/ implementers*
- A false air of understanding pervades the *project*.
- Everyone will be shocked when *users* reject the delivered software.
- This is known as **working in a vacuum.**

KEY: Involve real users throughout (development) process

Failure from Being Blinded By Technology

Don't be blinded by the technology being used to build the software itself:

- *Tools* can break (be erroneous) … be ready for it
- Project *complexity* can grow exponentially … can your tools *scale* up accordingly?
- *Third-party suppliers* often do not deliver on promises (if at all)
- *Hardware* advances can out-run *software* development
- *Technology* can fuel changes to users' expectations
- New languages/tools/methods are prone to **premature adoption**

KEY: Do not bind your project/process to any single-source technology (except if it is a known risk which is outweighed by the competitive advantage offered)

No process is perfect

Even the most successful projects seem to take longer, involve more effort, and require more *crisis management* than we really believe they ever should. We must never rely on the process pulling a project through. The process can never be completely rational:

- Users typically don't know what they want
- Users typically can't express what they want
- Requirements are incomplete and/or change
- Implementation architectures change
- We all bring intellectual/technological baggage to projects
- Systems built by humans are always subject to human error
- Fundamental limits to the amount of complexity which can be handled

KEY:Make your process as rational as possible and understand the fundamental limits

Decisions and Compromises

Every software engineering project is about making compromises.

Many steps of development involve technical decisions.

A software process provides a framework to help developers make compromises/decisions in a consistent and coherent fashion.

Strategic decisions have sweeping, perhaps global, implications on the system being developed

Tactical decisions are localised and have little to do with the primary reasons why software projects fail, although they will have impact on strategy.

A good software process allows freedom in tactics whilst supporting more-rigid strategy.

Identifying Primary Characteristics For Strategy

It is impossible for a system to be perfect. At the beginning of a project, we must decide upon the primary characteristics which will drive the project. A good software process lets one evaluate characteristics such as:

- Time-to-market
- Completeness
- Performance
- Quality
- Fault tolerance
- Scale-ability
- Extensibility
- Portability
- Re-usability
- Cost

KEY: The task of software management is to balance a *set of requirements* to produce a system that is optimal for its essential minimal characteristics

Project Styles --- providing a focus

There are many different ways of balancing project characteristics. Certain styles are commonly seen in most industrial projects. These styles correspond to the drive towards a certain focus:

- Calendar-driven
- Requirements-driven
- Documentation-driven
- Architecture-driven
- Quality-driven

KEY: A good software process should help you to focus

Calendar-driven Projects

These are often characterised by an obsessive focus on the *schedule*.

Decisions are made primarily to meet the next deadline and such short-term expediency is usually detrimental to the project.

But, a good process should force the project members to be aware of the importance of schedule without making it too important.

There are times when an obsessive calendar-driven approach is the only solution:

- the organisation will be out of business if project delivery is late
- the delivery deadline is fixed and critical
- if it is the only way to break into a market (typically for a young company)

KEY: A calendar-driven approach should be used only when really necessary

Requirements Driven Projects

Focus on external, observable, behaviour of system:

- Decisions are made primarily according to local needs of each requirement.
- More likely to slip schedules to reach required functionality
- Quality is usually only related to meeting requirements expressed
- Documentation usually adds little other than tracing *what-to-how*
- Little motivation for scale-ability, extensibility, portability or re-usability (unless explicitly stated as a requirement)

KEY: A requirements-driven approach is best when the observable behaviour is well-defined and largely unchanging. It often yields a system that is fairly optimal for a static set of needs.

Documentation Driven Projects

One of the most common/popular, but…

- Generally, not a good way of developing software
- Worst aspects of calendar-driven and requirements-driven
- Management imposes controls on creative aspects of project

Documentation is important, but …

- No more than 5% of project documentation is ever critical
- Gratuitous documentation can be very expensive

KEY: the only time documentation should drive a project is when the documentation is central to requirements (e.g. when developing a library of re-usable components).

Architecture Driven Projects

These represent the most mature style of development… if not the best!

Focus on creating a framework which is both rigid to known requirements and flexible to new requirements.

They are an evolutionary step above requirements driven approaches

Support scale-ability, extensibility, portability, re-usability

KEY: An architecture-driven style is usually the best approach for the creation of most complex software-intensive systems. Object oriented methods are generally accepted as being *the most architecture oriented* ...

Quality Driven Projects - the need for rigour

Focus on quantifiable measures and mostly concentrate on external system-wide behaviours:

- seconds of down time per year/decade
- number of transactions per second
- mean time between failures ...

Other measures may focus on internal, software specific measures:

- 'shape' of class hierarchy
- complexity of individual classes ...

Decisions are usually made to optimise the selected measures.

KEY: Quality is essential in some safety-critical domains and must never be compromised when lives are at stake.

Software Engineering – for *correctness*

- We emphasise *correctness* as being critical to all good software engineering.
- *Correctness* can be built-in through rigorous engineering discipline at all stages of development
- *Correctness* does not come for free
- It is not just an all or nothing choice
- We must know how to engineer to different degrees of confidence in our *correctness,* no matter what else is driving our engineering process

Software Engineering Process Life Cycle

Having a good process depends on:

- Understanding software and software engineering
- Understanding the software life cycle
- Measuring Process Quality
- Improving Process Quality
- Matching life cycle to process
- Managing life cycle (complexity)

Software Engineering and Life Cycle Models

- What is (software) engineering?
- What is a software (engineering) process?
- What is the software life cycle?
- What is a life cycle model?
- Why are these *symbiotic*?

What is Engineering?

Ask any engineer and they will all come up with a different answer; here's mine:

> *Engineering is the construction (or design) of solutions to problems: based on the foundation of science, implemented by the force of technology, aided by experience and fed by intuition.*

Now, we ask '*what is science'*. Again, here is my personal view:

> *Science is the synthesis and analysis of mathematical models, based on observation and experiment, used to capture properties (and define abstractions) of the real world.*

Finally, we ask *'what is mathematics'*. Here is my favourite:

> *Mathematics is speaking the language of nature*

Note: these definitions correspond to actions or processes (verbs). We can easily talk about a body of work (or study) associated with these actions in order to define them as nouns

What is Software Engineering?

The establishment and use of sound engineering principles (methods) in order to obtain economically software that is reliable and works on real machines

(Bauer, F. L. Software Engineering. Information Processing 71., 1972).

> Mr. Bauer was a principal organiser of the 1968 NATO conference that led to the widespread use of the term "software engineering."

(2)*The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software*

The study of approaches as in (2) (IEEE Std 610-1990).

What is Software Engineering ? … continued

the technological and managerial discipline concerned with systematic production and maintenance of software products that are developed and modified on time and within cost estimates

(Fairley, R. *Software Engineering Concepts*. McGraw-Hill, 1985).

the computer science discipline concerned with developing large applications. It covers not only the technical aspects of building software systems, but also management issues, such as directing programming teams, scheduling, and budgeting

(WebReference Webopedia).

What is Software Engineering? …the process view

Engineering is about making useful things, software engineering is about making useful software!?

Software engineering is an approach to various software *lifecycle activities* that emphasises the use of *systematic techniques* to attain specified *quality objectives:*

- **Life cycle activities**: specification, design, implementation, testing, maintenance, evolution, reuse.
- **Systematic techniques**: methodologies of disciplined development, processes involving specified analyses and measurements, repeatability.
- **Quality objectives**:
  - Correctness, reliability, robustness, safety.
  - Security, privacy.
  - Performance, economy.
  - Usability (user-friendliness), predictability, interoperability.
  - Portability, modifiability, adaptability.
  - Reusability.

Software engineering: problem solving (using a computer)

What	How
Problem	Solution
Informal	Formal
Abstract	Concrete

KEY: going from left to right requires structure, communication and transformation

What is the Software Process?

A process is a systematic approach performed to achieve a specific purpose.

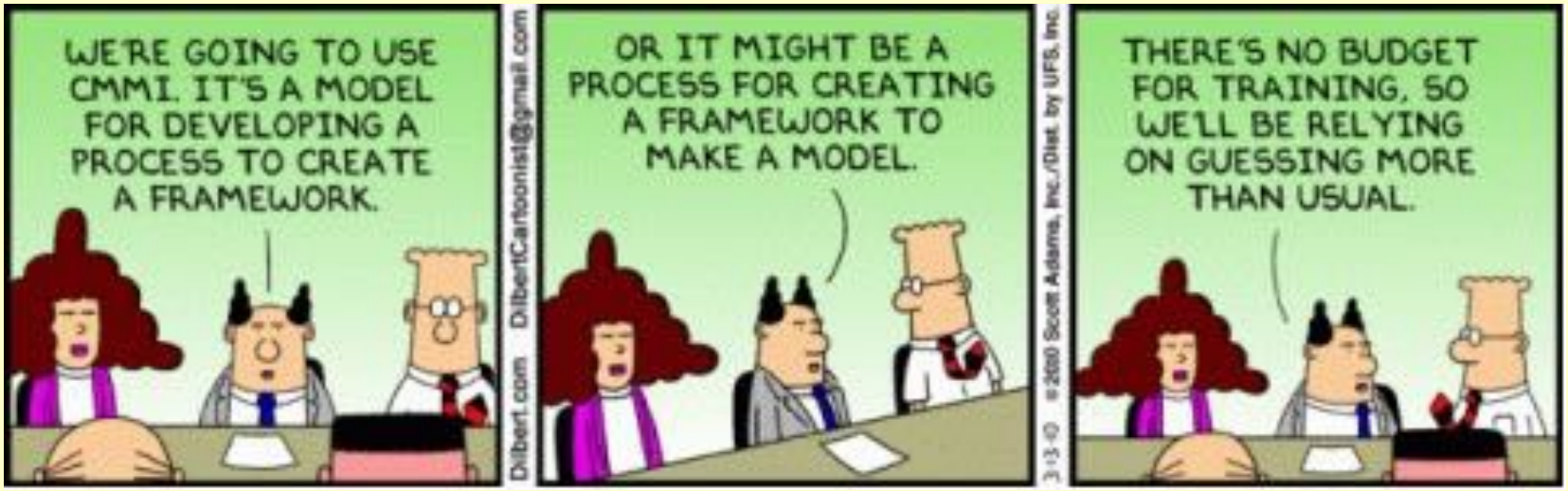
A software process is the set of activities, methods, practices, and transformations used to develop software and associated products that are released with it.

Software Process Capability is the range of expected results that are achievable by following the software process.

Software process performance is the actual result achieved in the development of software by following a software process.

Capability Maturity Model Integration (CMMI)

Software Process Maturity is the extent to which a Software Process is defined, managed, controlled, measured and effective.



What is the Software Process? … continued

There is a Capability Maturity Model (CMM) for Software Process where the SEI (Software Engineering Institute) describes 5 levels of maturity:

- Initial
- Repeatable
- Defined
- Managed
- Optimising

We will (briefly) look at each of these in turn. The categorisation gives some insight into what a software process is trying to achieve.

CMM: Level 1 - Initial

The lowest level:

- software process is ad hoc
- few formalisms
- managers fly by gut feel and seat-of-pants
- success depends on individual efforts and heroics

The capability is a characteristic of individuals, not the organisation.

CMM: Level 2 - Repeatable

- basic process are established
- basic software management controls
- procedures to implement policies
- a process discipline is in place
- realistic commitments
- earlier success can be repeated on similar applications

The process is disciplined.

CMM: Level 3 - Defined Level

- management practices are defined and documented
- peer reviews, inter-group co-ordination, training
- engineering practices are defined and documented
- practices are integrated into a standard process
- all projects use a tailored form from the generic model of project management

The capability of level 3 institutions is *standard consistent* as the process is stable and repeatable

CMM: Level 4: Managed Level

- Measures of the software process and product quality are employed.
- The software products and process are quantitatively understood and controlled.

Organisations are quantified and predictable because the process is measured.

The firm can predict trends in process quality.

The process is understood and variations analysed, understood and correct.

Products are of predictable high quality.

CMM: Level 5: Optimising Level

Improvements in SPM suggested by previous development are fed back into the generic SPM in use.

New technologies and methodologies are piloted and incorporated if successful.

Defect prevention methods are in place.

Such firms are adaptive with continually improving capabilities.

Automating the Software Process --- Tools

Software process automation has much in common with the field of business process automation

Workflow and workgroup products can be used in both fields, and some commercial products are targeted to both markets

Apart from the fact that the applications tools used will not always be the same in both areas, the complexity of interactions between tools and the complexity of processes in the software field are likely to be more pronounced.

Thus, we most look at the underlying life cycle which is unique to software, and analysis the modelling of this life cycle as a means of incorporating our understanding in the process tools

What is the Software Life Cycle?

The software life cycle is the collection of phases through which a software product passes from initial conception through to retirement from service.

- Every software product has a life cycle.
- Life cycles are typically quite long—some software products have been "alive" for 30 years.

Life Cycle Phases - Implicitly or explicitly, all software products go through at least the following phases:

- Requirements—determine customer needs and product constraints
- Design—determine the structure/organisation of the software system
- Coding—write the software
- Testing—exercise the system to find and remove defects
- Maintenance—correct and enhance product after customer deployment

Software Life Cycle Models

A process is a collection of activities, with well-defined inputs and outputs, for accomplishing some task.

A life cycle model is a description of a process for carrying a software product through all or part of its life cycle.

- Life cycle models tend to focus on major life cycle phases and their relationships to one another.
- Recent work on software processes has examined many aspects of development and maintenance in great detail.
- A life cycle model is a software process description, but the term life cycle model predates recent discussions of software processes.

Life Cycle Models and the Software Process

The core of any software project is the coding ---architecture, abstraction, implementation

Life cycle models revolve around this core --- how does the software evolve as the project progresses?

All life-cycle models are based on the simple idea of feedback --- synthesis and analysis are mutually defined and recursively interdependent.

The differences between the life-cycle models lie in the ways in which the feedback is organised.

Note: the non-core (support) aspects of software process are also open to feedback … as are most complex systems

Trial and Error

•The most primitive life cycle model is trial and error, sometimes called build-and-fix or hack-and-foist

•In this life cycle model, the first version of the system is built without planning, documentation, or control

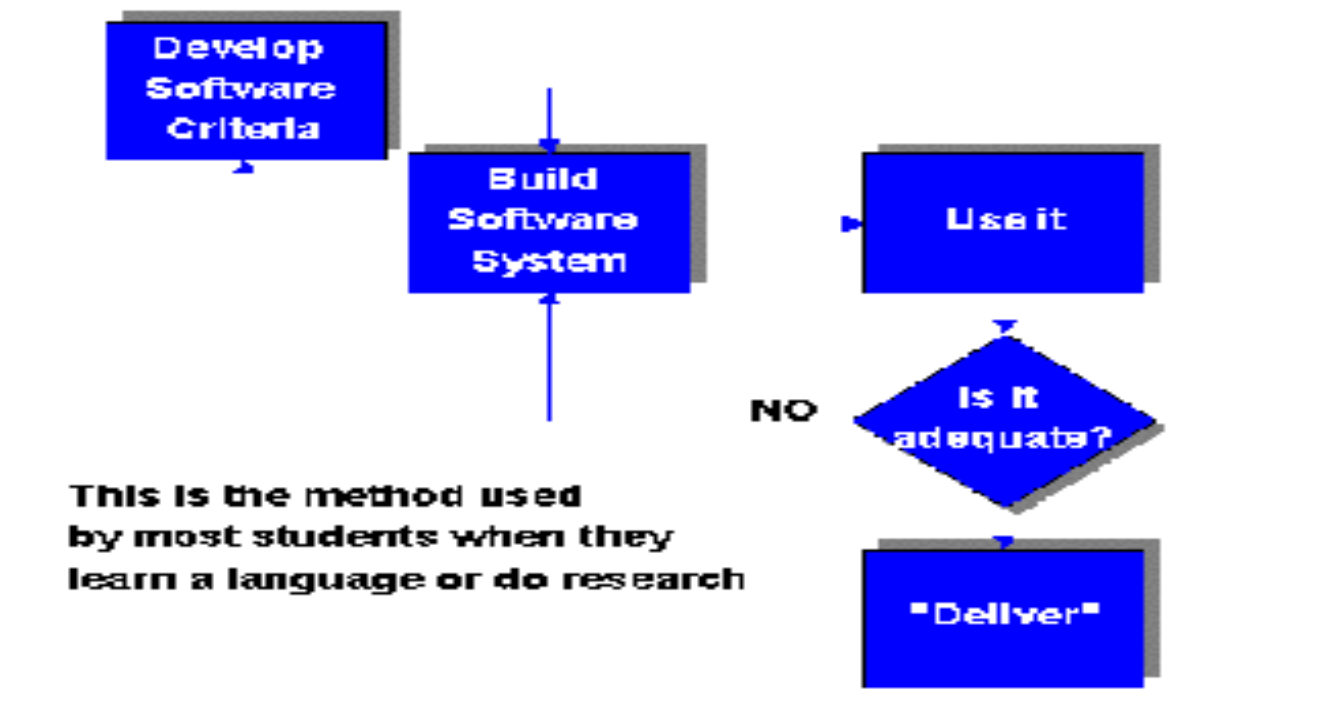
•If the product is accepted, the developers face an interminable period of confusion, frustration, and drudgery fixing an endless stream of problems

The feedback can be very primitive --- will we accept the first and only version of the system (yes/no)

Exploratory Programming

A bit better than trial-and-error (but not much):

- it establishes feedback before delivery to customer
- it allows multiple feedback
- it separates specification from implementation



The Waterfall Model

The waterfall model is the oldest life cycle model; is was proposed by Winston Royce in 1970.

This model is called a waterfall because it is usually drawn with a cascade of activities through the phases of the life cycle "downhill" from left to right:

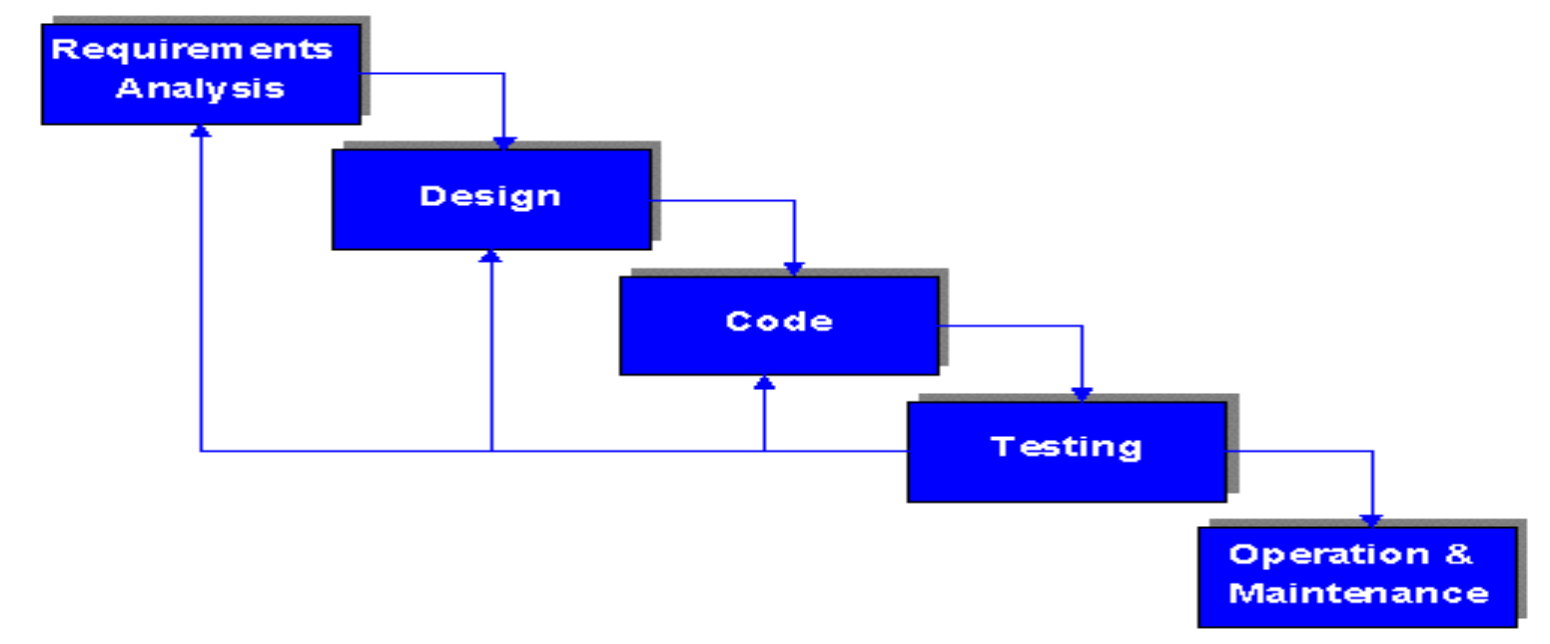
- analysis, requirements, specification, design, implementation, testing, maintenance

There are many versions of the waterfall model:

- the phases can be structured to different levels of detail
- the feedback can be more or less flexible

Non-strict Waterfall Model

Although the waterfall model stresses a linear sequence of phases, in fact there is in practice always an enormous amount of iteration back to earlier phases, a point made by the arrows leading back up the waterfall, in the following diagram.



Note: feedback is only from **testing** phase to *any* previous stage

Analysis of waterfall method

Strengths:

- Emphasises completion of one phase before moving on
- Emphasises early planning, customer input, and design
- Emphasises testing as an integral part of the life cycle
- Provides quality gates at each life cycle phase

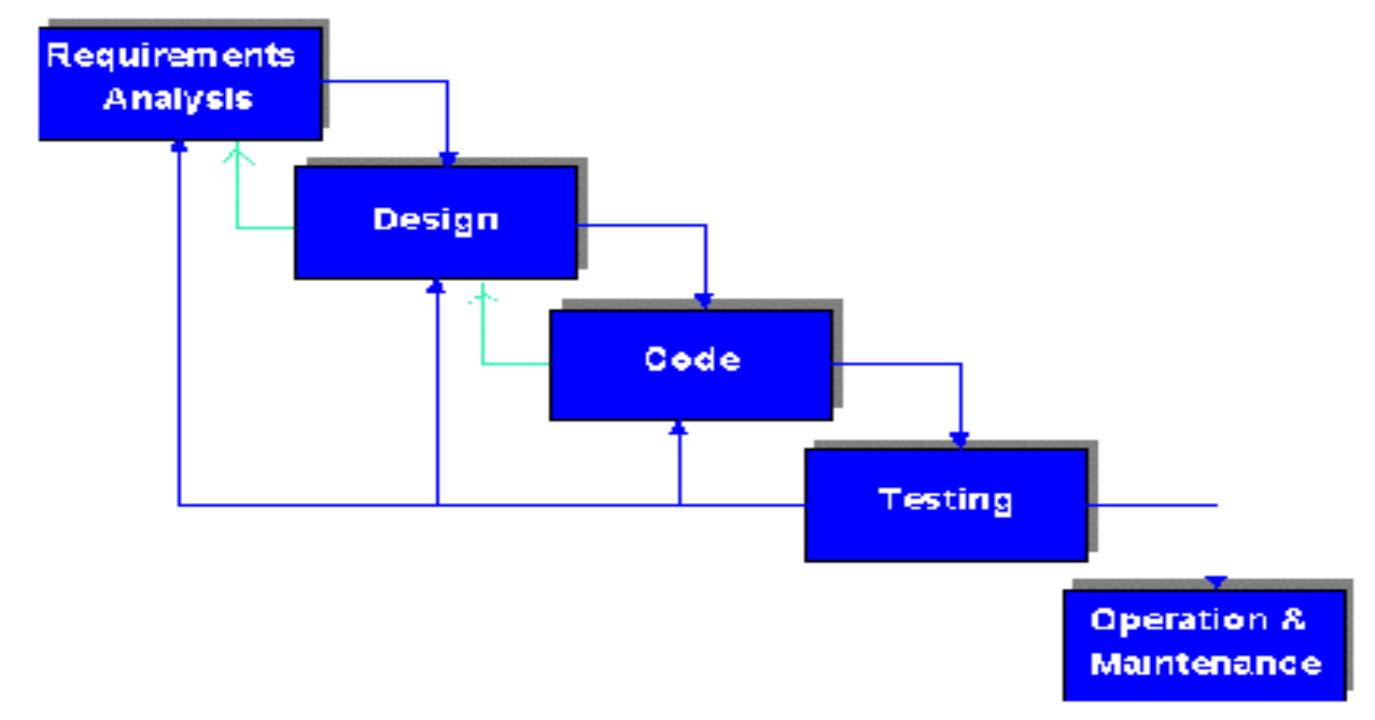
Weaknesses:

- Depends on capturing and freezing requirements early in the life cycle
- Depends on separating requirements from design
- Not politically feasible in some organisations
- Emphasises products rather than processes

Iterative Feedback Model

Like the waterfall method except that feedback is allowed from any phase to the previous phase.

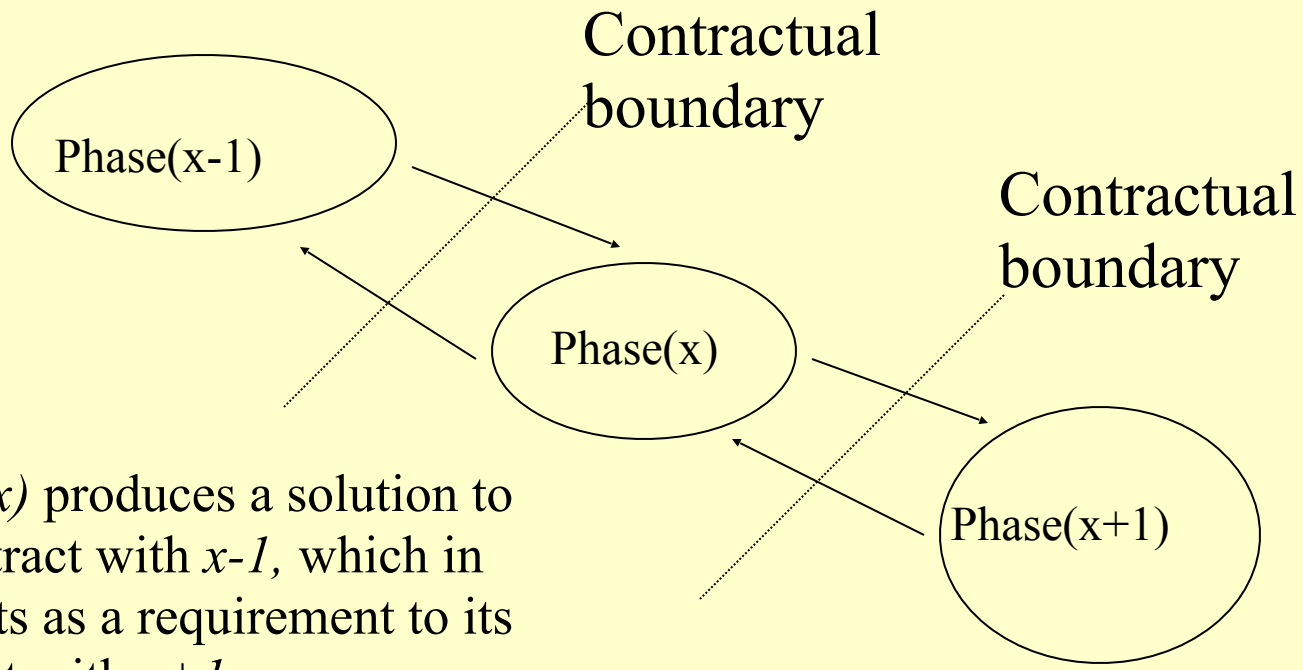
Note that we can still jump anywhere from testing!



Contractual Models: a more formal approach

Like the iterative method except that feedback is allowed only from any phase to the previous phase.

Note that we can no longer jump anywhere from testing!



Phase*(x)* produces a solution to its contract with *x-1,* which in turn acts as a requirement to its contract with *x+1*

Prototyping Models

A prototype is a working model of (part of) a final system.
Prototypes can be used in two ways:

- in the requirements or design phases of a waterfall model, called throwaway prototyping or rapid prototyping
- in a prototype evolution model, also called iterative enhancement, incremental development, or exploratory programming

Prototyping is becoming more popular all the time, and people often refer to prototypes in the literature.

Unfortunately, a variety of terminology is used, so it is often difficult to tell what is meant when people discuss prototyping.

Note also that rapid prototyping and prototype evolution are very different uses of prototypes.

Rapid Prototyping

Strengths:

- Requirements can be set earlier and more reliably
- Requirements can be communicated more clearly and completely
- between developers and clients
- Requirements and design options can be investigated quickly and cheaply
- More requirements and design faults are caught early

Weaknesses:

- Requires a rapid prototyping tool and expertise in using it–a cost for the development organisation
- The prototype may become the production system

Prototype Evolution Model

In a prototype evolution life cycle, an initial set of requirements is used to build a rough prototype than can then be evaluated by clients and developers alike. Feedback is then used to fix and extend the requirements,followed by revision of the original prototype. This cycle can continue as long as necessary, ending only when a satisfactory product has been developed.

Strengths:

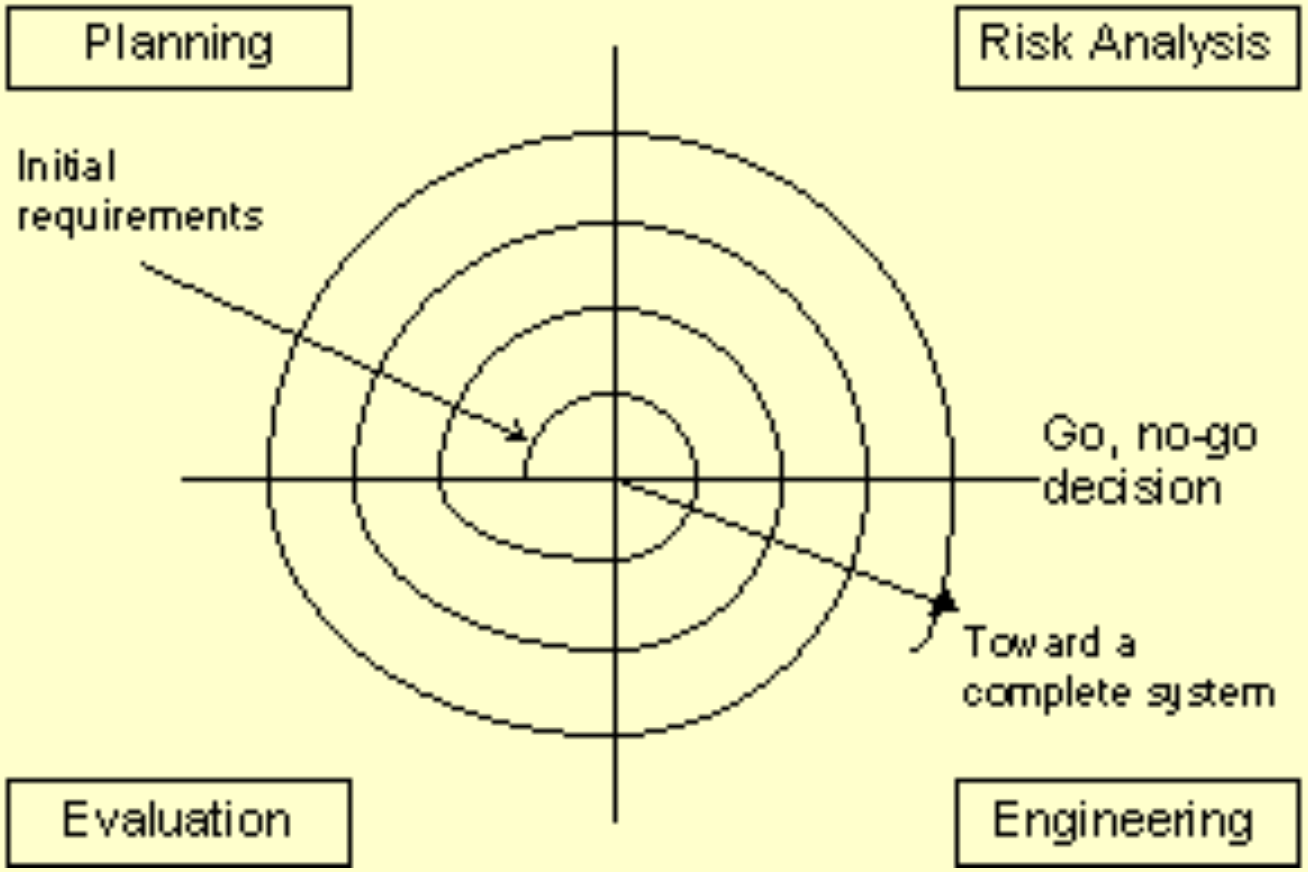
- Avoids building systems to bad requirements
- Delivers a working system early and cheaply
- Fits top-down implementation and testing strategies
- Probably improves developer productivity

Weaknesses:

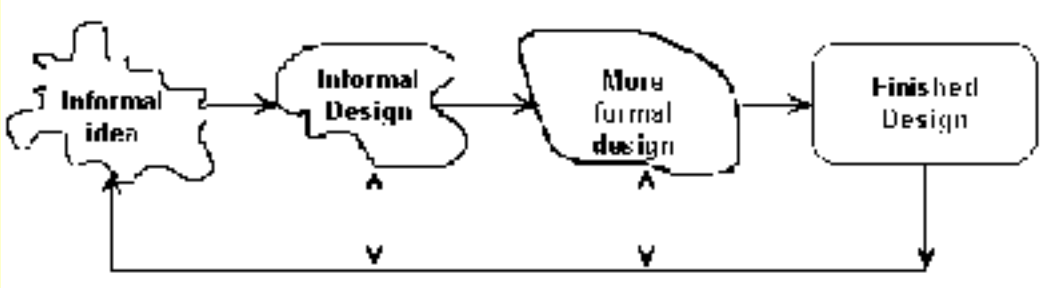
- Users may become frustrated:
- The process may collapse into trial & error
- It is difficult to measure progress or estimate completion times
- Systems tend to develop poor design structures, leading to maintenance problems

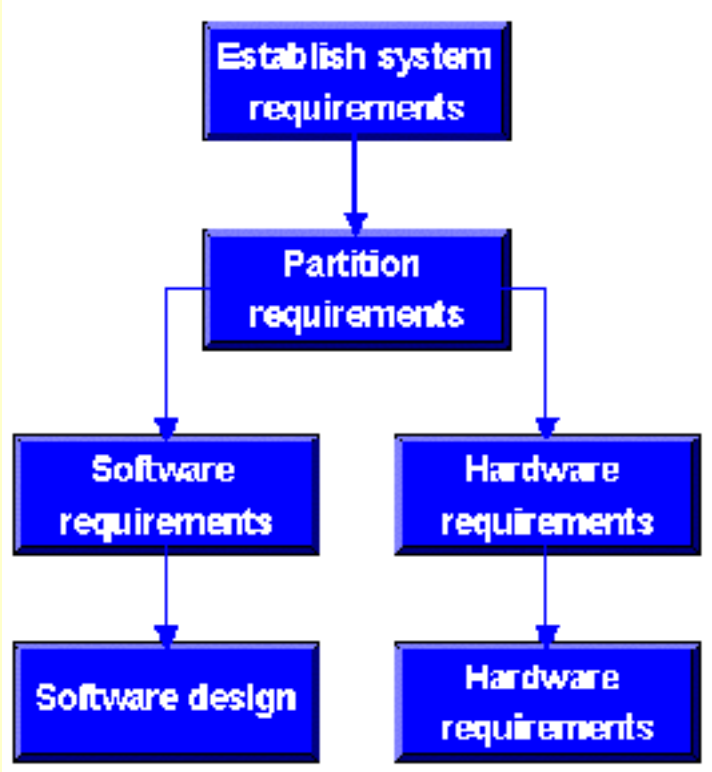
Generic Spiral Model

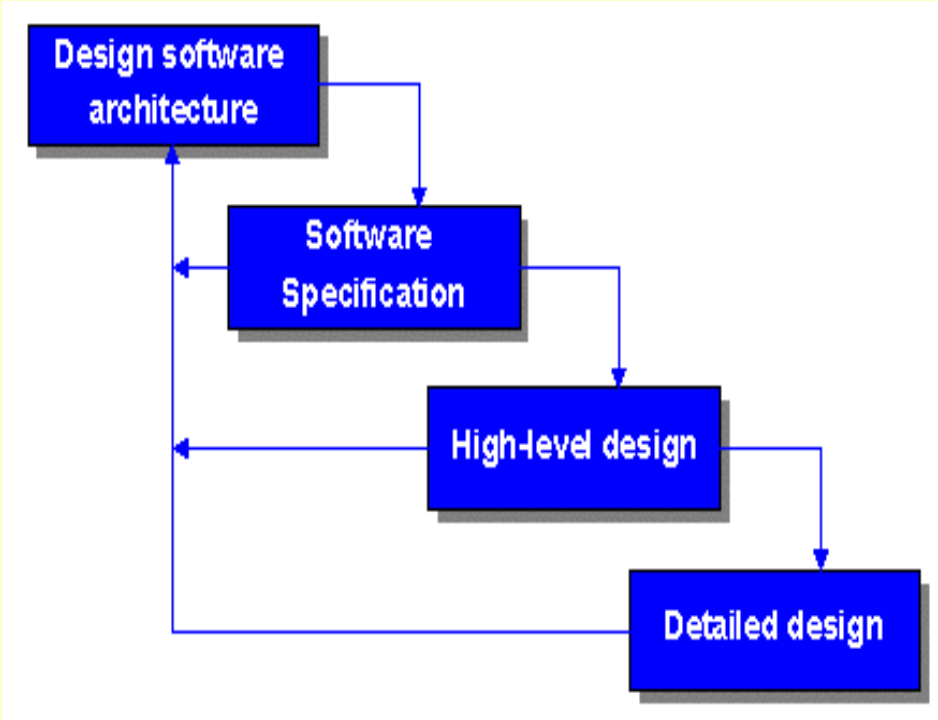
Barry W Boehm



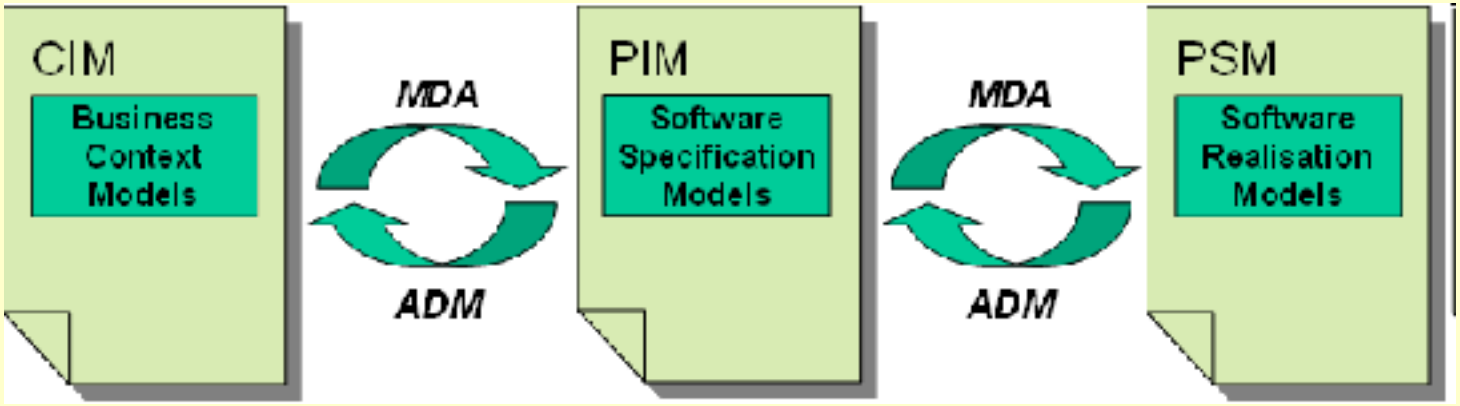
Design Oriented Process: because everything is design







Model Driven Development/Architecture (MDD/MDA)



Code Generation

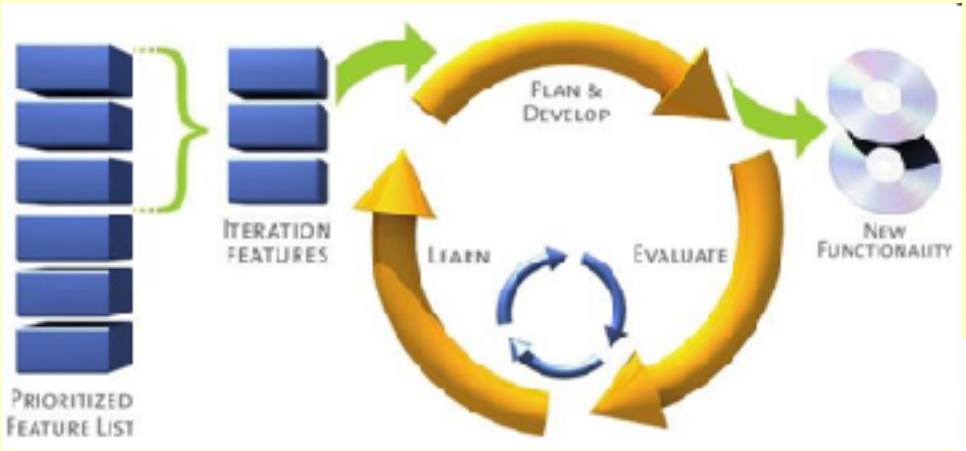
Computation Independent Model (CIM)

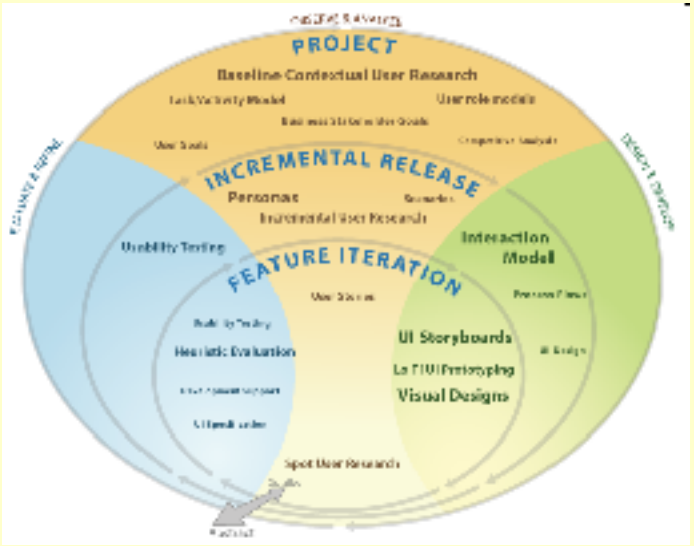
Platform Independent Model (PIM)

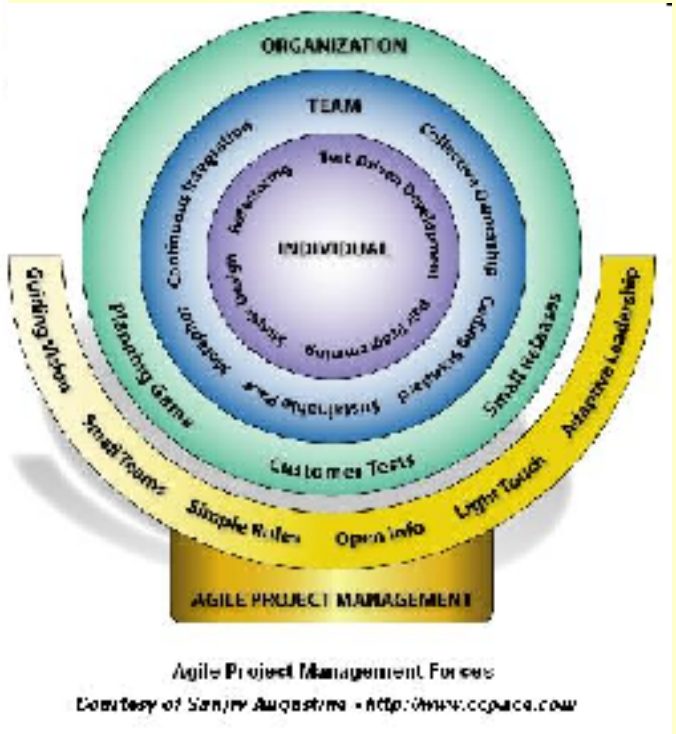
Platform Specific Model (PSM)

Agile Methods

The chief aim is that each *short* iteration produces a *functional, usable piece of software*



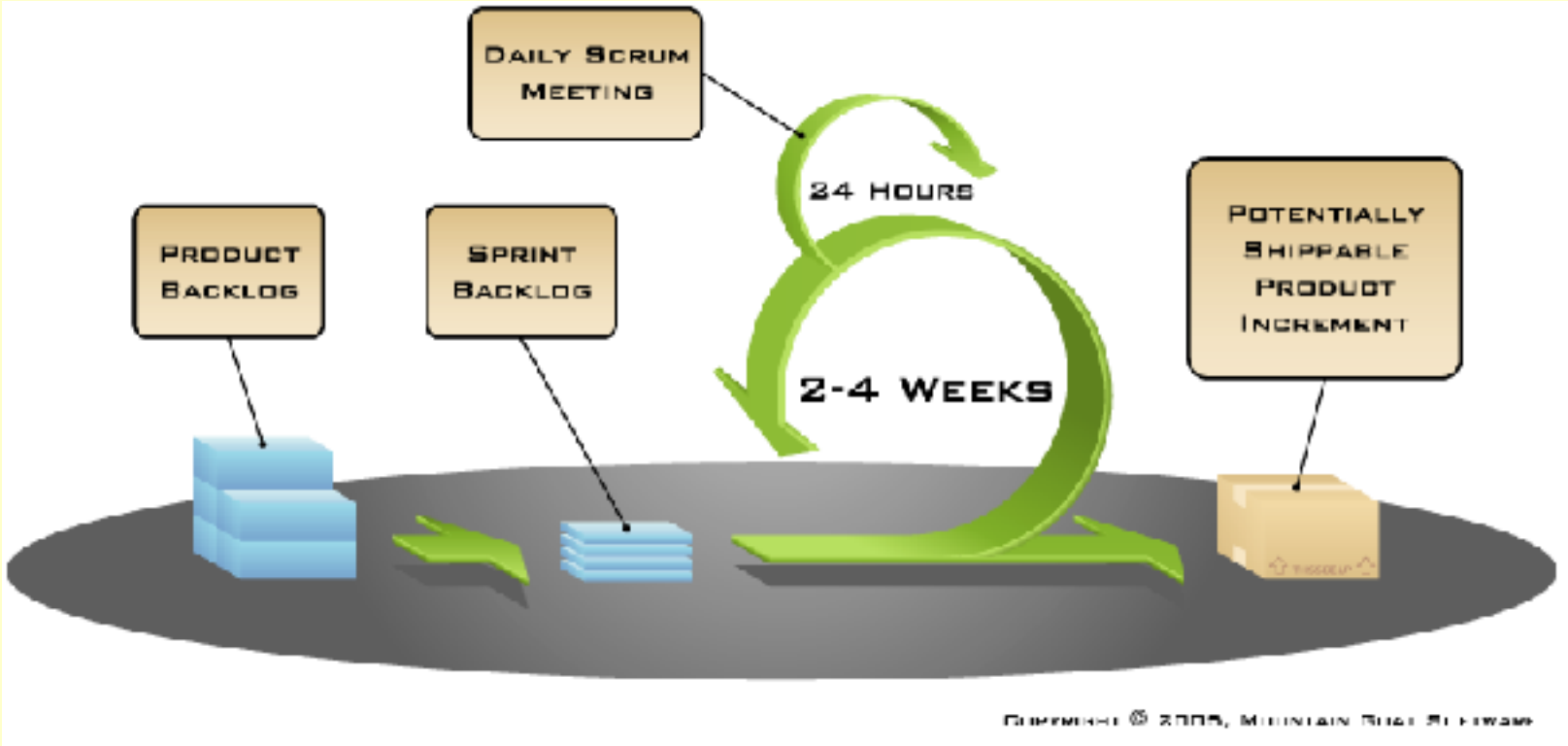




Agile Project Management Forces

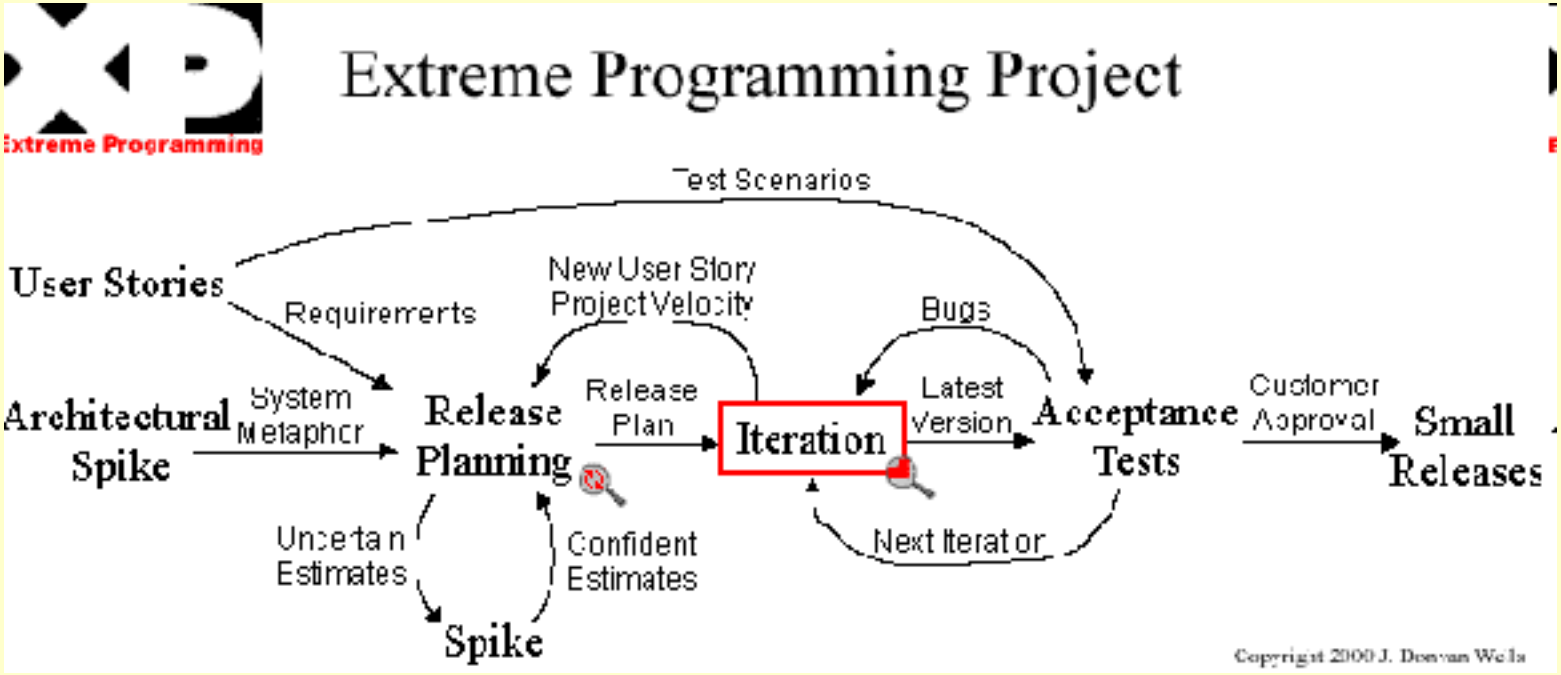
Courtesy of Sanjiv Augustine - http://www.ccpace.com

Agile Examples: Scrum

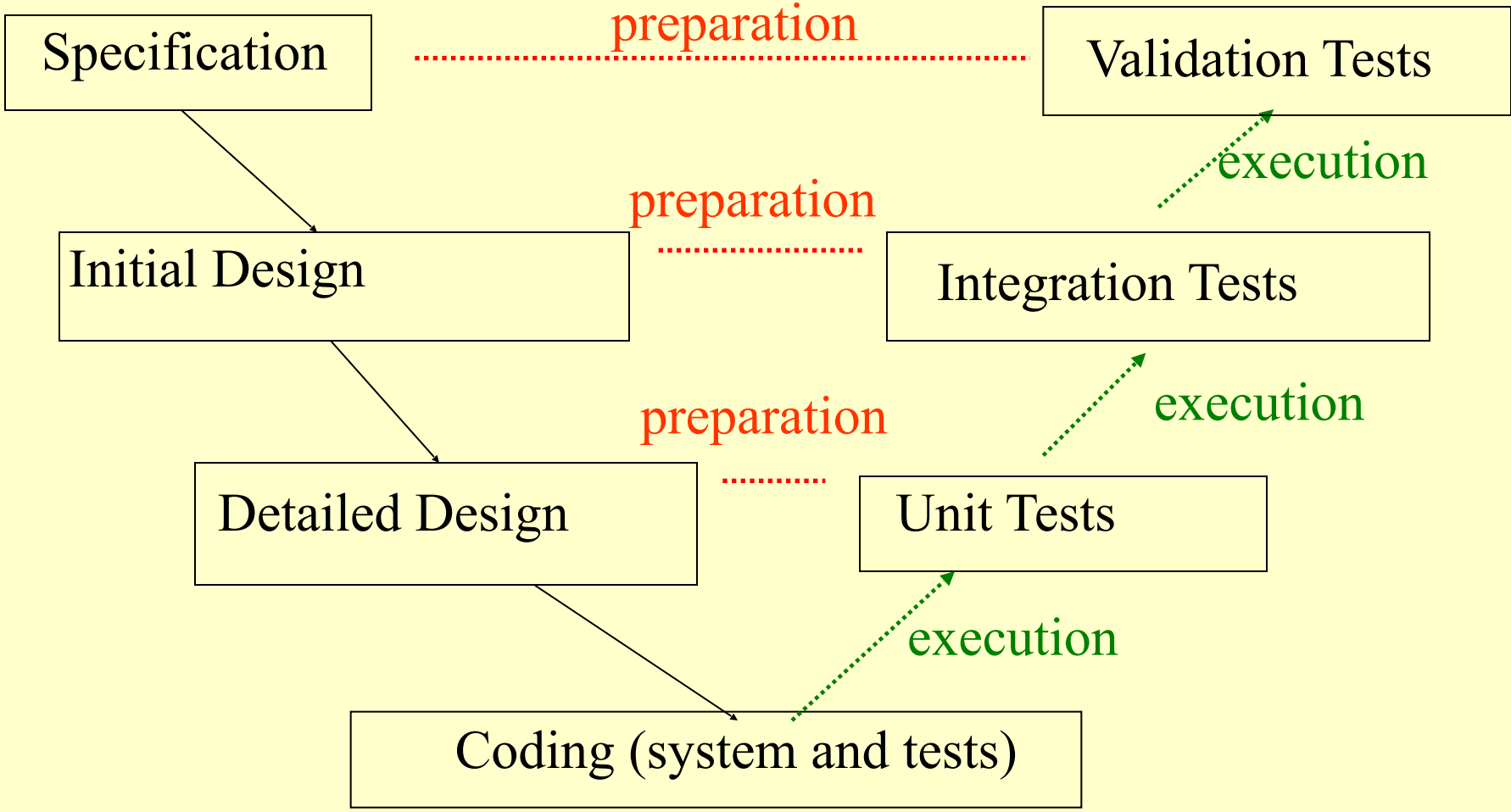


Agile Examples: Extreme Programming (XP)

Agile Examples: Extreme Programming



Tests Driven Process: Build tests in parallel with code



Life Cycle Model Complexities:

We have looked at models which are abstractions of what actually happens in the real world. As for all abstractions we try to emphasise what is important and hide what is irrelevant. But, this process is never perfect. So we should bear in mind the following aspects of the process which may complicate our life-cycle models:

- Working in parallel – Versioning and Change management
- Re-use – Versioning and Change Managment
- Top-down, bottom-up, inside out, …

NOTE: never forget that the people – not the machines/ procedures - are the key components of the process. Managing people is difficult/complex!