# Introduction to Software Engineering

Kyle E. De Freitas
Instructor
Department of Computing & Information Technology
University of the West Indies

# Software Engineering

- Software engineering is an engineering discipline that is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use.
- Engineering discipline
  - Using appropriate **theories** and **methods** to **solve problems** bearing in mind organizational and financial **constraints**.
- All aspects of software production
  - Not just **technical** process of development. Also project management and the development of **tools**, **methods, documentation** etc. to support software production.

# Software Engineering

| Question | Answer |
| --- | --- |
| What is software? | Computer programs and associated documentation. Software products may be developed for a particular customer or may be developed for a general market. |
| What are the attributes of good software? | Good software should deliver the required functionality and performance to the user and should be **maintainable**, **dependable** and **usable**. |
| What is software engineering? | Software engineering is an engineering discipline that is concerned with all aspects of software production. |
| *What are the fundamental software engineering activities?* ** | *Software specification, software development, software validation and software evolution.* |
| What is the difference between software engineering and computer science? | Computer science focuses on **theory and fundamentals**; software engineering is concerned with the p**racticalities of developing and delivering** useful software. |
| What is the difference between software engineering and system engineering? | System engineering is concerned with all aspects of computer-based systems development including hardware, software and process engineering. Software engineering is part of this more general process. |

Source: Sommerville 10th Ed

# Software engineering fundamentals

- Systems should be developed using a managed and understood development process. Of course, different processes are used for different types of software.
- **Dependability** and **performance** are important for all types of system.
- **Understanding** and **managing** the software specification and requirements (what the software should do) are important.
- Where appropriate, you should **reuse** software that has already been developed rather than write new software.
- The fundamental notions of software engineering are universally applicable to all types of system development

# Attributes of good software

| Product characteristic | Description |
|---|---|
| Maintainability | Software should be written in such a way so that it can evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable requirement of a changing business environment. |
| Dependability and security | Software dependability includes a range of characteristics including reliability, security and safety. Dependable software should not cause physical or economic damage in the event of system failure. Malicious users should not be able to access or damage the system. |
| Efficiency | Software should not make wasteful use of system resources such as memory and processor cycles. Efficiency therefore includes responsiveness, processing time, memory utilisation, etc. |
| Acceptability | Software must be acceptable to the type of users for which it is designed. This means that it must be understandable, usable and compatible with other systems that they use. |

Source: Sommerville 10th Ed

# General issues that affect software

| Issues | Description |
| --- | --- |
| Security and trust | As software is intertwined with all aspects of our lives, it is essential that we can trust that software. |
| Scale | Software has to be developed across a very wide range of scales, from very small embedded systems in portable or wearable devices through to Internet-scale, cloud-based systems that serve a global community. |
| Heterogeneity | Increasingly, systems are required to operate as distributed systems across networks that include different types of computer and mobile devices. |
| Business & social change | Business and society are changing incredibly quickly as emerging economies develop and new technologies become available. They need to be able to change their existing software and to rapidly develop new software. |

Source: Sommerville 10th Ed

# Application types

| Application type | Description |
|---|---|
| Stand-alone applications | These are application systems that run on a local computer, such as a PC. They include all necessary functionality and do not need to be connected to a network. |
| Interactive transaction-based applications | Applications that execute on a remote computer and are accessed by users from their own PCs or terminals. These include web applications such as e-commerce applications. |
| Embedded control systems | These are software control systems that control and manage hardware devices. Numerically, there are probably more embedded systems than any other type of system |
| Batch processing systems | These are business systems that are designed to process data in large batches. They process large numbers of individual inputs to create corresponding outputs. |
| Entertainment systems | These are systems that are primarily for personal use and which are intended to entertain the user. |
| Systems for modelling and simulation | These are systems that are developed by scientists and engineers to model physical processes or situations, which include many, separate, interacting objects. |

Source: Sommerville 10th Ed

# Application types

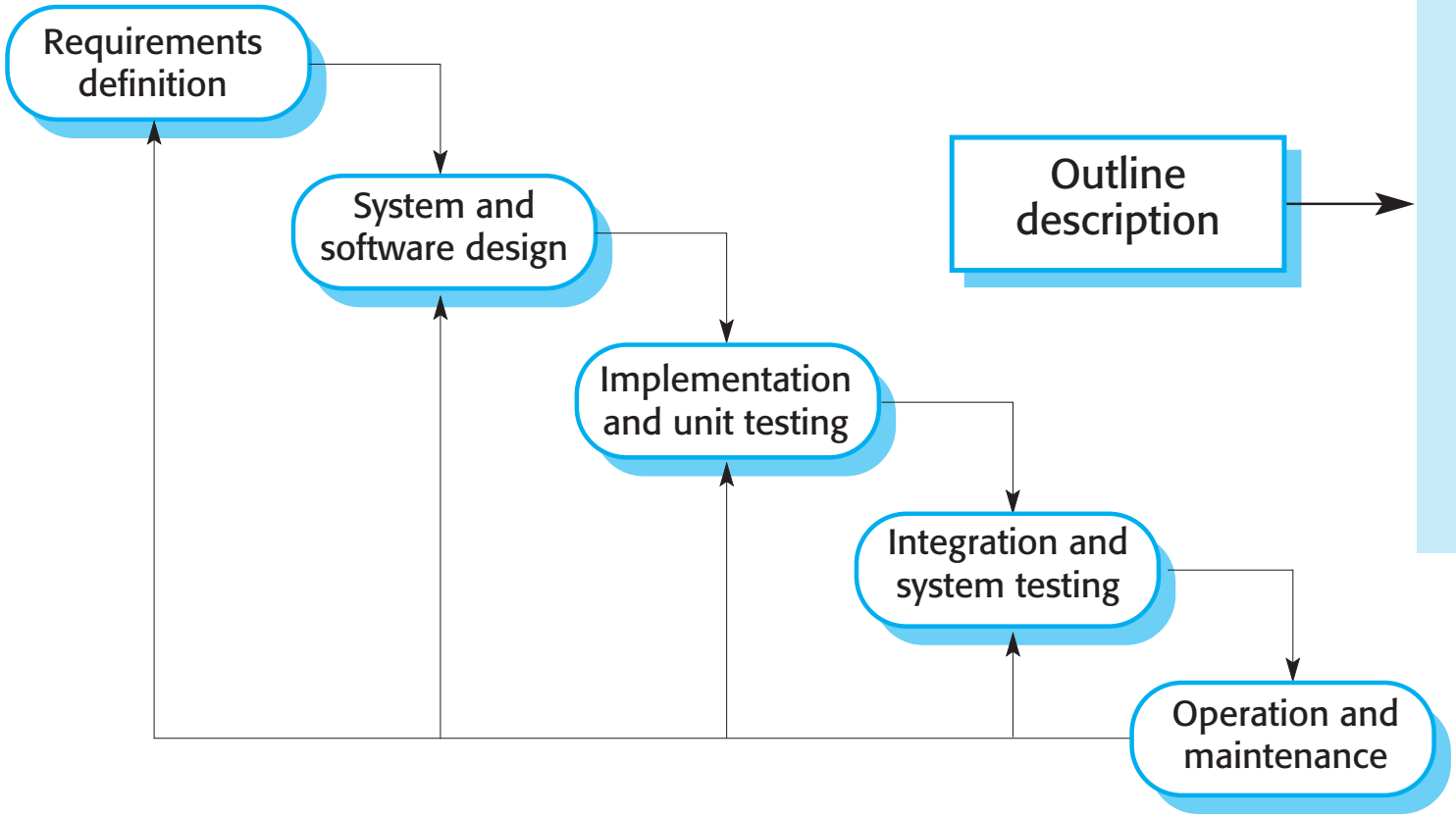
| Application type | Description |
| --- | --- |
| Data collection systems | These are systems that collect data from their environment using a set of sensors and send that data to other systems for processing. |
| System of systems | These are systems that are composed of a number of other software systems. |

Source:
Sommerville
10th Ed

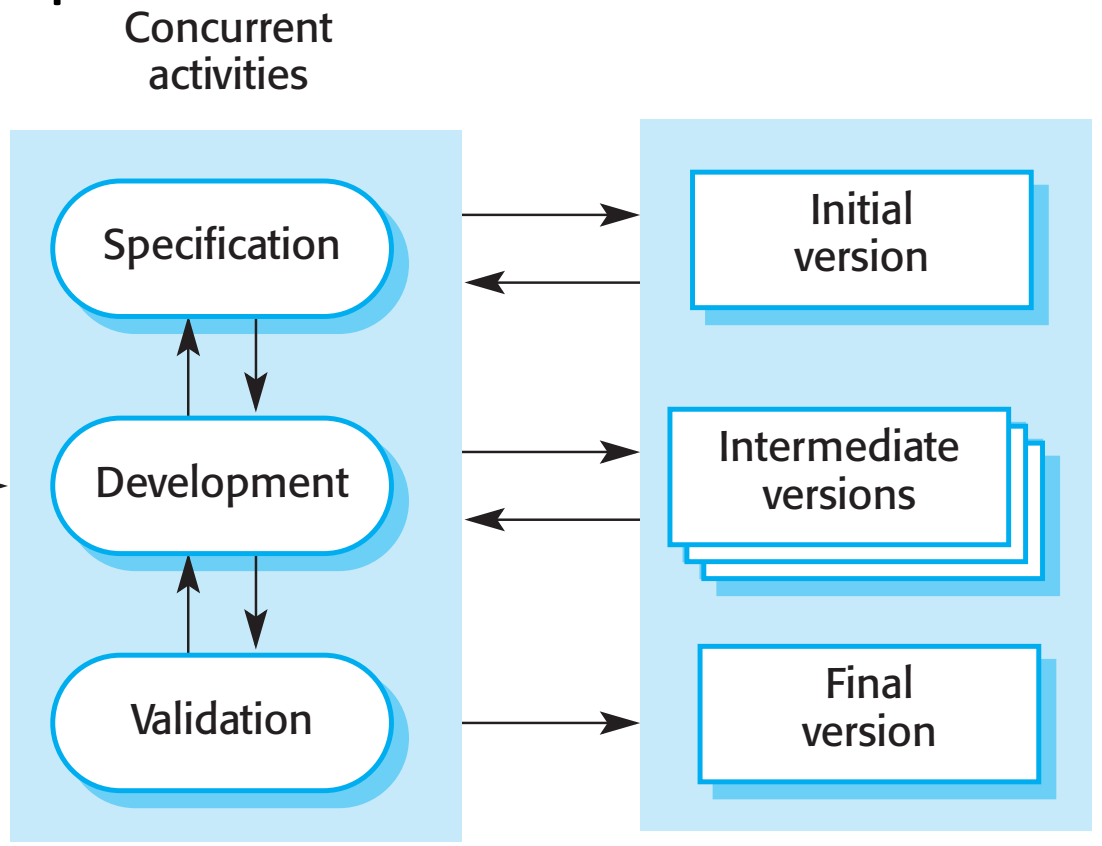
# Software process models

| Software process model | Description |
|---|---|
| Waterfall model | Takes the fundamental process activities of specification, development, validation and evolution and represents them as separate process phases. |
| Incremental development | Specification, development and validation are interleaved. May be plan-driven or agile. Accommodate changing requirements by rapidly delivering useful software as quickly as possible and integrating feedback within the development process. |
| Integration and configuration (Reuse-oriented software engineering) | The system is assembled from existing configurable components. May be plan-driven or agile.<br>This model consists of 5 key stages; requirements specification, software discovery and evaluation, refinement, application system configuration and component adaptation and integration. |

# Waterfall vs Incremental approach




Steps in the Waterfall Model




Incremental Development Model

# Waterfall vs Incremental approach

## Waterfall Model

| Benefits | Challenges |
| --- | --- |
| Easy to understand and use | No working software until end |
| Easy to manage<br>(each phase has specific measurable deliverables) | High risk<br>(of failure and mismatch of expectations) |
| Works well for project with clear and well understood requirements | Not suitable for projects where requirements are at a risk of changing |
| | |

## Incremental Model

| Benefits | Challenges |
| --- | --- |
| Working software early | Planning and design essential for success |
| Flexibility<br>(less costly to change requirements) | Poor software structure<br>(difficult to maintain cohesive structure through multiple iterations) |
| Easier to manage risk<br>(risk is identified and handled during specified iteration) | Total cost is higher than waterfall |
| | Holistic view still required<br>(clear definition needed before it can be broken down and built incrementally) |

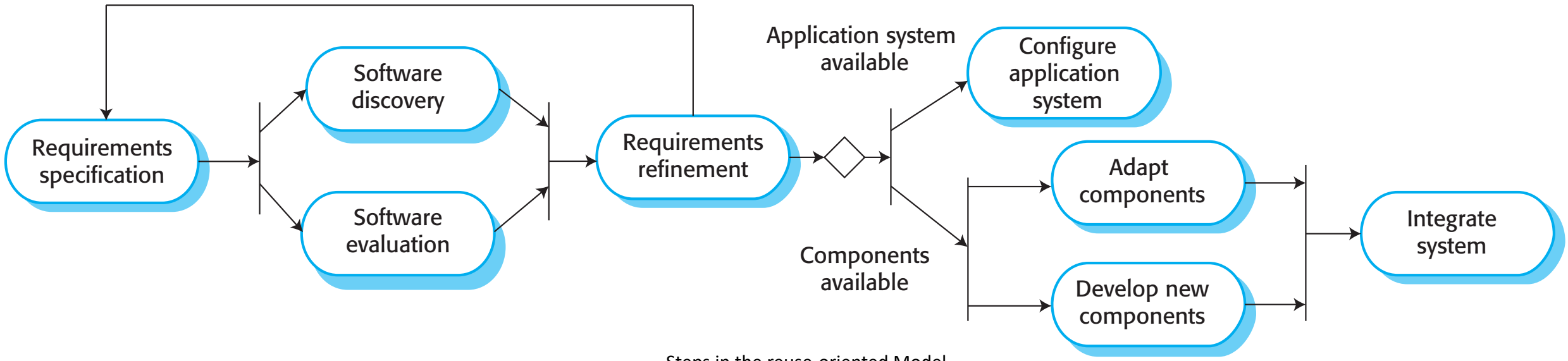
# When to Use

| Waterfall Model |
| --- |
| Used only when the requirements are very well known, clear and fixed |
| Product definition is stable |
| Ample resources with required expertise that are *freely* available |
| |
| |

| Incremental Model |
| --- |
| Uncertain and/or vague requirements<br>(for e.g.. Clients understanding of exact purpose unclear) |
| There is a need to get a product to the market early |
| A new technology is being used<br>(for e.g. the developers are using a new language/framework) |
| There are high risk features and goals |
| Resources with needed skill sets are not available |

NB: In general, there are very few examples where exact requirement are known or fixed. Therefore, this limits the applicability of the waterfall model within today's development cycle.

# Reuse-oriented software engineering

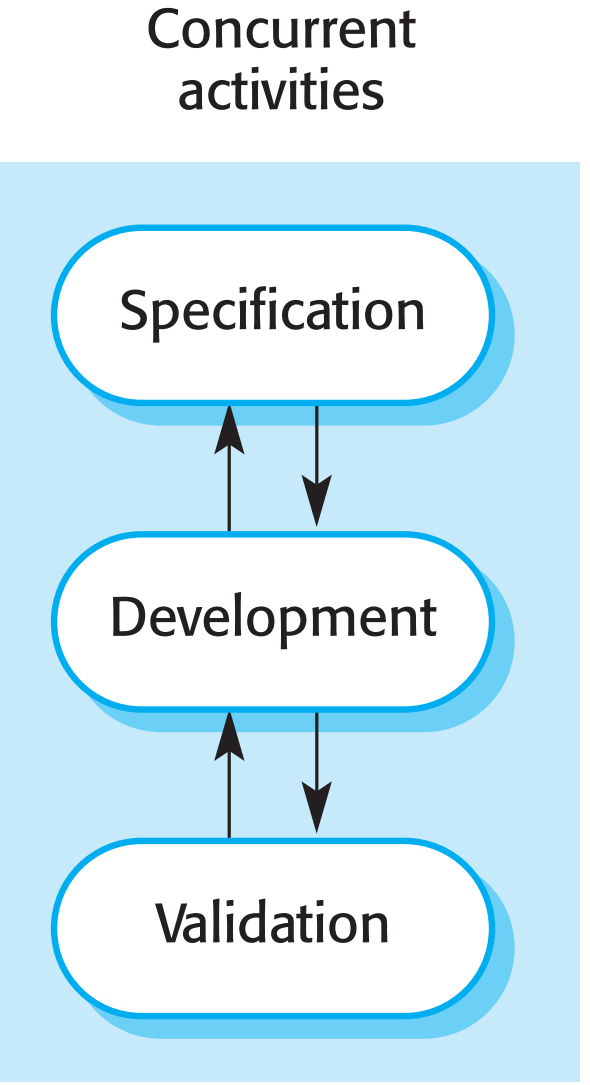



Steps in the reuse-oriented Model

# Process activities

# Software process activities

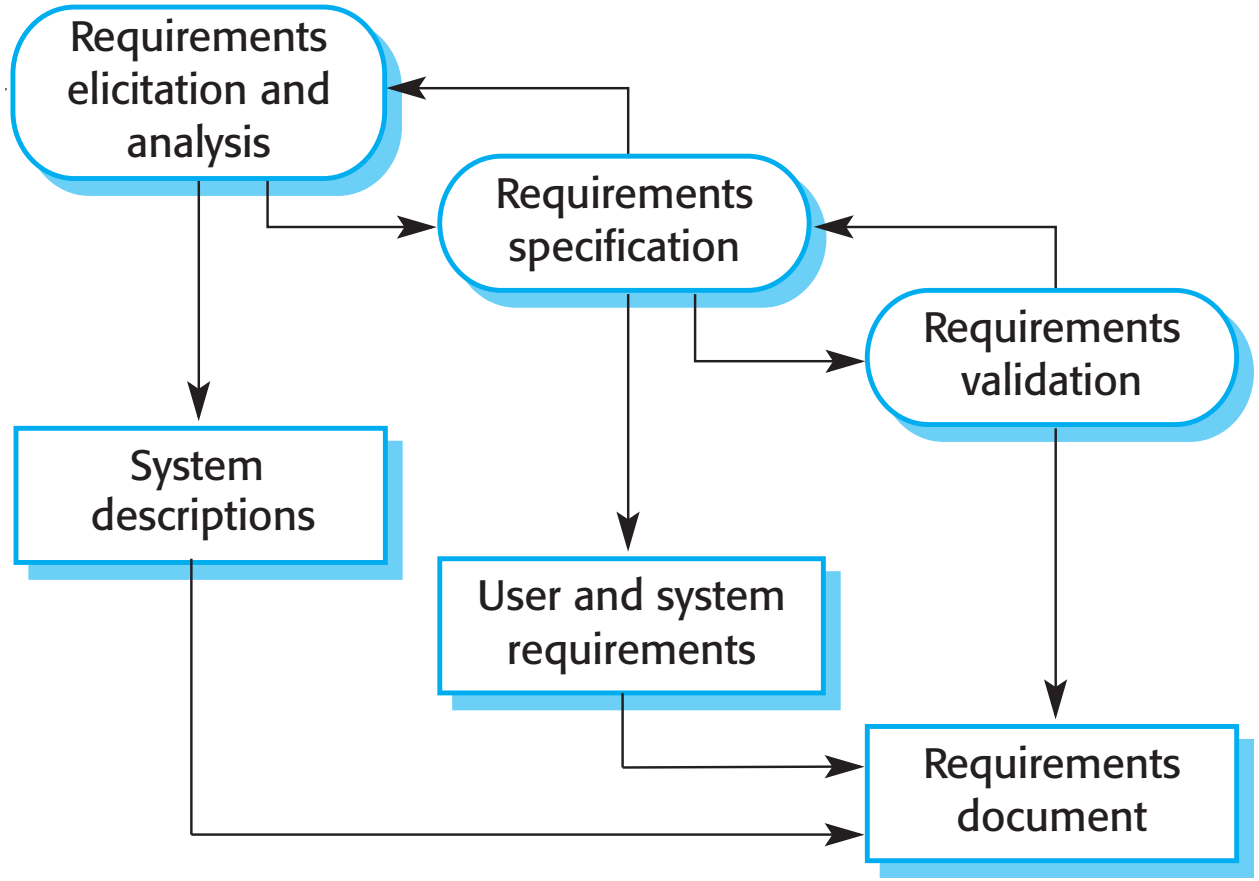
- **Software specification**: where customers and engineers define the software that is to be produced and the constraints on its operation.
- **Software development**: where the software is designed and programmed.
- **Software validation**: where the software is checked to ensure that it is what the customer requires.
- **Software evolution**: where the software is modified to reflect changing customer and market requirements.




NB: These activities are organized (and emphasized) differently in different development processes. (For e.g. the waterfall organize the activities in a sequence while they are interleaved in the incremental model)

# Software specification

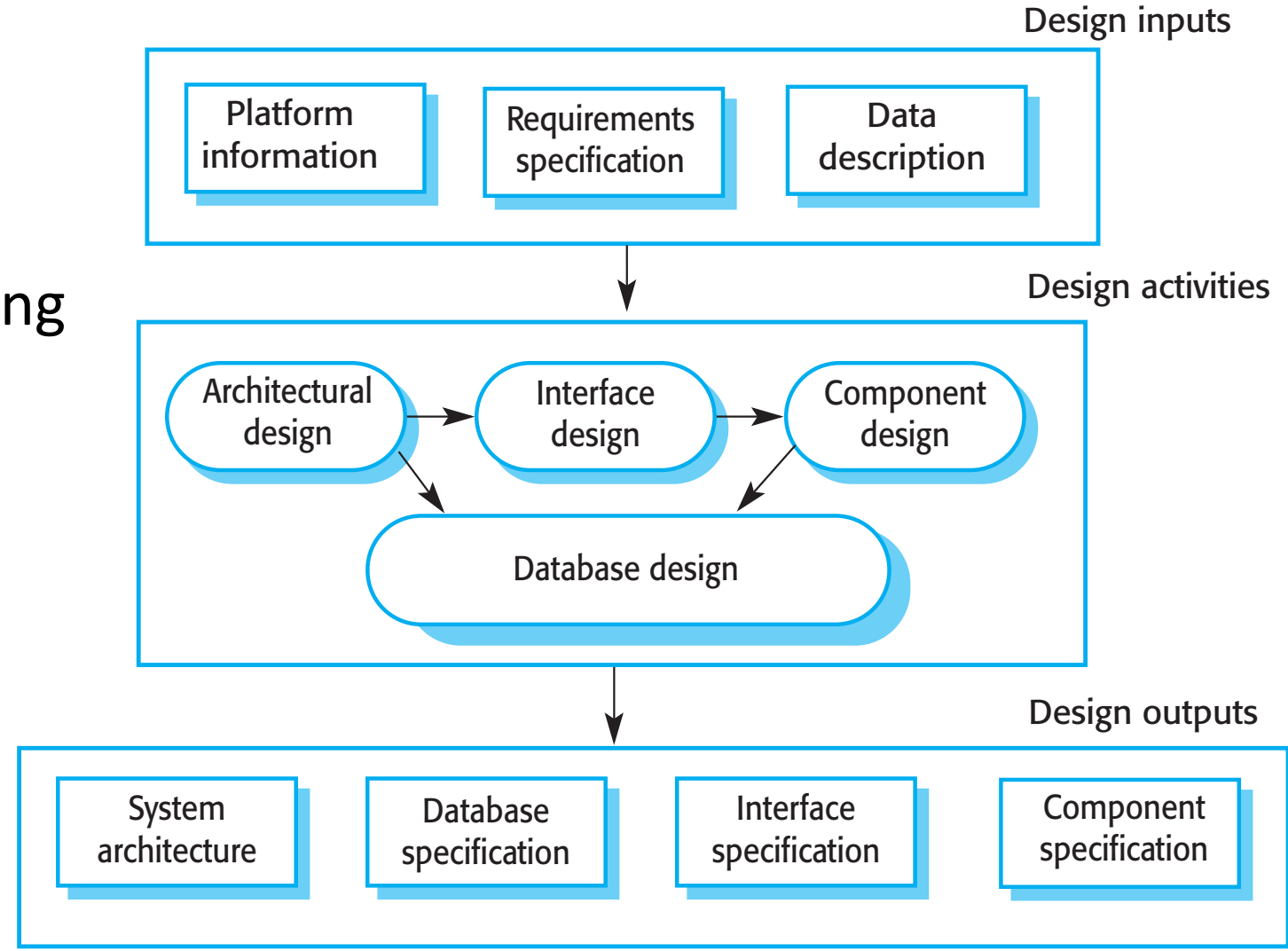
## Requirement Engineering Process




| Requirement Component | Description |
|---|---|
| Elicitation and analysis | What do the system stakeholders require or expect from the system? |
| Specification | Defining the requirements in detail |
| Validation | Checking the validity of the requirements. This includes both functional and end-user evaluations. |

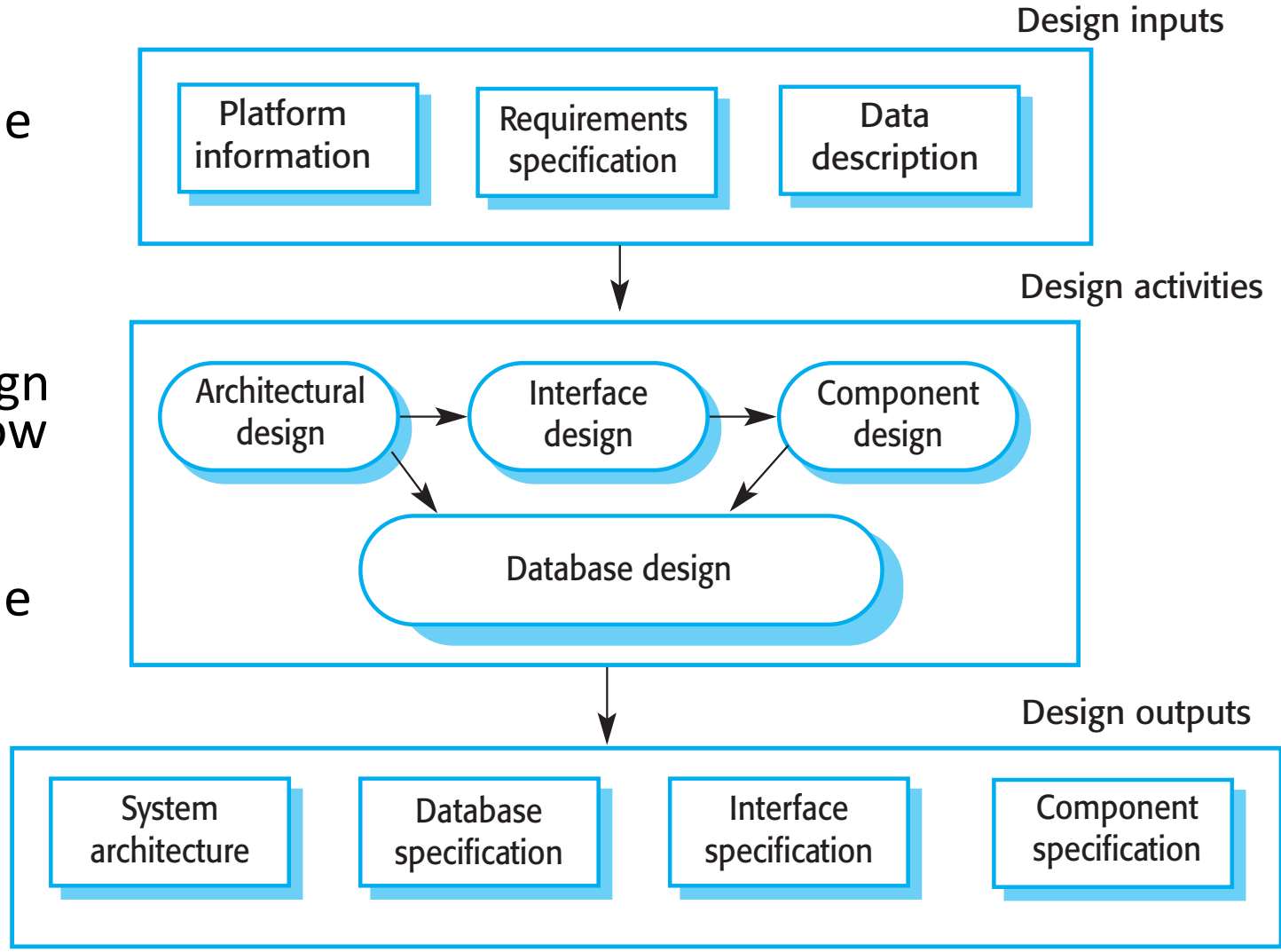
# Design

- Design is the process of converting the requirements into a structure that can be subsequently used for realizing the specification (Implementation).



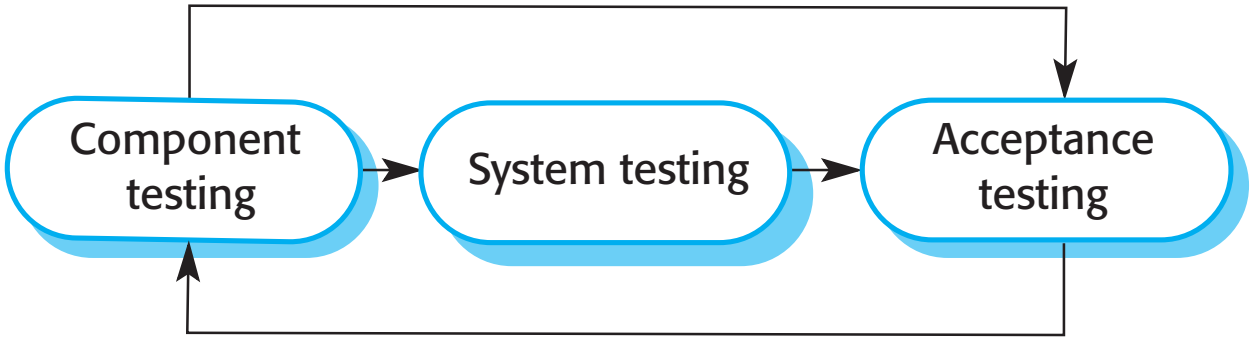
# Design

- ***Architectural design,*** where you identify the overall structure of the system, the principal components (subsystems or modules), their relationships and how they are distributed.
- ***Database design,*** where you design the system data structures and how these are to be represented in a database.
- ***Interface design,*** where you define the interfaces between system components.
- ***Component selection and design,*** where you search for reusable components. If unavailable, you design how it will operate



# Testing

- ***Component testing***
  - Individual components are tested independently;
  - Components may be functions or objects or coherent groupings of these entities.
- **Integration Testing**
  - Finds errors with connecting subsystems
- ***System testing***
  - Testing of the system as a whole. Testing of emergent properties is particularly important.
- ***Acceptance testing***
  - Testing with customer data to check that the system meets the customer's needs.



# Testing

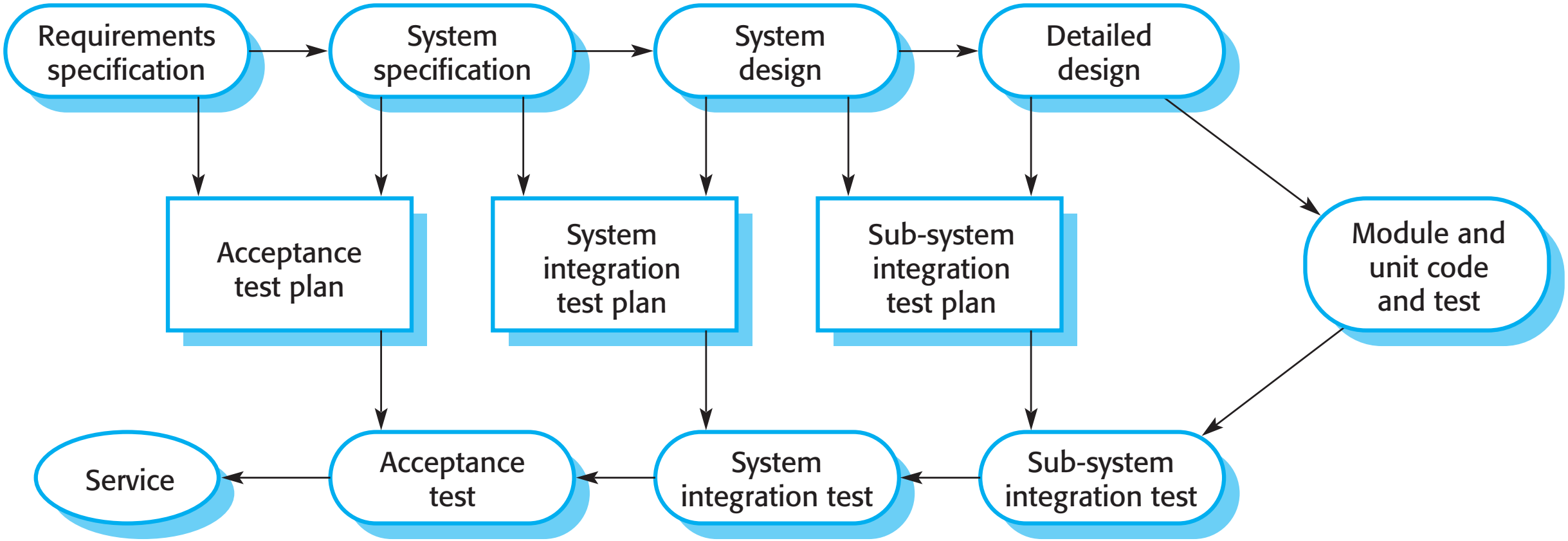
## *Component based testing*

- ***Black-box test***: focuses on the input/output of each component
  - ***Boundary test***: testing conditions on bounds between equivalent classes
- ***White-box test***: focuses on the internal states of objects
  - ***Path test***: an attempt to use test input that will pass once over each path in the code
- ***Regression test***: re-executing all prior test after a code change

## *System Testing*

- ***Functional test:*** black-box test through each use case
- ***Performance test:*** verify non functional performance requirements
- ***Pilot test:*** Install and use system by a set of users (alpha, beta, usability)

# Testing




Testing phases in a V-model testing process

# Testing – Quality Control Techniques

- **Fault avoidance**: prevent errors by finding faults before the system is released
  - Development methodology: TDD
  - Configuration management: Consistent subsystem interfaces
  - Review: manual inspection of system
- **Fault detection:** find existing faults without recovering from the errors
  - Testing: exposes errors in a planned/strategic way
- **Fault tolerance:** when system can recover from failure itself
  - Modular redundancy: assigns more than one component for the same task

# Coping with Change

- Change is inevitable in all large software projects
  - (business changes, new technologies, platform changes)

| Changing Requirements | |
|---|---|
| ***System prototyping*** – system developed quickly to check against customer requirements and expectations | |
| | |

# Coping with Change

- Change is inevitable in all large software projects
  - (business changes, new technologies, platform changes)

| Changing Requirements | |
|---|---|
| ***System prototyping*** – system developed quickly to check against customer requirements and expectations | |
| ***Incremental delivery*** – System increments are delivered to customer for comment and experimentation | |

# Coping with Change

- Change is inevitable in all large software projects
  - (business changes, new technologies, platform changes)

| Changing Requirements | Managing Costs |
| --- | --- |
| ***System prototyping*** – system developed quickly to check against customer requirements and expectations | ***Change Anticipation*** – include activities that anticipate change before significant rework is required |
| ***Incremental delivery*** – System increments are delivered to customer for comment and experimentation | |

# Coping with Change

- Change is inevitable in all large software projects
  - (business changes, new technologies, platform changes)

| Changing Requirements | Managing Costs |
| --- | --- |
| ***System prototyping*** – system developed quickly to check against customer requirements and expectations | ***Change Anticipation*** – include activities that anticipate change before significant rework is required |
| ***Incremental delivery*** – System increments are delivered to customer for comment and experimentation | ***Change Tolerance*** – process designed so that changes can be accommodated at relatively low cost |

# Agile Development

# Principles of Agile Development

- There are 12 principles of the agile manifesto:
  1. Highest priority is to satisfy the customer
     - (through early and continuous delivery of valuable software)
  2. Welcome changing requirements
     - (even late in the development)
  3. Deliver working software frequently
     - (from a couple of weeks to a couple of months with a preference to the shorter timescale)
  4. Business people and developers must work together
     - (daily and throughout the project)
  5. Build projects around motivated individuals.
     - (Give them the environment and support they need, and trust them to get the job done)

# Principles of Agile Development

- There are 12 principles of the agile manifesto (cont'd):
  6. Face to face conversation
     - (The most efficient and effective method of conveying information)
  7. Working software is the primary measure of progress
  8. Processes promote sustainable development
     - (The sponsors, developers and users should be able to maintain a constant pace indefinitely)
  9. Continuous attention to technical excellence and good design enhances agility
  10. Simplicity is essentially
      - (the art of maximizing the amount of work not done)

# Principles of Agile Development

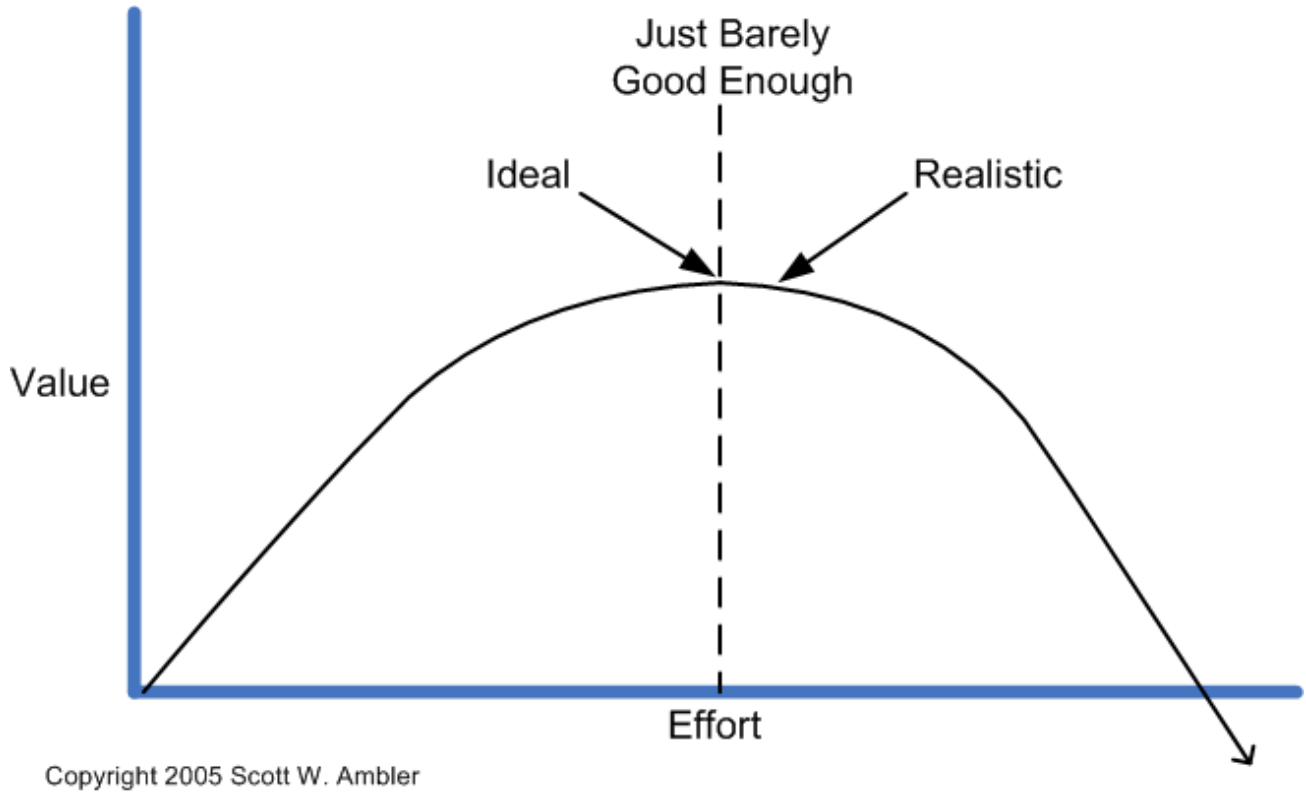
- There are 12 principles of the agile manifesto (cont'd):
  11. The best architectures, requirements and designs emerge from self-organizing teams
  12. Team reflection
      - (Reflect at regular intervals on how to become more effective, the tune and adjust the team behaviour accordingly)

# Agile Model Driven Development

- A Model Driven Development (MDD) is an approach to software development where extensive models are created before source code is written.
- While some agile strategies emphasize the delivery of products over documentation some agile models such as the Rational Unified Process (RUP) require MDD explicitly as part of the process
- The Agile Model Driven Development (AMDD) differentiates itself from the typical MDD by creating models that are just barely good enough (JDGE) to drive your overall development efforts.

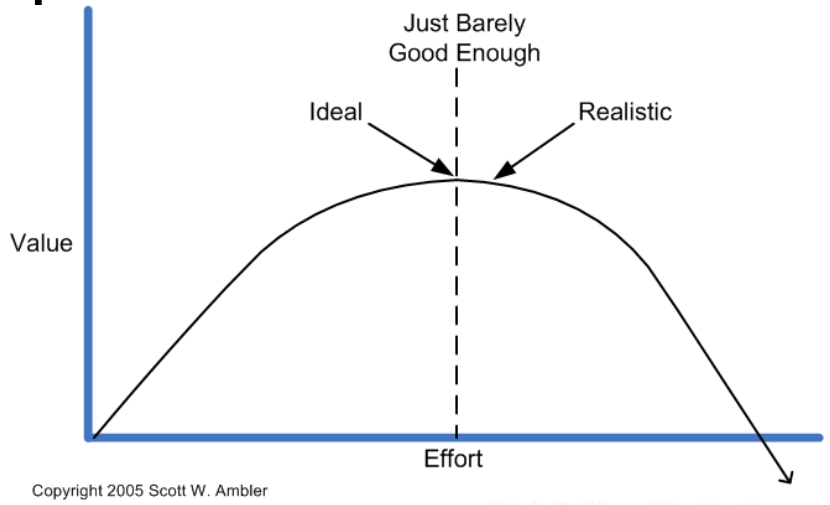
# Agile Model Driven Development - JBGE

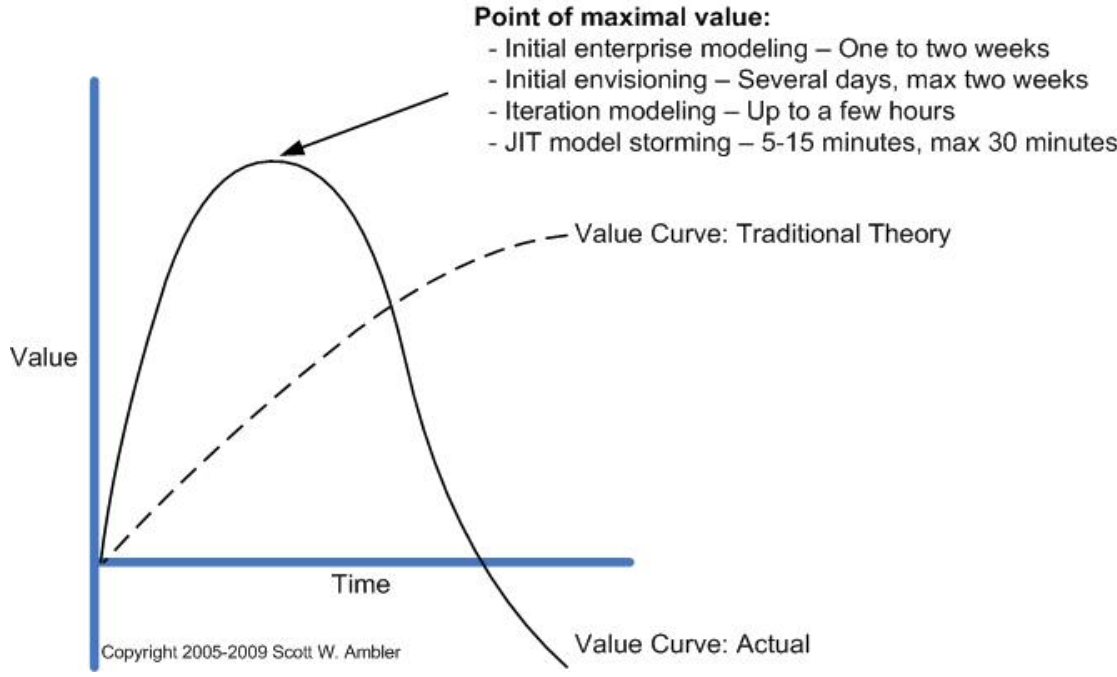
- Just Barely Good Enough:
  - **Is actually very effective**: *Once an artefact fulfils its intended purpose then any more investment in it is simply busy work.*
  - **Is Situational**: *The use of the diagram and complexity of the diagram must reflect the needs of the situation*
  - **Does not imply low quality**: *The audience of the artefact determines if its sufficient not the creators of it*



# Agile Model Driven Development - JBGE

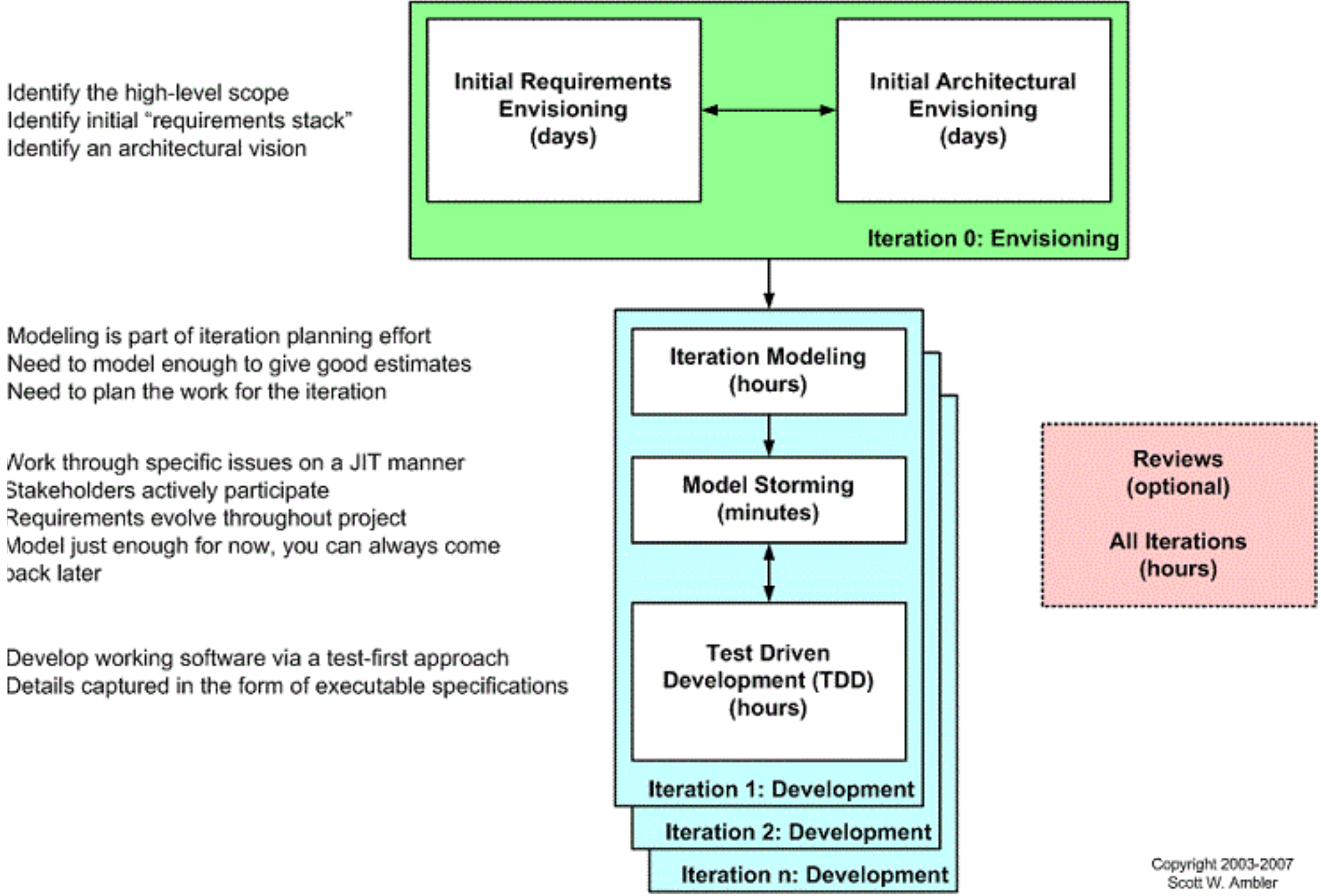
- Just Barely Good Enough:
  - **Changes over time:** *An artefact can move along this value curve in both directions along its lifecycle and must be updated accordingly*
  - **Comes sooner than you think:** *With improved communication and throughout thinking allows the value of the modelling to be realized quickly.*



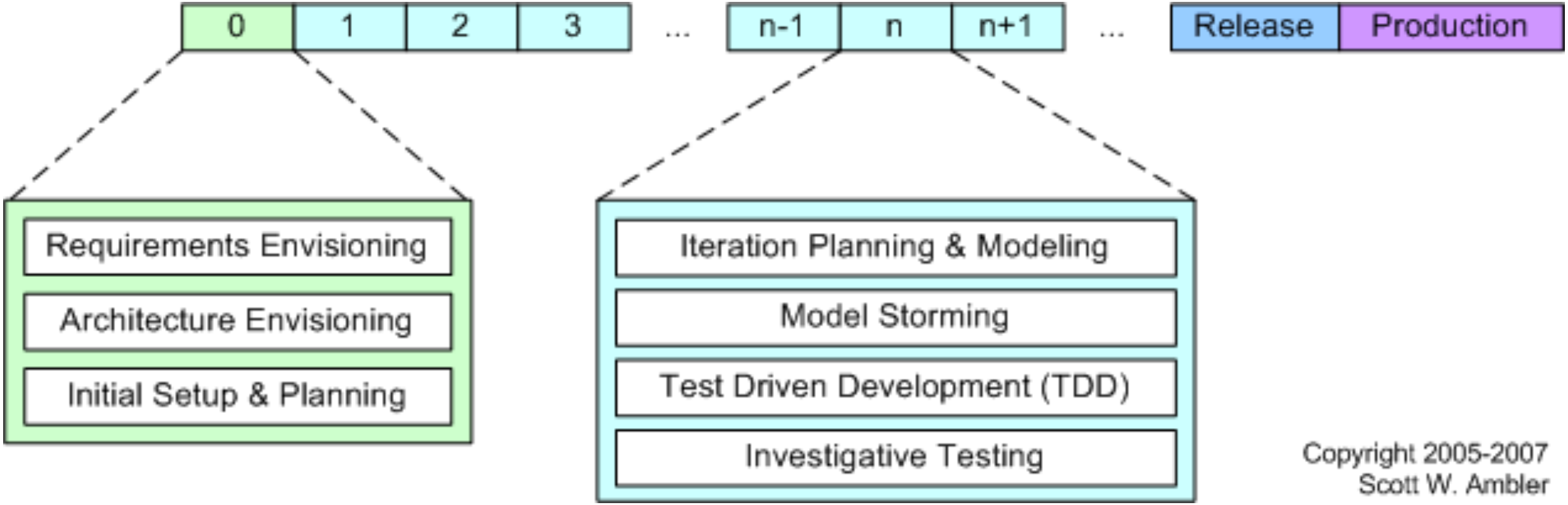
# Agile Model Driven Development

- AMDD Enables:
  - Meeting Project Planning Needs:
    - Identifying high-level requirements early, and by identifying a potential architecture early, you have enough information to produce an initial cost estimate and schedule
  - Management of technical risk:
    - Initial modelling efforts enable you to identify the major areas of technical risk early in the project without taking on the risk of later refactoring of the system.
  - Minimization of waste:
    - Ensures that you model the system that is needed immediately
  - Better questions:
    - Modelling can increase understanding and allow developers to ask more intelligent questions
  - Clarity for stakeholders:
    - Models can help stakeholders to have a better and tangible reference to understand the system




AMDD Lifecycle: Modelling activities throughout the lifecycle of a project

# Agile Model Driven Development




AMDD Through the Agile Development Lifecycle

# Agile Model Driven Development

Where is the modelling applied in each approach?

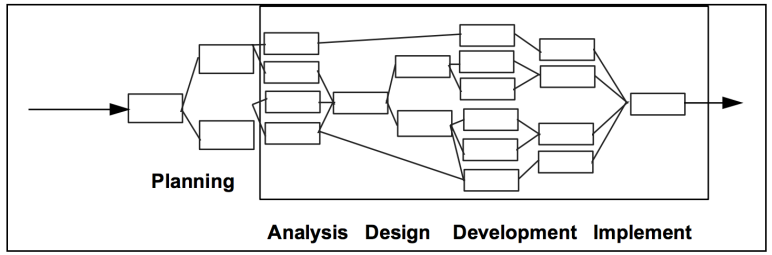



Figure 1 : Waterfall Methodology

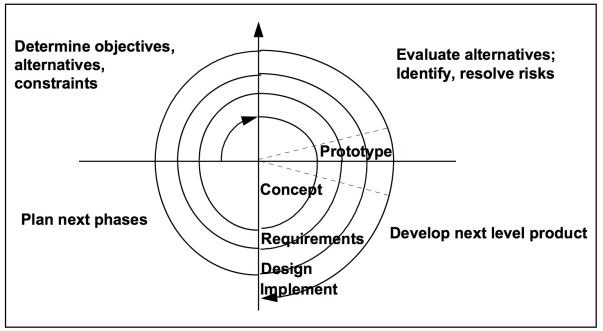



Figure 2 : Spiral Methodology

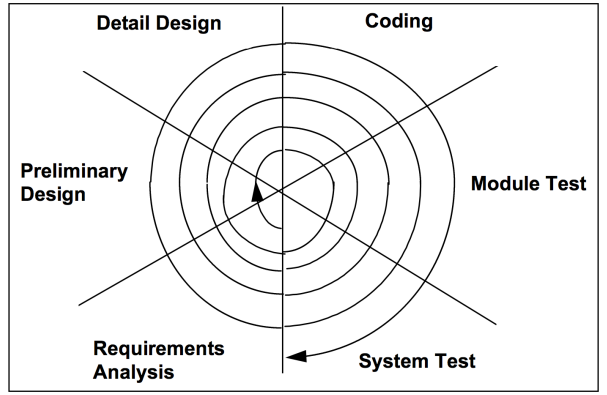



Figure 3 : Iterative Methodology

# Agile Model Driven Development

Where is the modelling applied in each approach?

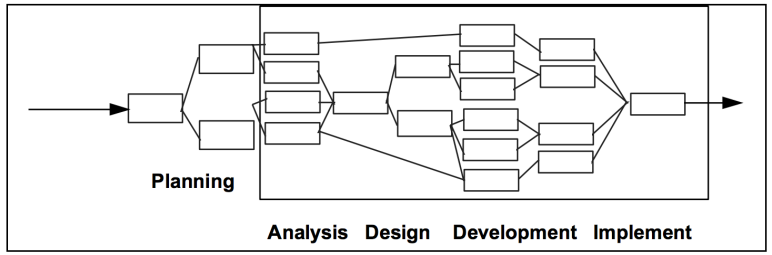



Figure 1 : Waterfall Methodology

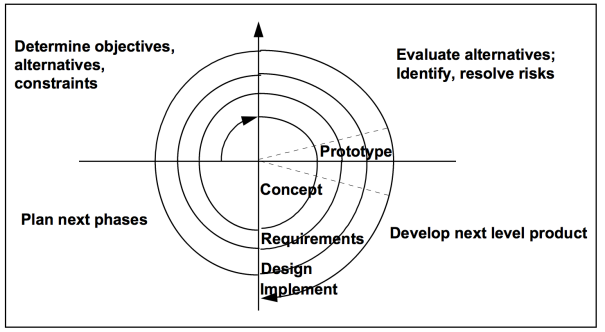



Figure 2 : Spiral Methodology

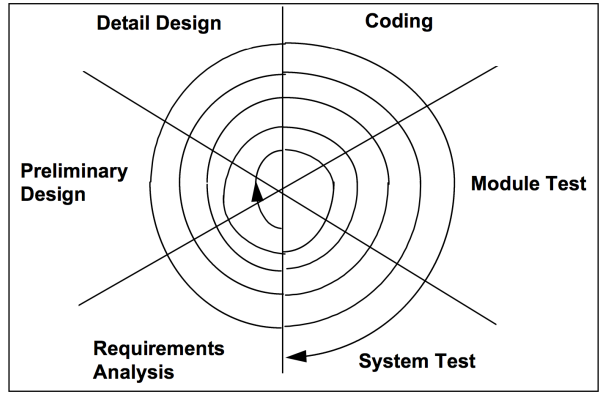



Figure 3 : Iterative Methodology

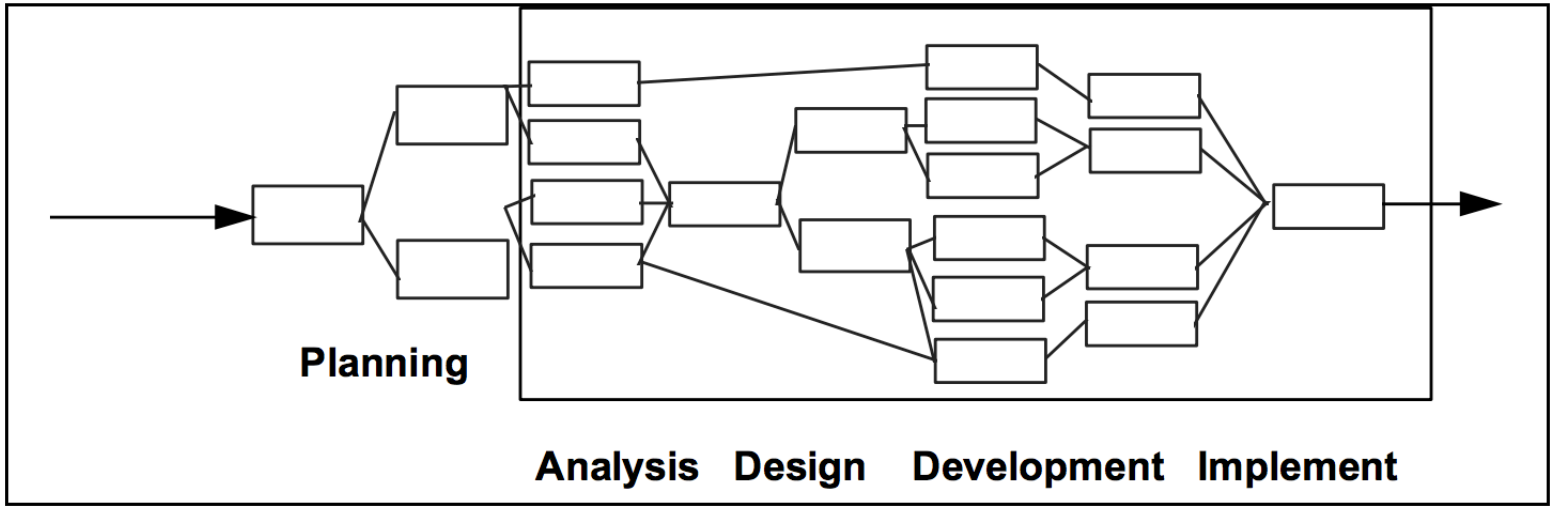



**Figure 1 : Waterfall Methodology**

# Agile Model Driven Development

Where is the modelling applied in each approach?

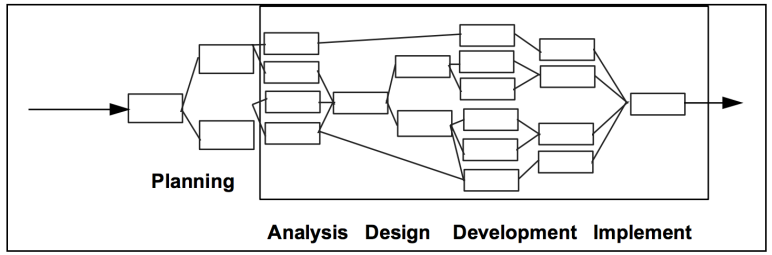



Figure 1 : Waterfall Methodology

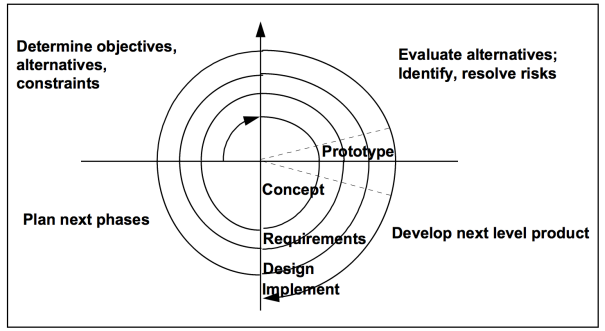



Figure 2 : Spiral Methodology

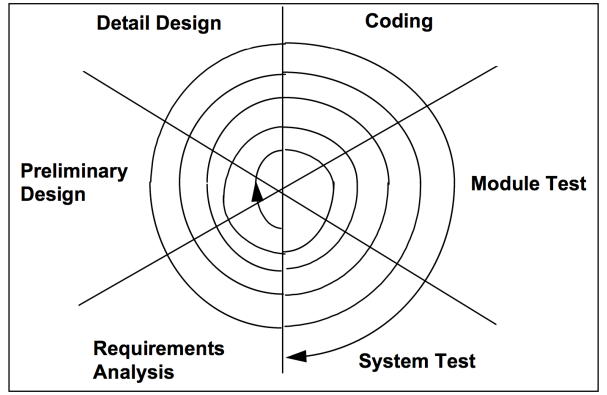



Figure 3 : Iterative Methodology

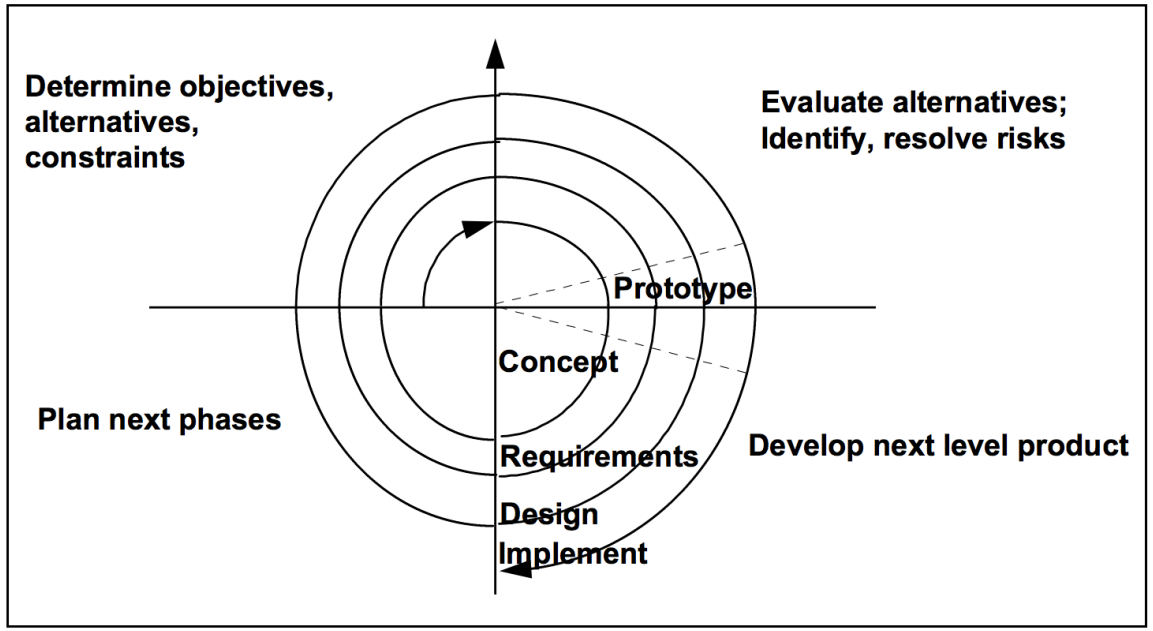



**Figure 2 : Spiral Methodology**

The Spiral methodology “peels the onion”, progressing through “layers” of the development process. A prototype lets users determine if the project is on track, should be sent back to prior phases, or should be ended. However, the phases and phase processes are still linear. Requirements work is still performed in the requirements phase, design work in the design phase, and so forth, with each of the phases consisting of linear, explicitly defined processes.

# Agile Model Driven Development

Where is the modelling applied in each approach?

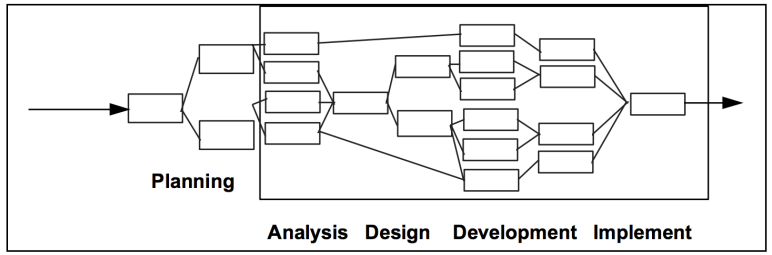



Figure 1 : Waterfall Methodology

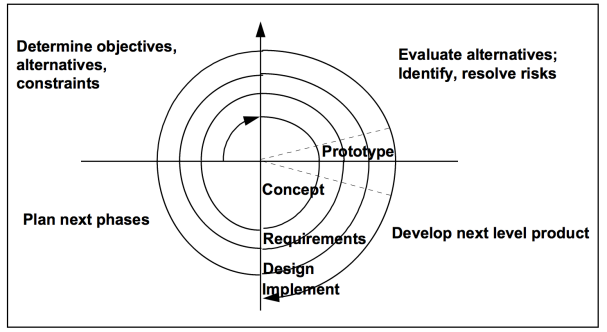



Figure 2 : Spiral Methodology

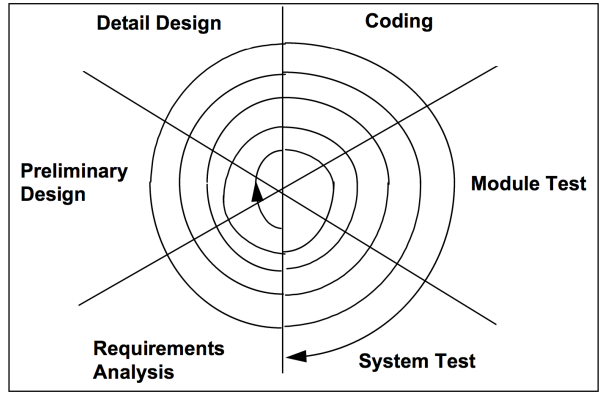



Figure 3 : Iterative Methodology

The Iterative methodology improves on the Spiral methodology. Each iteration consists of all of the standard Waterfall phases, but each iteration only addresses one set of parsed functionality. The overall project deliverable has been partitioned into prioritized subsystems, each with clean interfaces. Further iterations can add resources to the project while ramping up the speed of delivery.

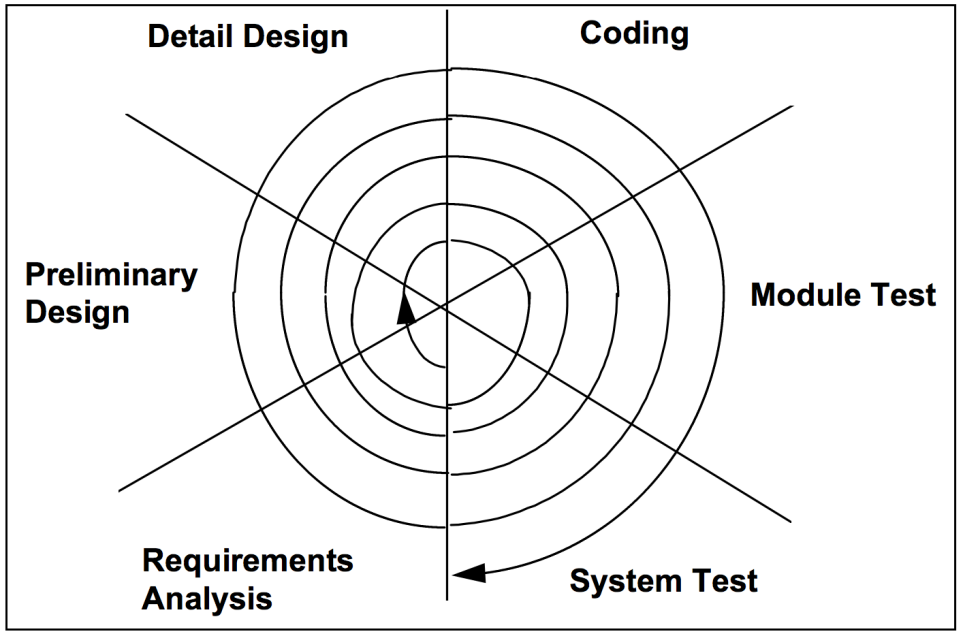



Figure 3 : Iterative Methodology

# Incremental (Agile) Methodologies

# Incremental (Agile) Methodologies

- In the last 5 to 10 years, many new and different agile methodologies cam into practice. While the names, activities and acronyms differ, they were all aimed at the same problem: create reliable software more quickly and better aligned to the users' requirements.

# Incremental (Agile) Methodologies

- These methods include:
  1. *Dynamic System Development Method* (Dane Faulkner and others)
  2. Adaptive Software Development (Jim Highsmith)
  3. Crystal Clear (a family of methods, Alistair Cockburn)
  4. **Scrum** (Ken Schwaber, Jeff Sutherland, Mark Beedle)
  5. **XP** (Kent Beck, Eric Gamma, and others)
  6. Lean Software Development (Mary and Tom Poppendieck)
  7. *Feature-Driven Development* (Peter Coad and Jeff DeLuca)
  8. Agile Unified Process (Scott Ambler)
  9. **Kanban** (David Anderson, Corey Ladas)

# Fun (useful) Reading

- Late projects, man-months and the software crisis
  - http://www.stellman-greene.com/2007/05/15/late-projects-man-months-and-the-software-crisis/
- The Humble Programmer - Edsger W. Dijkstra
  - http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD340.PDF
- The software engineering code of ethics and professional practice
  - http://www.acm.org/about/se-code

# References


- http://agilemodeling.com/
- http://agilemanifesto.org/
- Scaling software agility - http://www.infoq.com/