CSC301: Introduction to Software Engineering

Lecture 4

Wael Aboulsaadat

Introduction to Software Development Lifecycle SDLC

The Software Crisis: why?

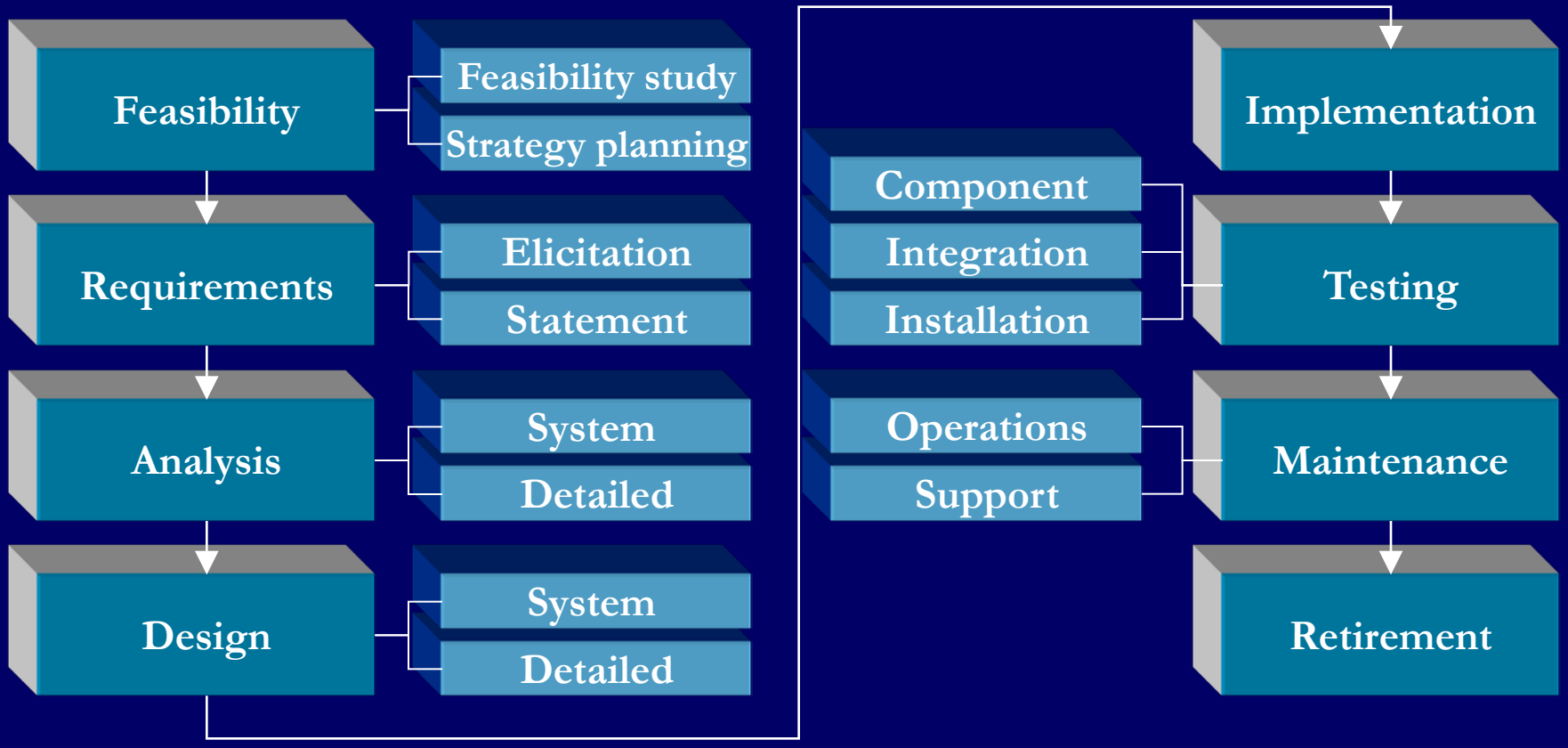
- Monolithic development is not effective for modern system development.
  - No process control
  - No product or process guarantees
  - No true management
  - No client confidence
  - No process visibility / traceability
  - No metrication
  - No communication

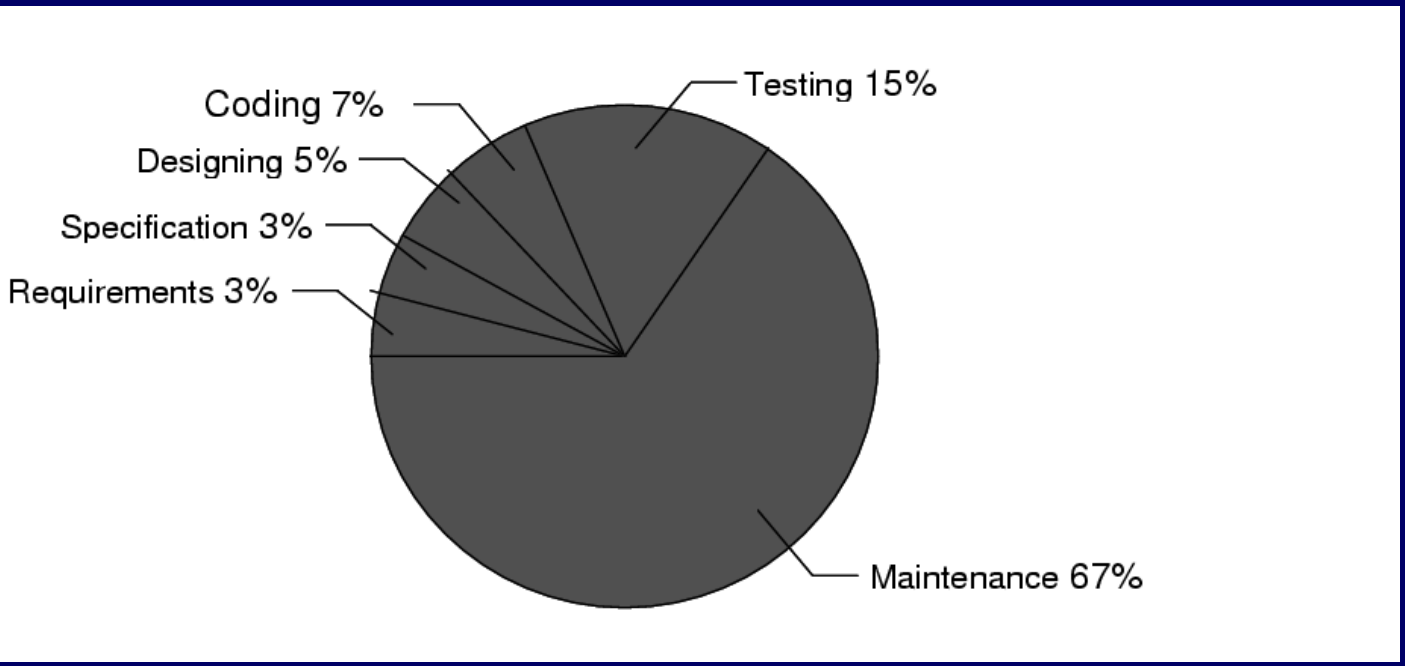
  => no quality!

Breaking the Monolithic Model

- Done by introducing “steps” into the software development process.
- Steps in the development process are called “phases” (or “stages”).
- Phases must be self contained and pre-defined.
- Phases should decrease abstraction as they progress.

Typical Phases in Software Development




Coding 7%
Testing 15%
Designing 5%
Specification 3%
Requirements 3%
Maintenance 67%

A Software Development Phase

A software development phase:

- is a delimited period of time within the process of development of a software system.
- has a definite starting set of data and a definite set of results.
- is based on the results set of earlier phases.

Some Advantages of Phased Development

- Phased development
  - Offers benchmarking
  - Offers insight
  - Offers mile-stoning niches
  - Offers a documentation-building framework
  - Offers a definite progression sequence
  - Offers possibilities for prototyping
  - Allows end-user and client participation
  - Offers possibilities for better testing strategies

A Development Milestone

- A software development milestone is a scheduled event…
  - for which some project member or manager is accountable.
  - is used to measure progress.
- A milestone typically includes:
  - a formal review.
  - the issuance of documents.
  - the delivery of a (sometimes intermediate) product.

Development Models

- **Development model definition:**
  - A particular interaction configuration of development phases leading to a final software product.

Life Cycle

- A life-cycle…
  - is a finite and definite period of time.
  - starts when a software product is conceived.
  - ends when the product is no longer available or effective for use.
- Any life-cycle is organised in (composed of) phases

The Development Life-Cycle

(aka The Software Development Process)

- A project is a set of activities, interactions and results.
- A life-cycle or a software process is the organisational framework for a project.

The Nature of an Effective Development Model (DM)

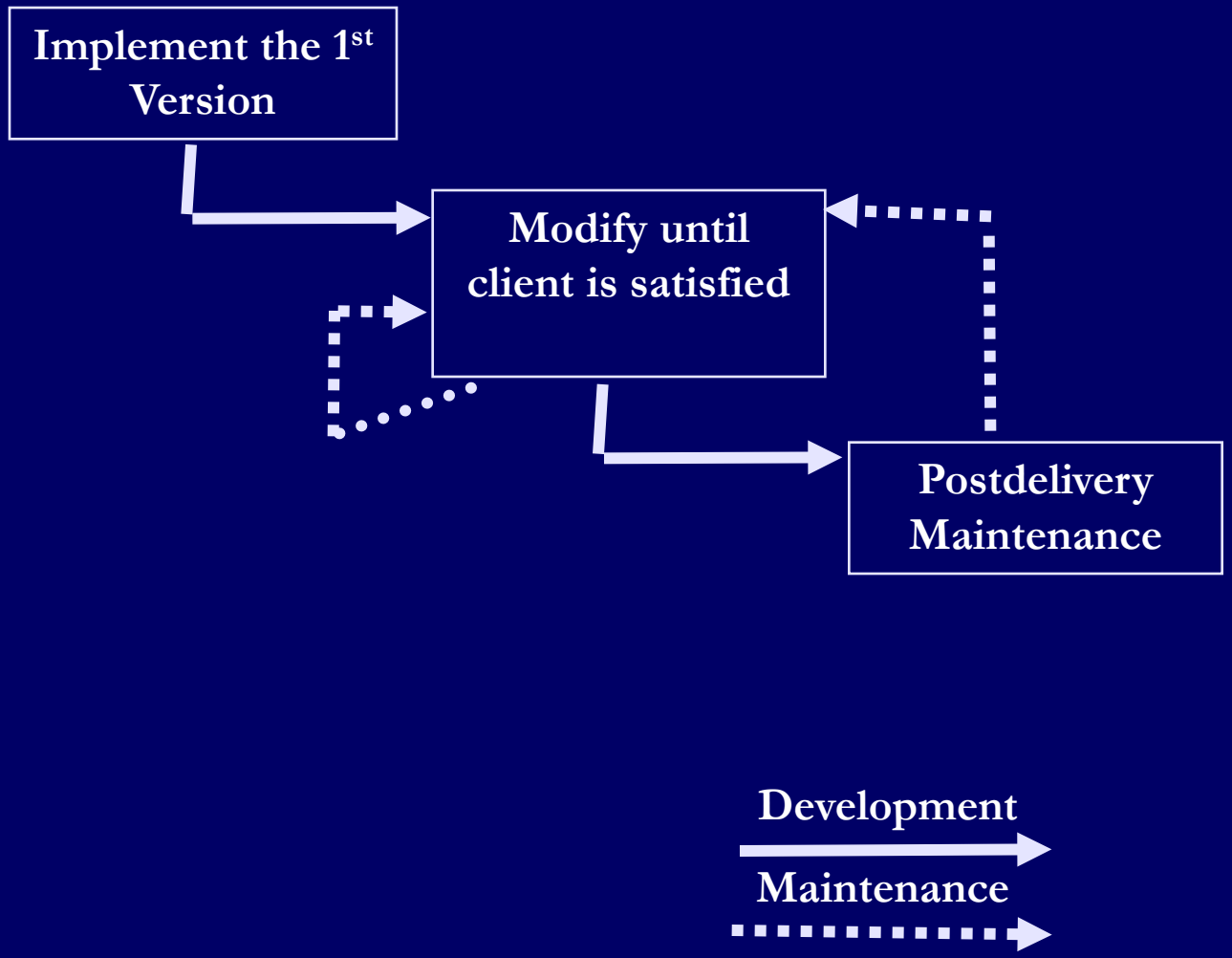
An effective DM is one that:

- Effectively links the phases it includes
- Focuses phases towards a definite goal
- Provides mechanisms for the controlled decrease of system abstraction
- Includes definite milestones
- Is transparent
- Is traceable between adjacent phases

Code-and-Fix model!

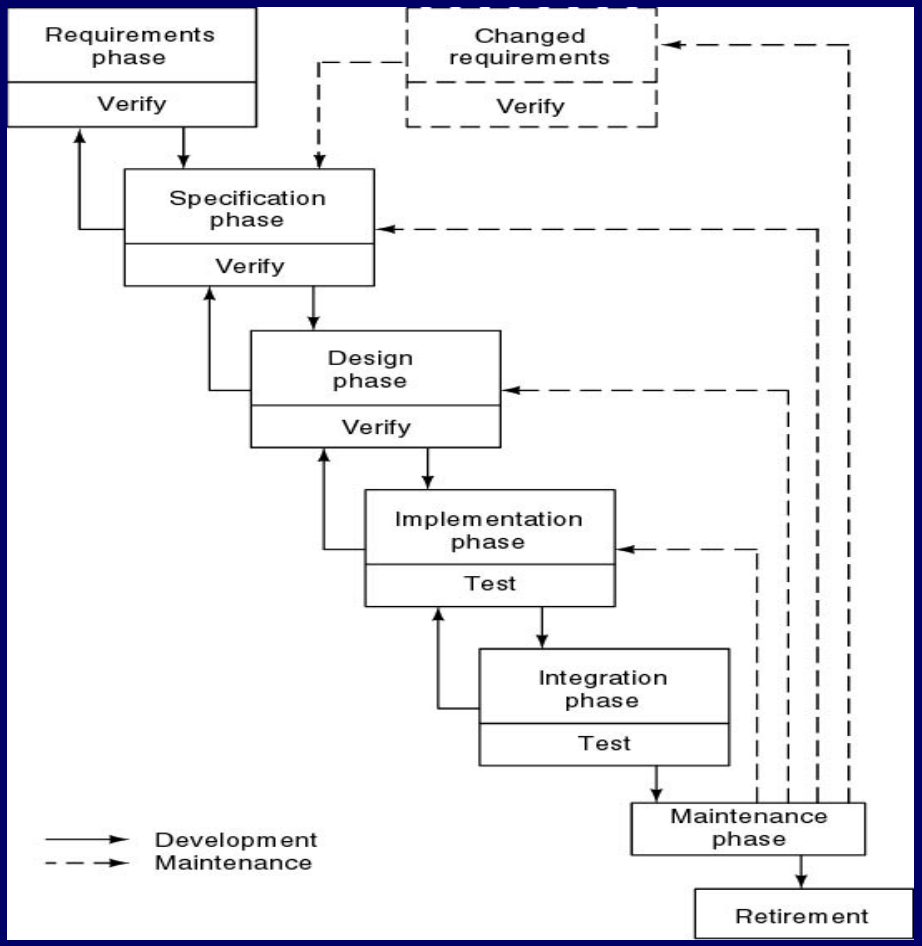
Code-and-Fix Model

- No design
- No specifications
  - Maintenance nightmare
- The easiest way to develop software
- The most expensive way
- Typically used by a start-up…

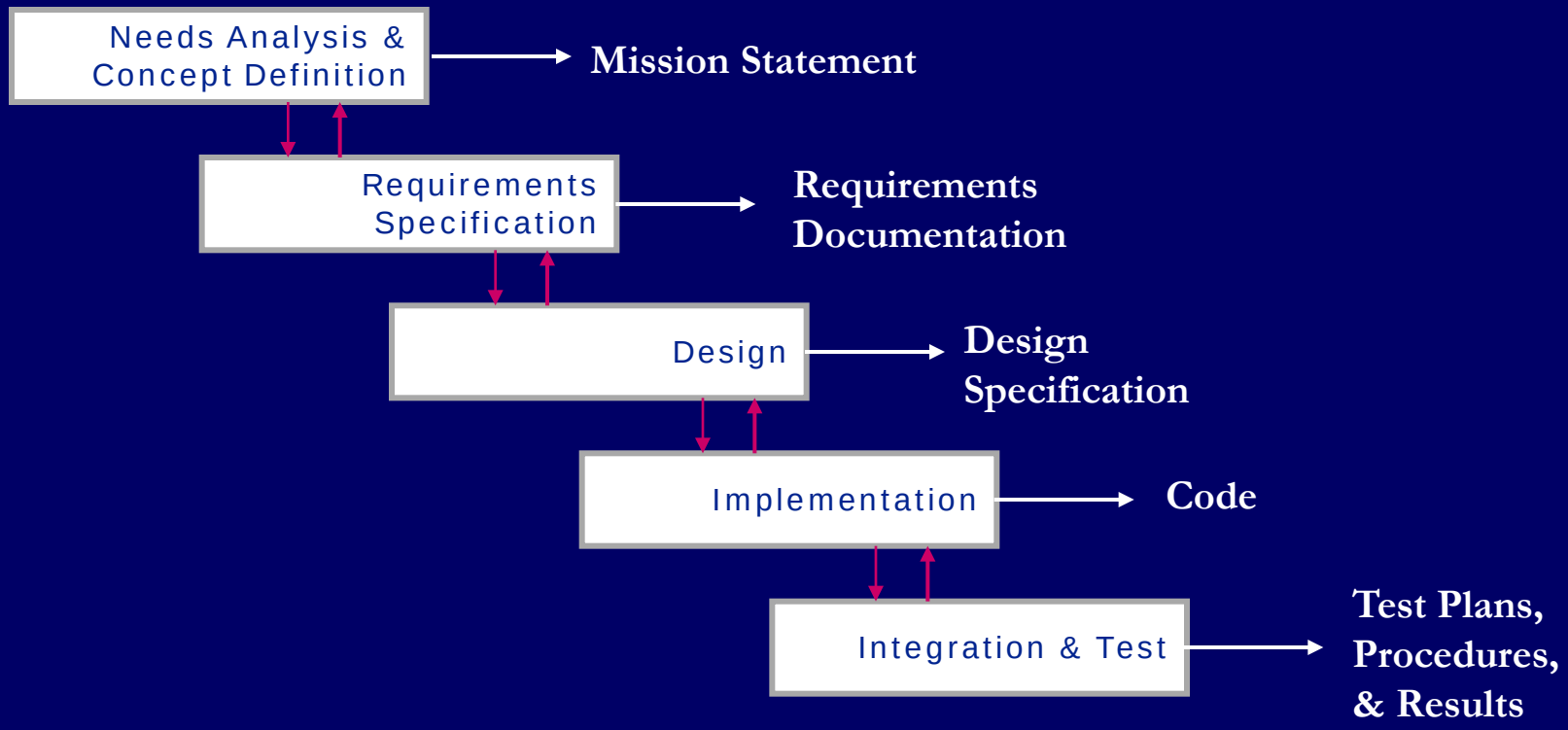


Waterfall Model

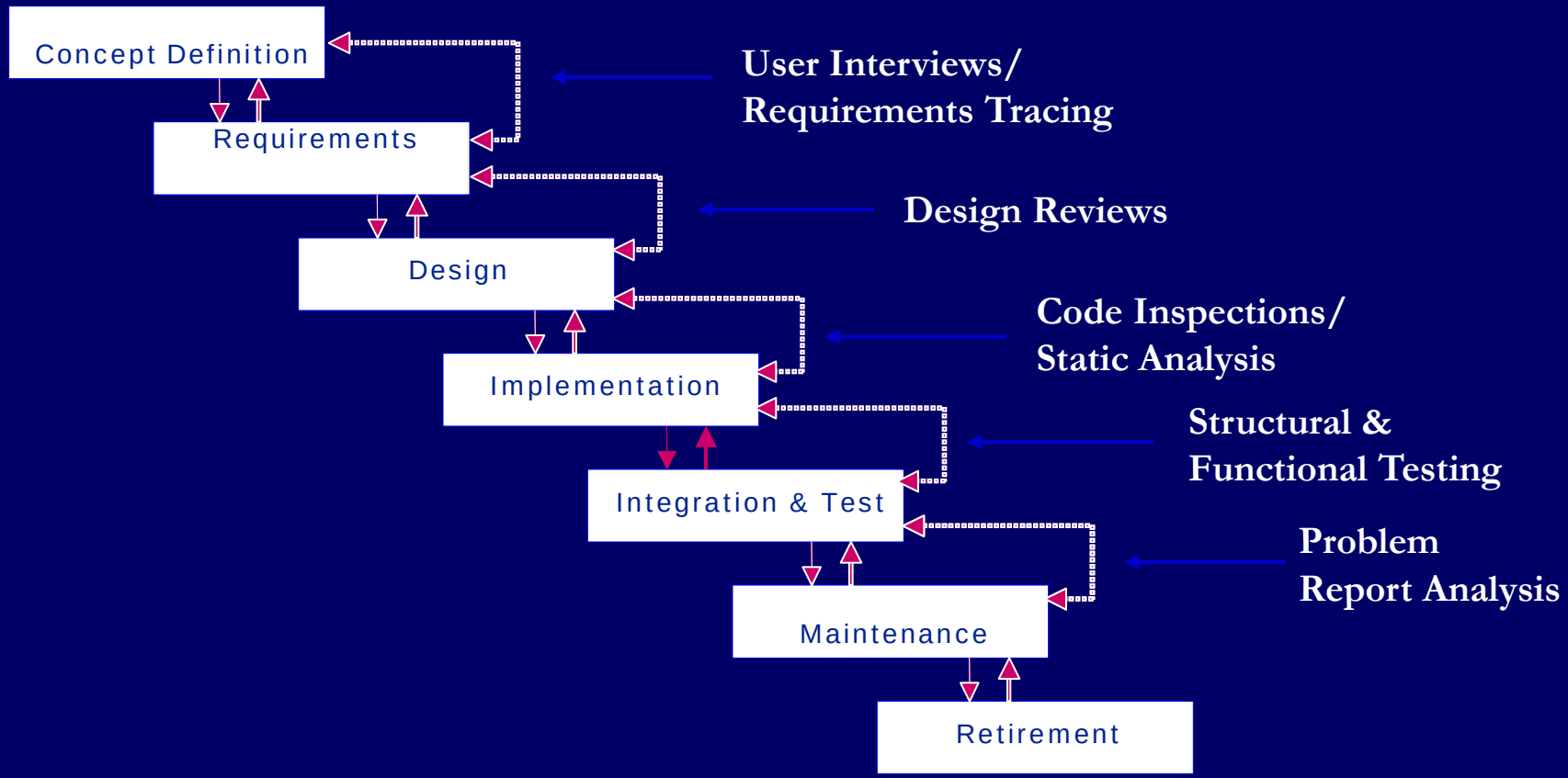
Waterfall model: Linear & Sequential



Traditional Artifacts

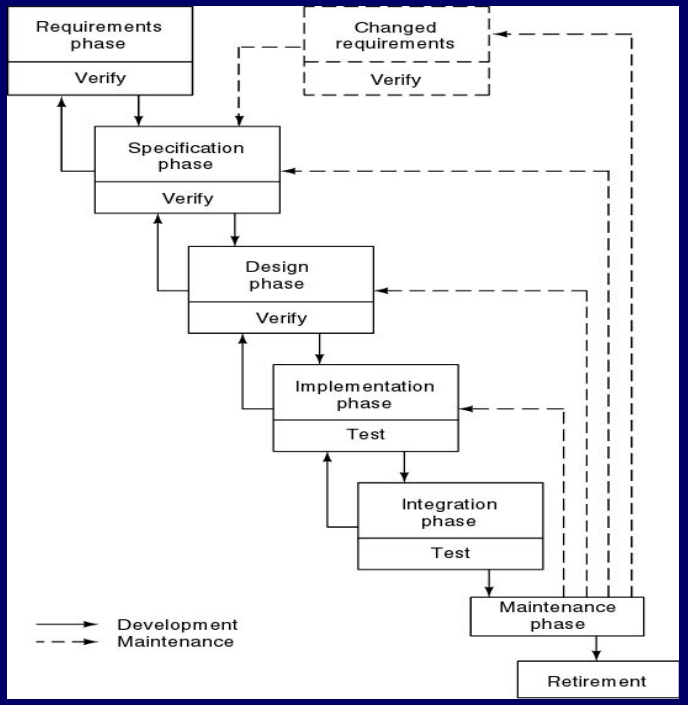


Verification Techniques



Waterfall Model Drawbacks

- sequential nature
- late tangible product maturity
  - late feedback
    to both customer and developer
  - minimal risk management
    for both customer and developer
- late testing maturity

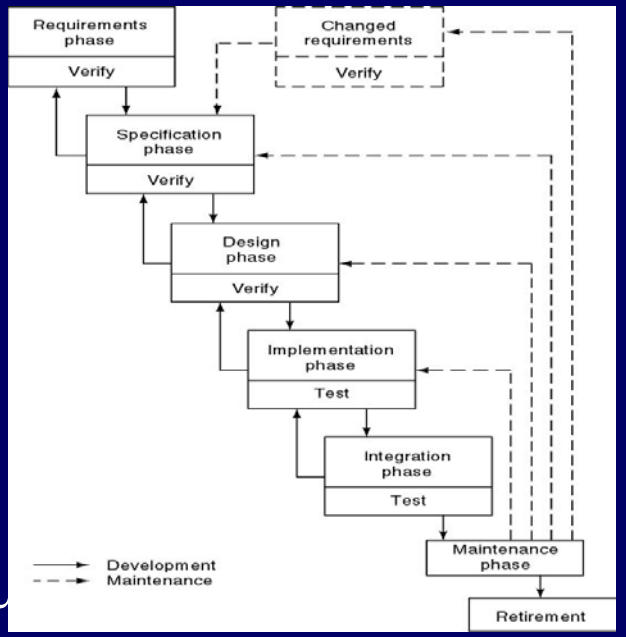


Pros and Cons of the Waterfall Method

Pros	Cons
1. Simple and easy to use. 2. Easy to manage due to the rigidity of the model – each phase has specific deliverables and a review process. 3. Phases are processed and completed one at a time. 4. Works well for smaller projects where requirements are very well understood.	1. Adjusting scope during the life cycle can kill a project 2. No working software is produced until late during the life cycle. 3. High amounts of risk and uncertainty. 4. Poor model for long and ongoing projects. 5. Poor model where requirements are at a moderate to high risk of changing

Winburg Case Study – The Real World Is Different

- **Episode 1**: The first version is implemented
- **Episode 2:** A fault is found
  - The product is too slow because of an implementation fault
- **Episode 3:** The requirements change
  - A faster algorithm is used
- **Episode 4:** A new design is adopted
  - Development is complete
- **Epilogue:** A few years later, these problems recu

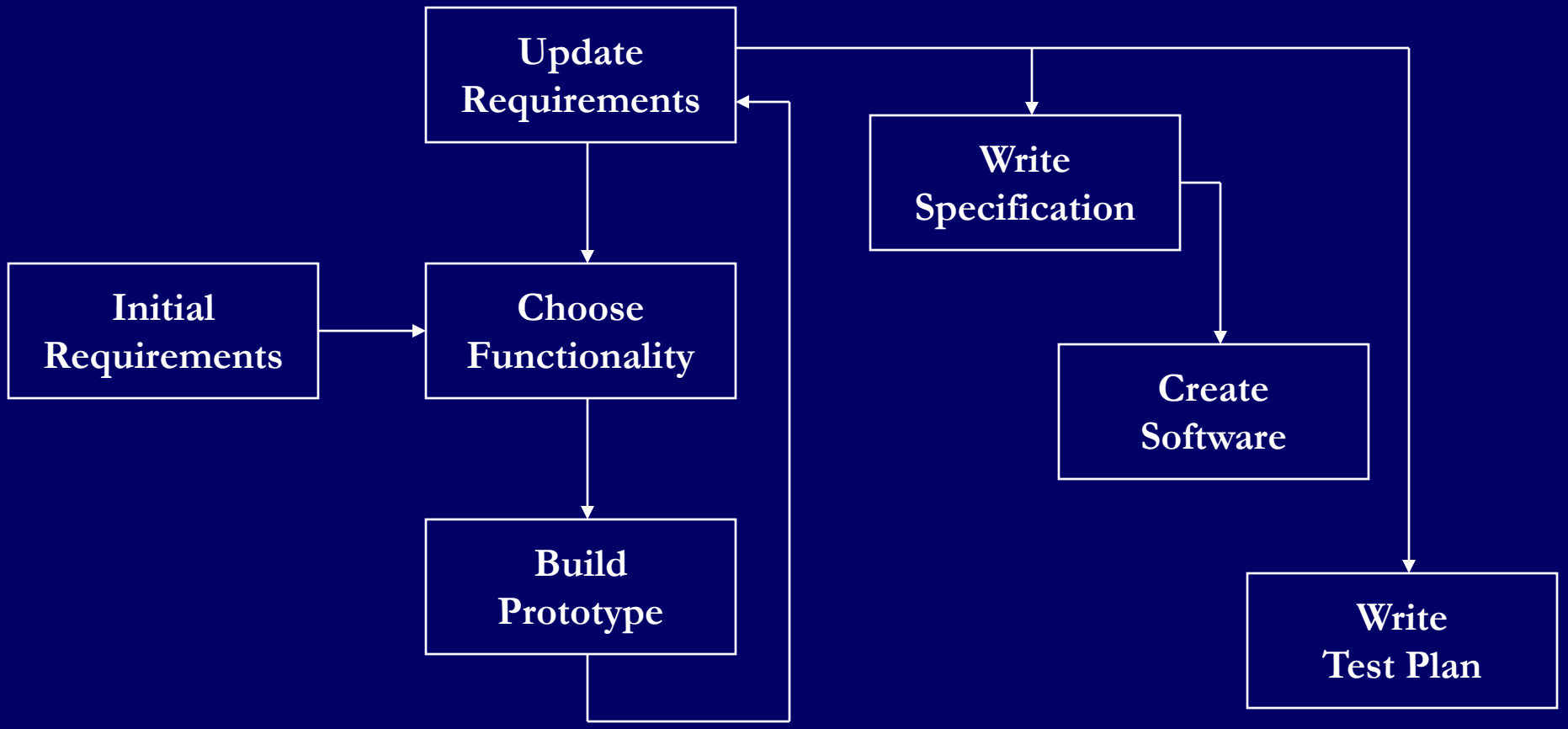


Moving Target Problem!!...

- Even if the reasons for the change are good, the software product can be adversely impacted
  - Dependencies will be induced
- Any change made to a software product can potentially cause a *regression fault*
  - A fault in an apparently unrelated part of the software
- If there are too many changes
  - The entire product may have to be redesigned and re-implemented
- Change is inevitable
  - Growing companies are always going to change
  - If the individual calling for changes has sufficient clout, nothing can be done about it
- There is no solution to the moving target problem

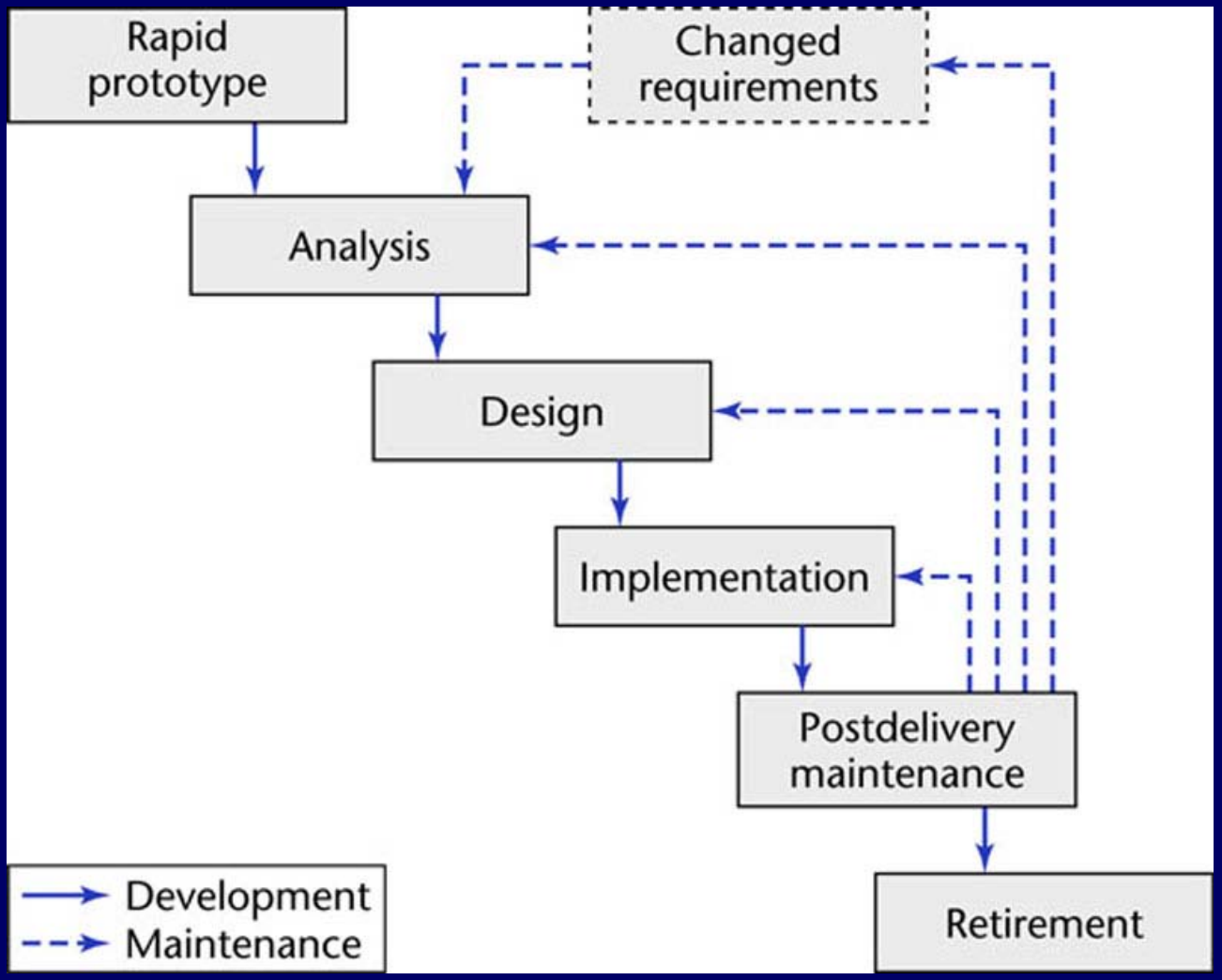
Rapid Prototyping

Rapid Prototyping + Waterfall



Rapid Prototyping Model

- Linear model
- "Rapid"



Motivation behind Rapid Prototype Model

- Increases likelihood that customers and developers are on the same page at time t0
- At t1 (>t0) the delivered function is higher for the rapid prototyping approa
- Shows overall, that function is closer to needs than the waterfall model

The Rapid Prototyping Model

- Goal: explore requirements
  - Without building the complete product
- Start with part of the functionality
  - That will (hopefully) yield significant insight
- Build a prototype
  - Focus on core functionality, not in efficiency
- Use the prototype to refine the requirements
- Repeat the process, expanding functionality

What is Prototyping?

- A definition (A. Davis):

  A prototype is a partial implementation of a system, constructed primarily to enable customer, end-user, or developer to learn about the problem and/or its solution.

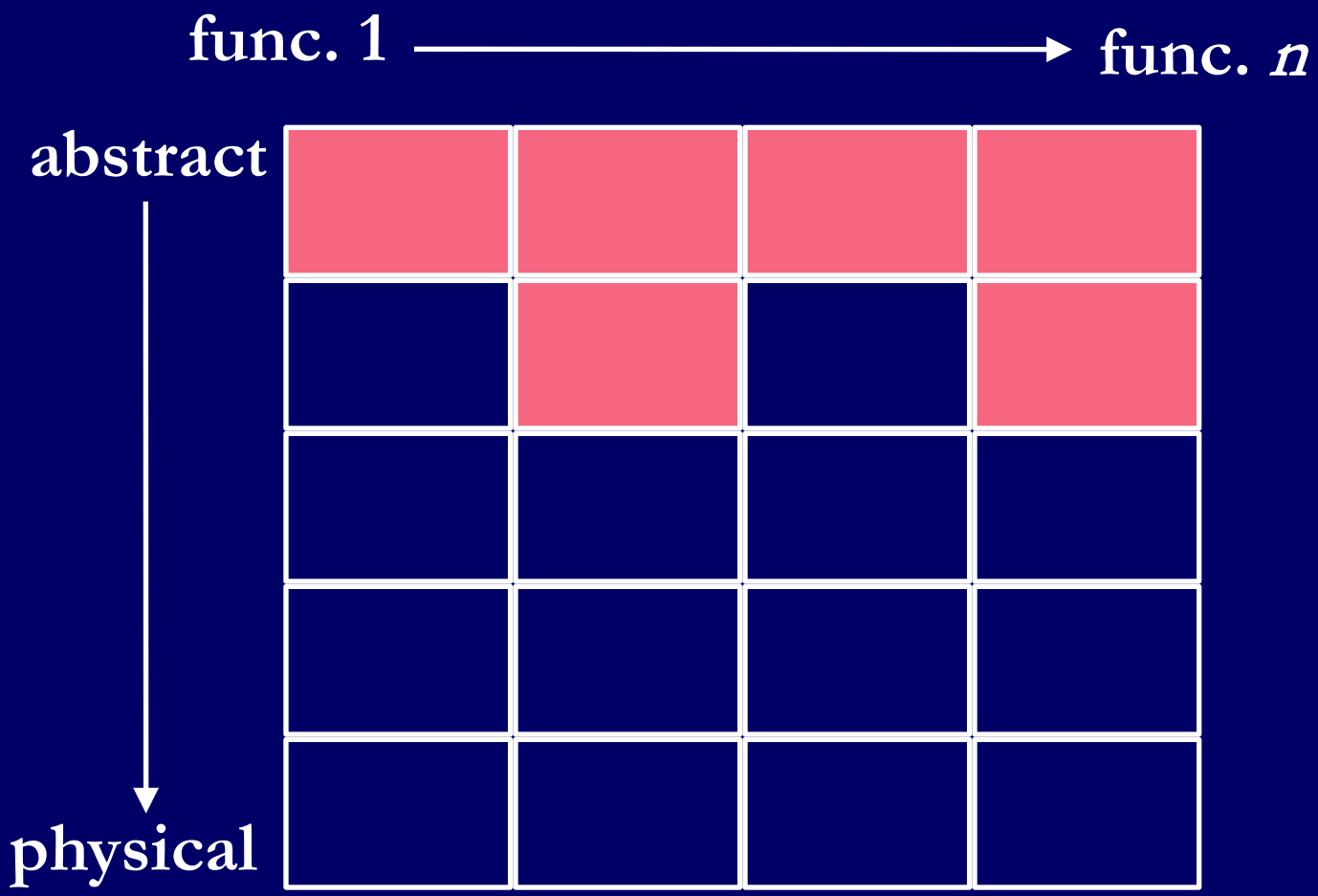
- Types:
  - evolutionary / throw-away
  - horizontal / vertical

The (Rapid) Prototyping Model

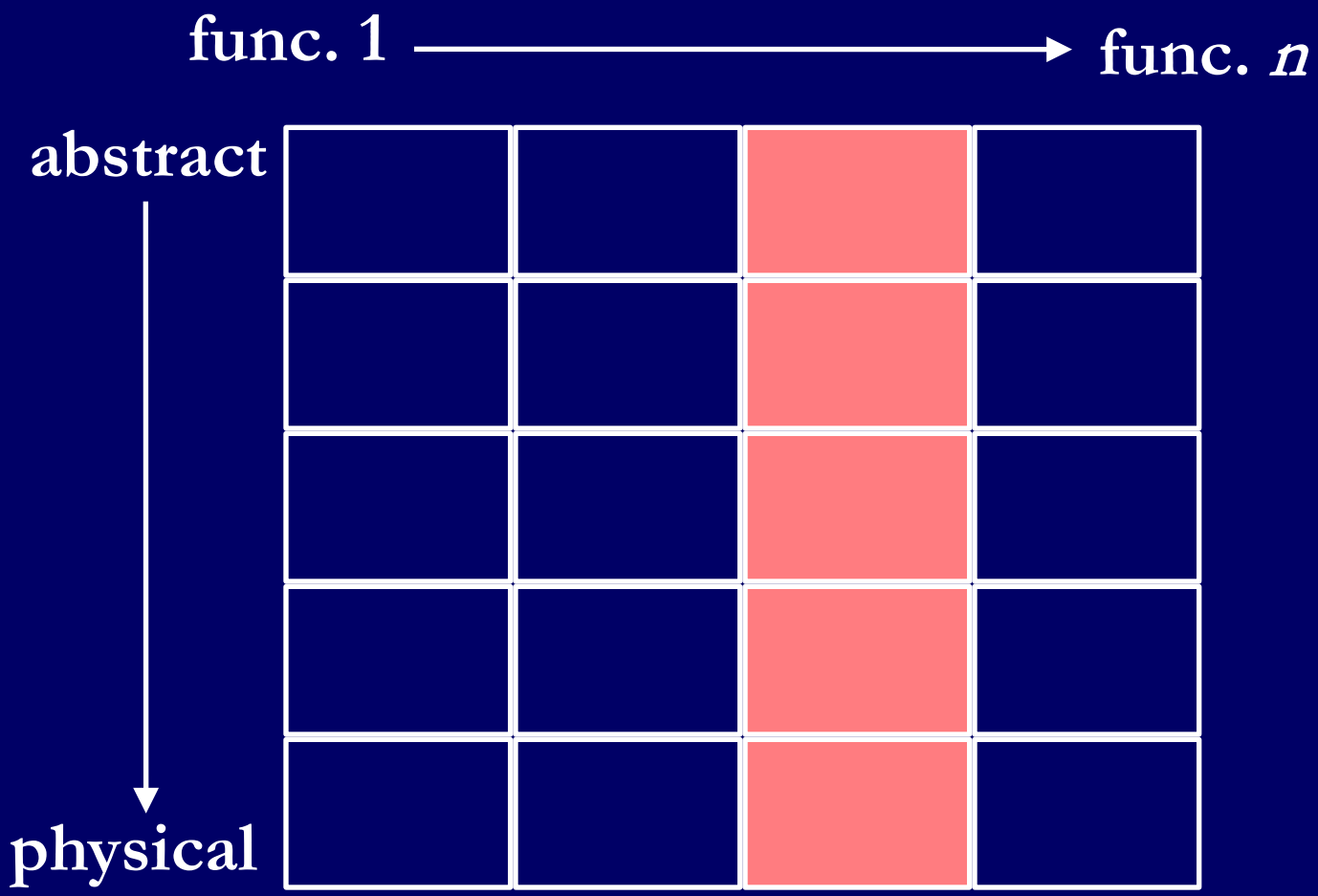
- **Goals:**
  - to break away from the sequential nature.
  - to speed up feedback.
  - to minimise risks

    for both customer and developer
  - to be incomplete but executable.
  - to be cheap and fast.

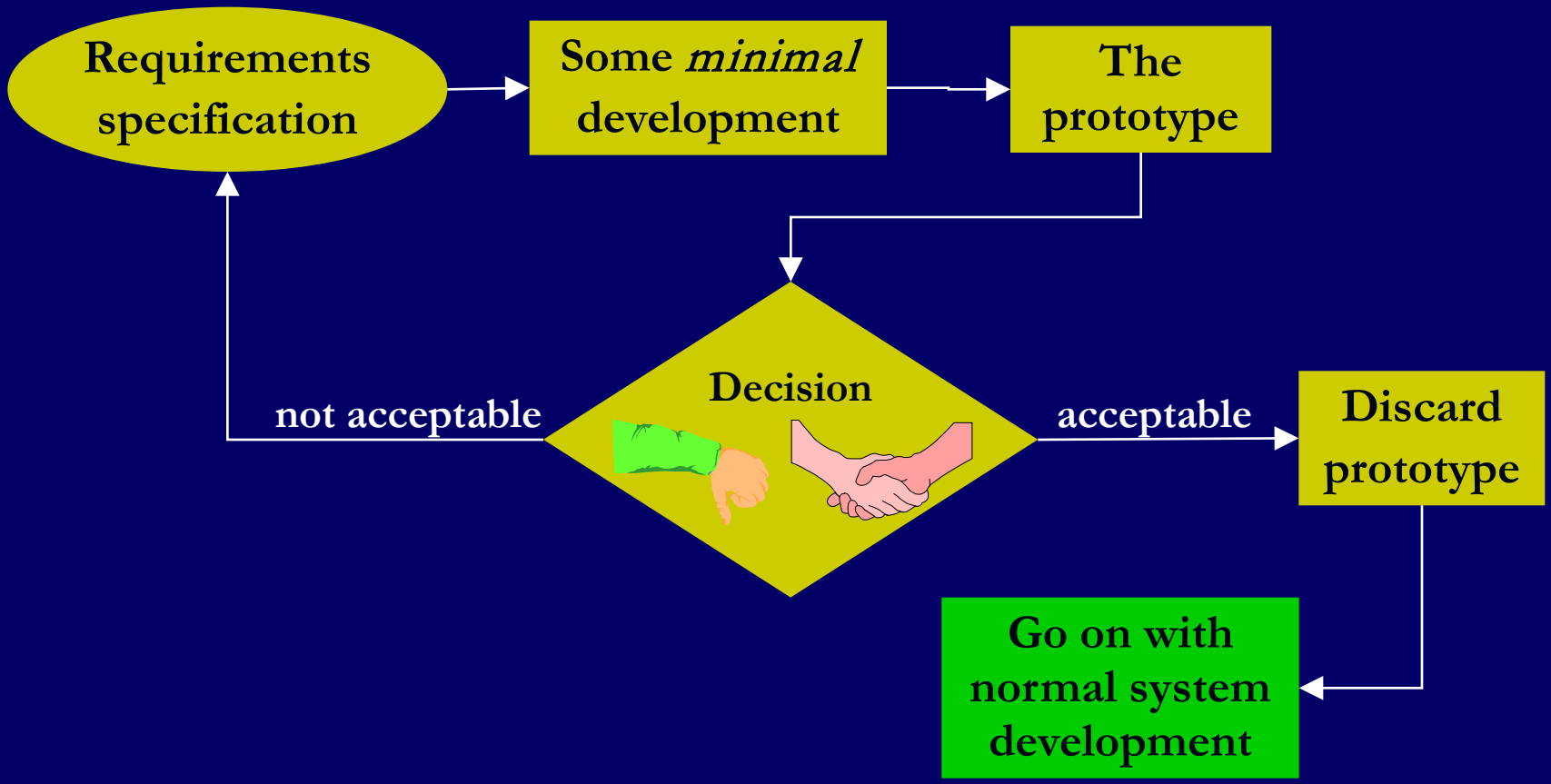
Horizontal Prototyping



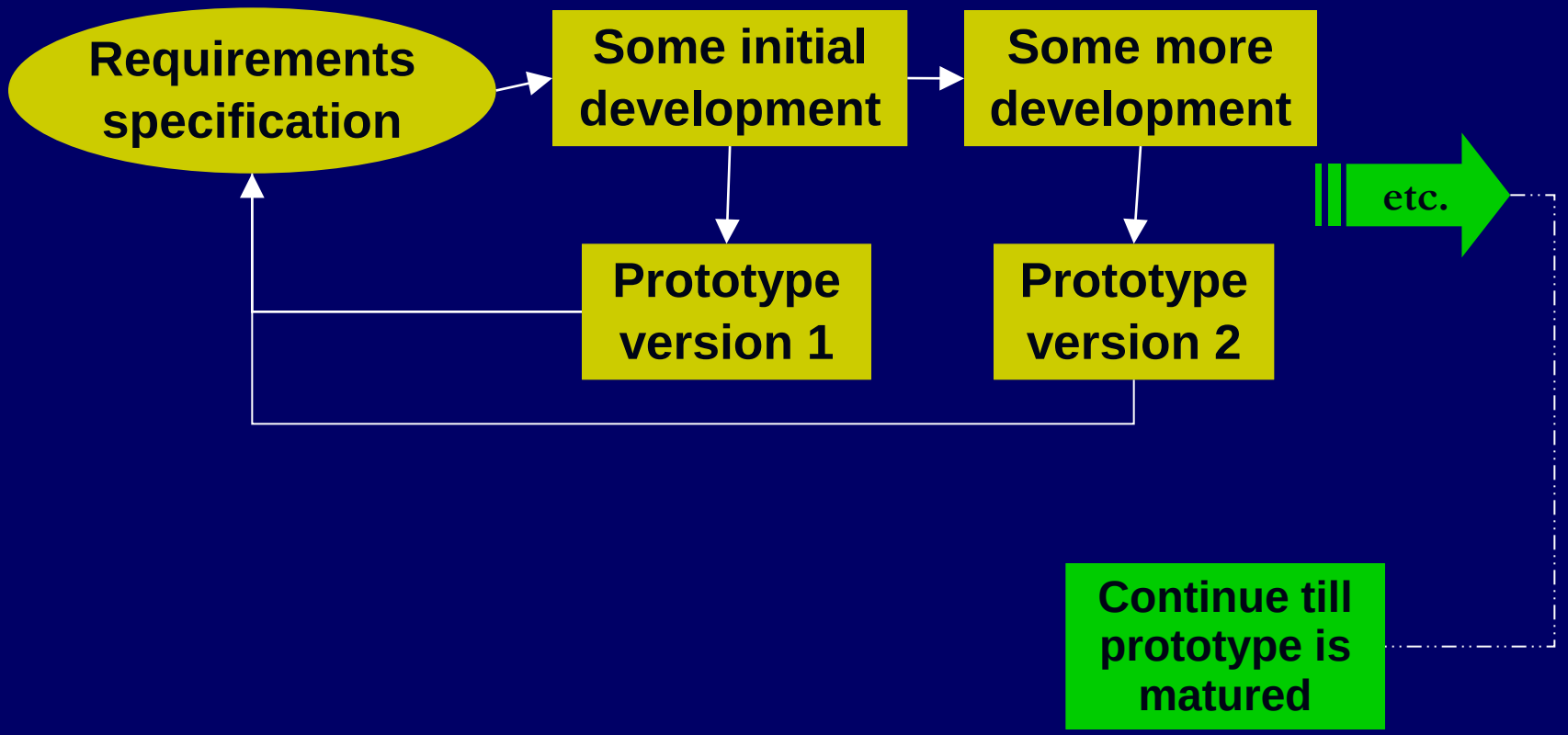
Vertical Prototyping



Throwaway Prototyping Model



A Visual Representation of The Evolutionary Prototyping Model

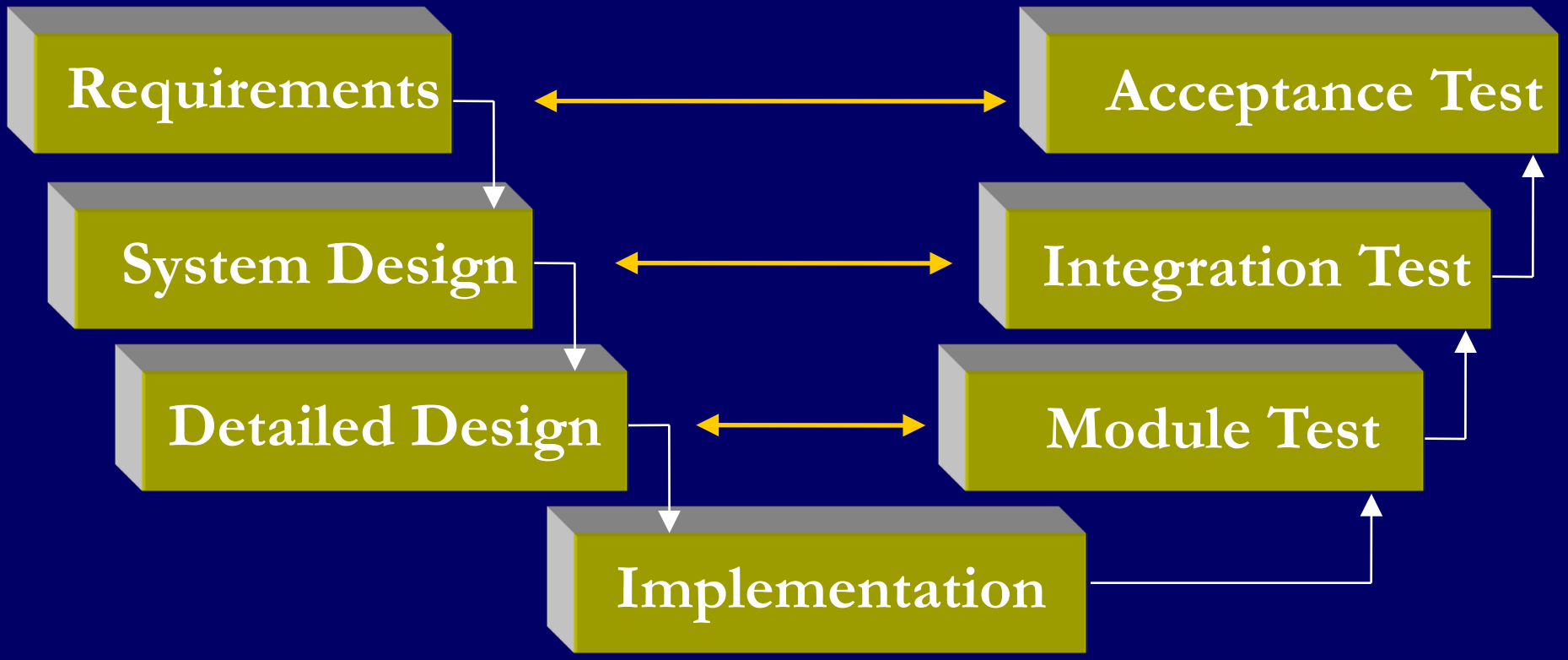


Analysis of The Prototyping Model

- Improves:
  - breaks the sequential nature.
  - supports fast feedback.
  - offers an opportunity for risk management.
- Problems:
  - has no definite (i.e. strictly defined) organisational structure.

V-Model

The V-Model


Requirements
Acceptance Test
System Design
Integration Test
Detailed Design
Module Test
Implementation

Analysis of the V-Model

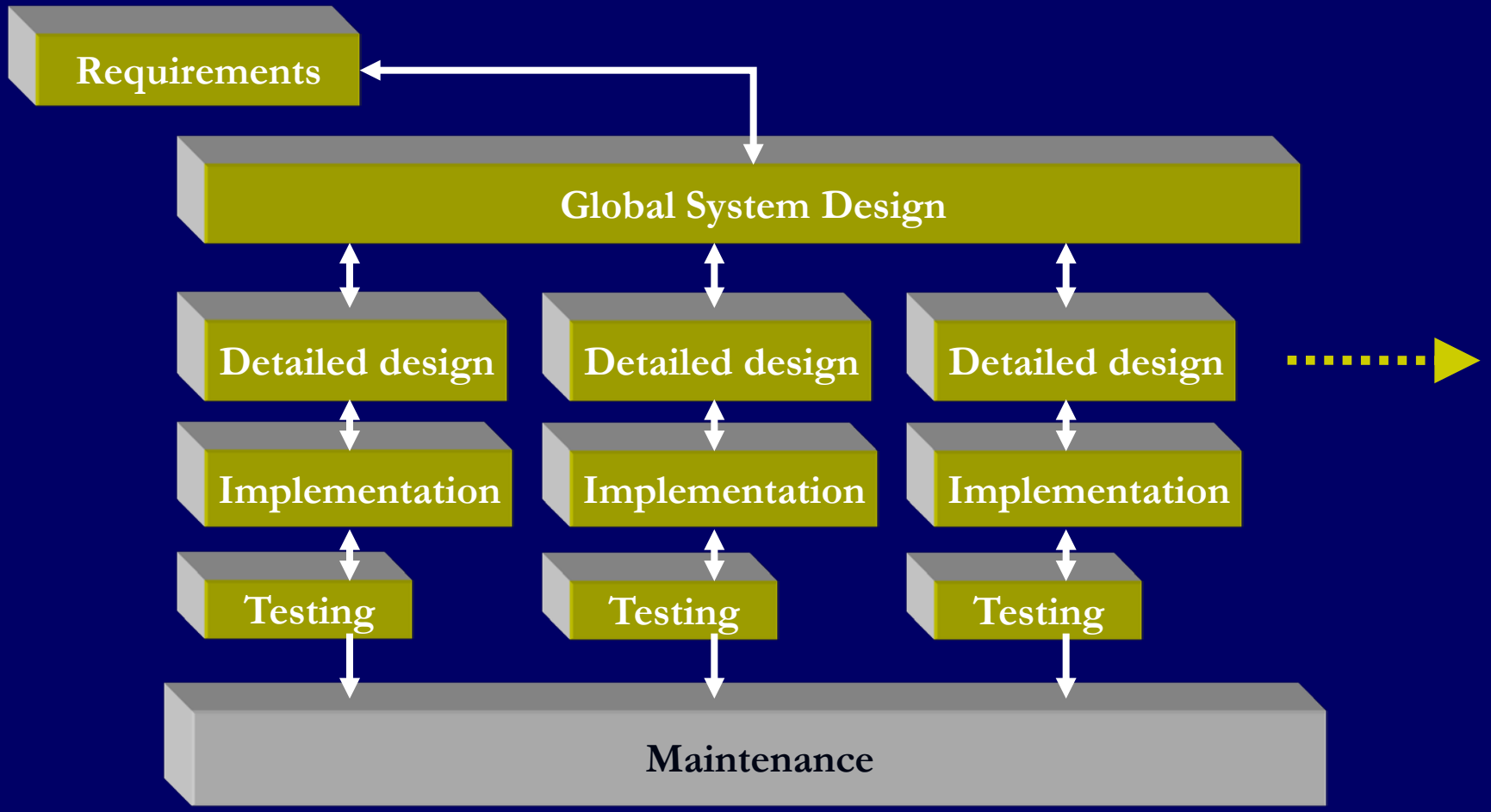
- Improves testing strategies
- Does not particularly improve:
  - sequential nature
  - feedback
  - developmental risk management

Miller's Law

- At any one time, we can concentrate on only approximately seven *chunks* (units of information)
- To handle larger amounts of information, use *stepwise refinement*
  - Concentrate on the aspects that are currently the most important
  - Postpone aspects that are currently less critical
  - Every aspect is eventually handled, but in order of current importance
- This is an *incremental* process

Incremental Model

The Incremental Model



Motivation behind Incremental Model

- Deliberately built to satisfy fewer requirements initially, but facilitates incorporation of new requirements which increases adaptability
- Initial development time is reduced because of limited functionality
- Software can be enhanced more easily for a longer period of time
- Stair steps show series of well-defined, planned, discrete builds of the system

Analysis of The Incremental Model

- Assumes independent sub-systems.
- Improves (by delivering smaller units):
  - feedback (in steps)
  - testing
- Avoids the production of a monolithic product.
- Does not particularly improve:
  - developmental risk management
  - Sequential nature (still present in sub-systems)

Incremental Model Strengths

- Develop high-risk or major functions first
- Each release delivers an operational product
- Customer can respond to each build
- Uses "divide and conquer" breakdown of tasks
- Lowers initial delivery cost
- Initial product delivery is faster
- Customers get important functionality early
- Risk of changing requirements is reduced

Incremental Model Weaknesses

- Still requires good planning and design..
- Requires early definition of a complete and fully functional system to allow for the definition of increments
- Well-defined module interfaces are required (some will be developed long before others)
- Total cost of the complete system is not lower

When to use the Incremental Model

- Risk, funding, schedule, program complexity, or need for early realization of benefits.
- Most of the requirements are known up-front but are expected to evolve over time
- A need to get basic functionality to the market early
- On projects which have lengthy development schedules
- On a project with new technology

Agile Model

Agile SDLC

- Somewhat controversial new approach…

- A collection of new paradigms characterized by
  - Less emphasis on analysis and design
  - Earlier implementation (working software is considered more important than documentation)
  - Responsiveness to change
  - Close collaboration with the client

Agile SDLC – cont'd

- Speed up or bypass one or more life cycle phases
- Usually less formal and reduced scope
- Used for time-critical applications

Manifesto for Agile Software Development

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

Some Agile Methods

- Adaptive Software Development (ASD)
- Feature Driven Development (FDD)
- Crystal Clear
- Dynamic Software Development Method (DSDM)
- Rapid Application Development (RAD)
- Scrum
- Extreme Programming (XP)
- Rational Unify Process (RUP)

Extreme Programming - XP

- For small-to-medium-sized teams developing software with vague or rapidly changing requirements
- Coding is the key activity throughout a software project
- Communication among teammates is done with code
- Life cycle and behavior of complex objects defined in test cases – again in code

XP Practices (1-6)

1. Planning game – determine scope of the next release by combining business priorities and technical estimates
2. Small releases – put a simple system into production, then release new versions in very short cycle
3. Metaphor – all development is guided by a simple shared story of how the whole system works
4. Simple design – system is designed as simply as possible (extra complexity removed as soon as found)
5. Testing – programmers continuously write unit tests; customers write tests for features
6. Refactoring – programmers continuously restructure the system without changing its behavior to remove duplication and simplify

XP Practices (7 – 12)

7. Pair-programming -- all production code is written with two programmers at one machine
8. Collective ownership – anyone can change any code anywhere in the system at any time.
9. Continuous integration – integrate and build the system many times a day – every time a task is completed.
10. 40-hour week – work no more than 40 hours a week as a rule
11. On-site customer – a user is on the team and available full-time to answer questions
12. Coding standards – programmers write all code in accordance with rules emphasizing communication through the code

XP is “extreme” because

Commonsense practices taken to extreme levels

- If code reviews are good, review code all the time (pair programming)
- If testing is good, everybody will test all the time
- If simplicity is good, keep the system in the simplest design that supports its current functionality. (simplest thing that works)
- If design is good, everybody will design daily (refactoring)
- If architecture is important, everybody will work at defining and refining the architecture (metaphor)
- If integration testing is important, build and integrate test several times a day (continuous integration)
- If short iterations are good, make iterations really, really short (hours rather than weeks)

Unusual Features of XP

- The computers are put in the center of a large room lined with cubicles
- A client representative is always present
- Software professionals cannot work overtime for 2 successive weeks
- No specialization
- *Refactoring* (design modification)

Evaluating Agile Processes and XP

- XP has had some successes with small-scale software development
  - However, medium- and large-scale software development is very different

- The key decider: the impact of agile processes on postdelivery maintenance
  - Refactoring is an essential component of agile processes
  - Refactoring continues during maintenance
  - Will refactoring increase the cost of post-delivery maintenance, as indicated by preliminary research?

Evaluating Agile Processes and XP (contd)

- Agile processes are good when requirements are vague or changing
- It is too soon to evaluate agile processes
  - There are not enough data yet
- Even if agile processes prove to be disappointing
  - Some features (such as pair programming) may be adopted as mainstream software engineering practices