CS 123
Introduction to Software Engineering

04: Software Life Cycle

DISCS

SY 2013-2014

Overview

- Software Life Cycle Activities
- Software Life Cycle Models
  - Code and Fix
  - Waterfall
  - Rapid Prototyping
  - Iterative and Incremental
  - Agile and XP
  - Open-source
  - Synchronize and Stabilize
  - Spiral

Learning Objectives

- To explain what Software Life Cycle (SLC) is
- To describe the different SLC models

Software Life Cycle (SLC)

- Sequence of different activities that take place during software development.
- From management's perspective:
  - Deliverables – usually tangible objects
  - Milestones – points/events that tells status of the project

Types of SLC Activities

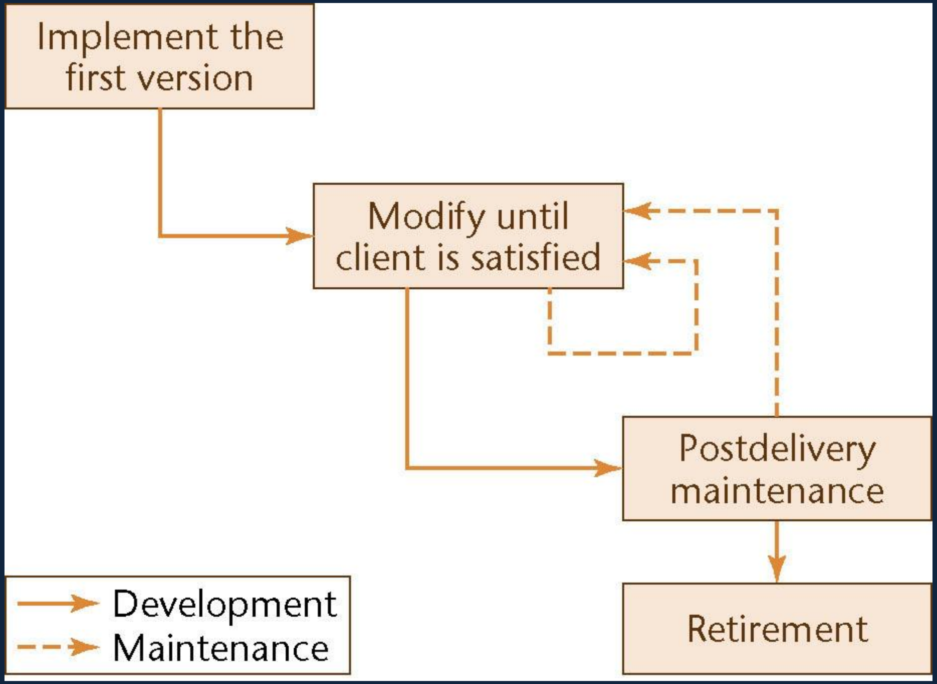
- Feasibility and Market Analysis
- Requirements Engineering
- Project Planning
- Design
- Implementation
- Testing
- Delivery
- Maintenance

SLC Models

- Code and Fix
- Waterfall
- Rapid Prototyping
- Iterative & Incremental
- Agile and XP
- Open-source
- Synchronize and Stabilize
- Spiral

Code and Fix Model

- No requirement
- No design
- No specifications
- Maintenance nightmare

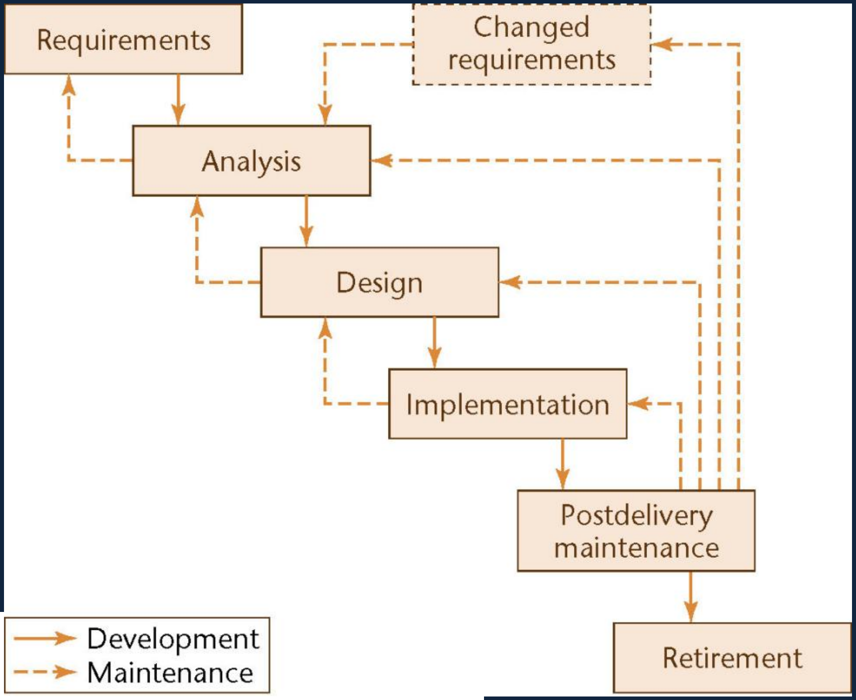


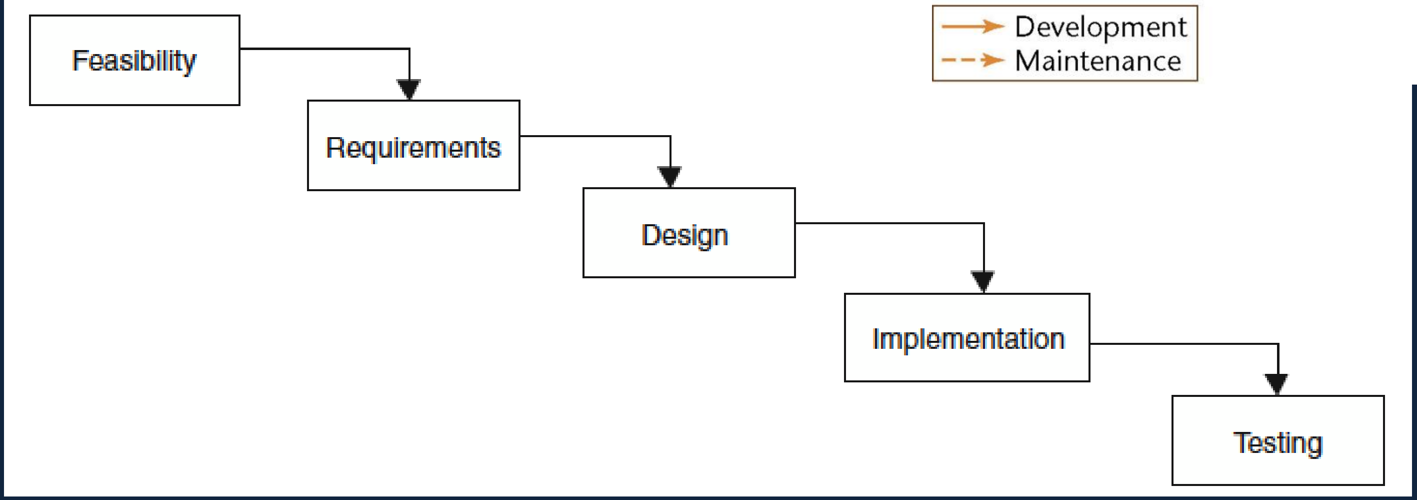
Code and Fix Model

- The easiest way to develop software
  - Works well for 100-200 LOC
  - Programming exercise
- The most expensive way
- The most difficult to maintain
- The worst Software Life Cycle

Waterfall Model

- aka Linear Sequential Model
  - Generally one phase follows after completion of another
- Many versions





Waterfall Model

- No phase is complete until documentation of that phase has been approved by Software Quality Assurance (SQA) group
- Modification of documentation is also checked by SQA (feedback)
- Testing is not a separate phase but done throughout the software process

- Characterized by:
  - Feedback loops
  - Documentation-driven
  - Some use DFD and UML

Advantages of Waterfall Model

- Standardized series of steps in software development
- Ensures that no important areas had been overlooked
- Formal contract exist between users & developers → Evidence to arbitrate disputes
- Suitable for large projects with any people
- Enforced Discipline approach with meticulous check by SQA
- Documentation
- Maintenance is easier

Drawbacks of Waterfall Model

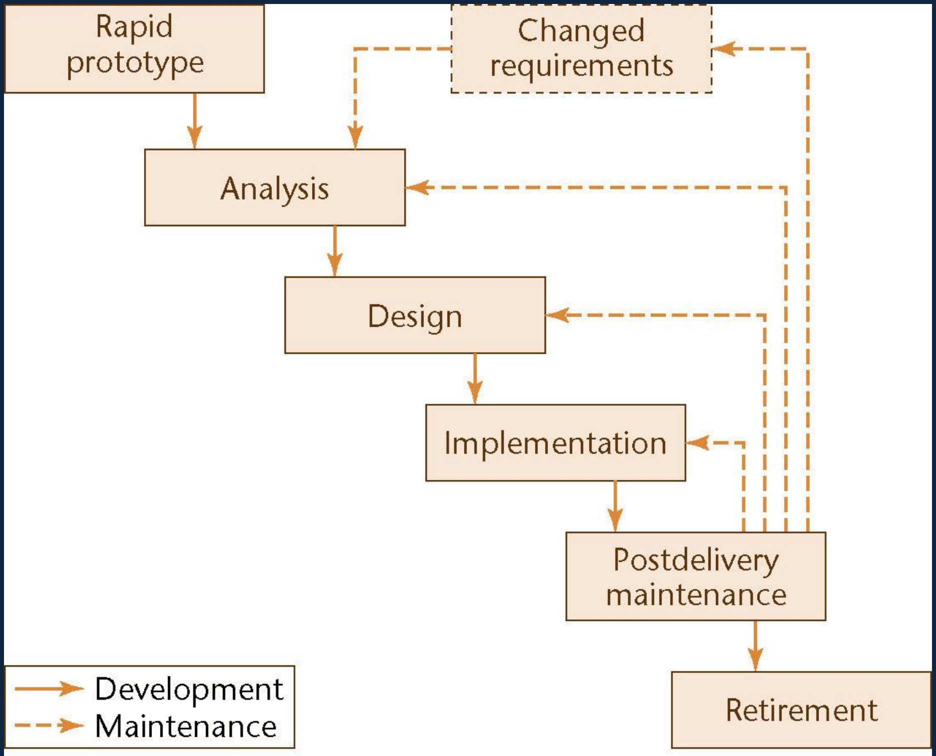
- Too rigid for the change of user requirement
- Poor communication between user & developer → only based on written specification & documentation
- Poor user’s involvement

Detailed Problems of Waterfall Model

- Specification document & diagram hardly understood by users (boring)
- Specification & diagrams do not describe how the finished product will work
- User just sign documents they don’t understand
- Very low user involvement (only during interview of requirement)
- User see working product *after* entire product has been coded
- “I know this is what I asked for but this is not what I wanted”

Rapid Prototyping Model

- Build a throwaway version (prototype)
- Intended to test concepts and requirements
- Promotes clarity among all stakeholders
- Effort spent pays off for the clarity
- After agreement from customer, usually same phases as waterfall



"Rapid" prototype is build first

Rapid Prototyping Model

- Rapid Application Development (RAD) emphasizes User involvement, prototype, reuse, and automated tools
- Involves highly trained team to build prototypes rapidly. SWAT = Skilled With Advanced Tools Team

Features of a Prototype

- Mostly GUI
- No Error checking
- No real access to database/real network
- No help screen
- Little use of Options
- Useful for user to try out, react to, comment on & finally approve with confidence that it meets their needs
- Missing features can be added later on

Advantages of Prototyping

- Improves flexibility: user can change the requirement during prototype
- Clearer requirement
- Problem can be detected early
- Improves communication between user & developer
- Suitable for in-house development (developer are paid by time, not as project based)
- Prototype gives insight to design team

Disadvantages of Prototyping

- Extension of Development schedules
- Requires more experienced team members to build a prototype rapidly
- Tendency of users tp make unnecessary changes that do not improve the usefulness of the finished product
- Nearly finish appearance & interface of prototype may mislead users into thinking that the system is nearly done (difficult to explain to users)

Iterative & Incremental Model

- Proposed by Jacobson, Booch & Rumbaugh 1999, fathers of OO & UML
- Ideas based on Miller‘s Law
- Stepwise refinement: the philosophy of continuous improvement until you reach the final target
- Compared with single shot building of classical SLC, it is much better

Iteration and Incrementation

- In real life, we cannot speak about “the analysis phase”
- Instead, the operations of the analysis phase are spread out over the life cycle
- The basic software development process is iterative
  - = Each successive version is intended to be closer to its target than its predecessor

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

Iteration and Incrementation

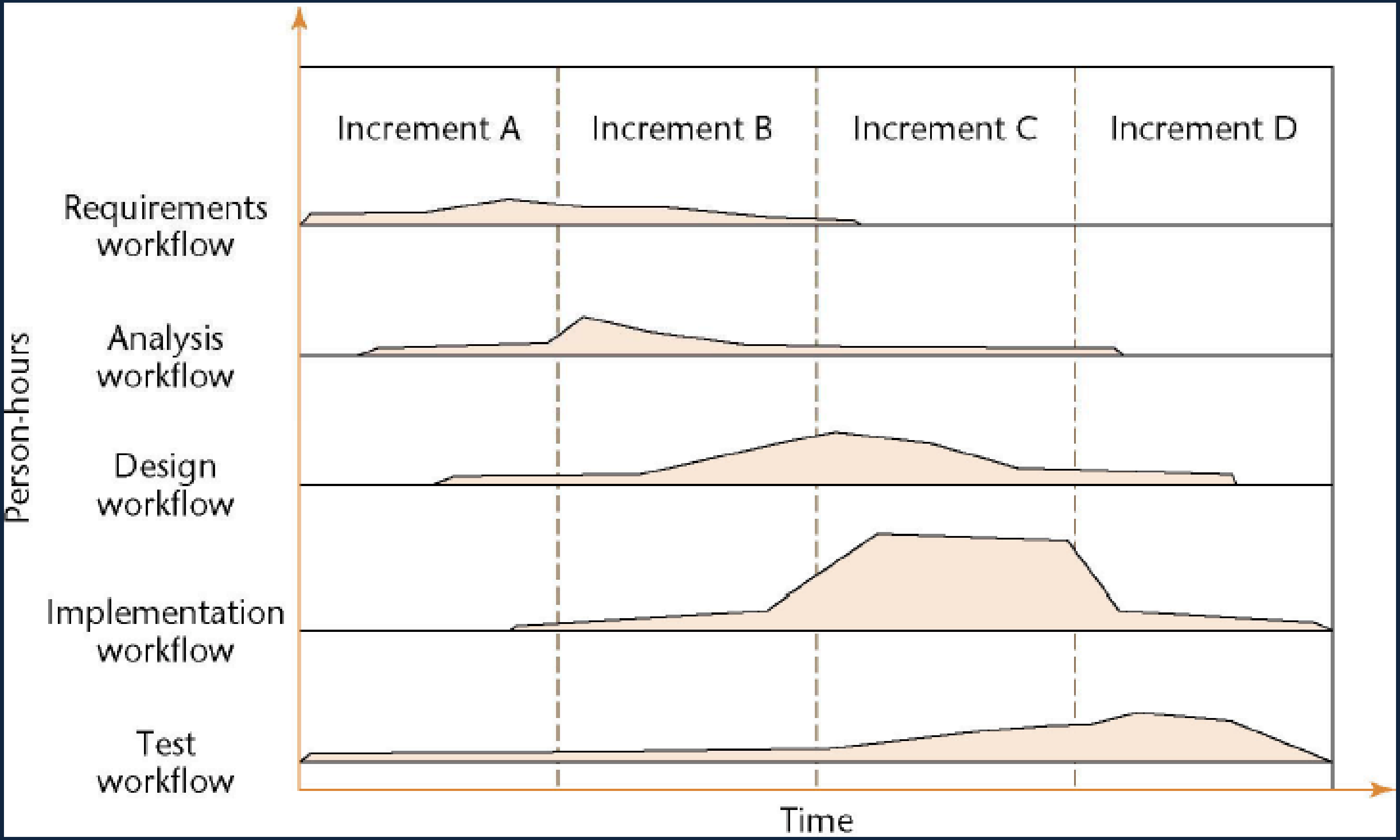
- Incrementation:

1. List all items (e.g. Requirements)
2. Sort by the order of importance
3. Consider only the first 7 most important items
4. In next iteration, consider the next 7 most important items
5. And so on

Iteration and Incrementation

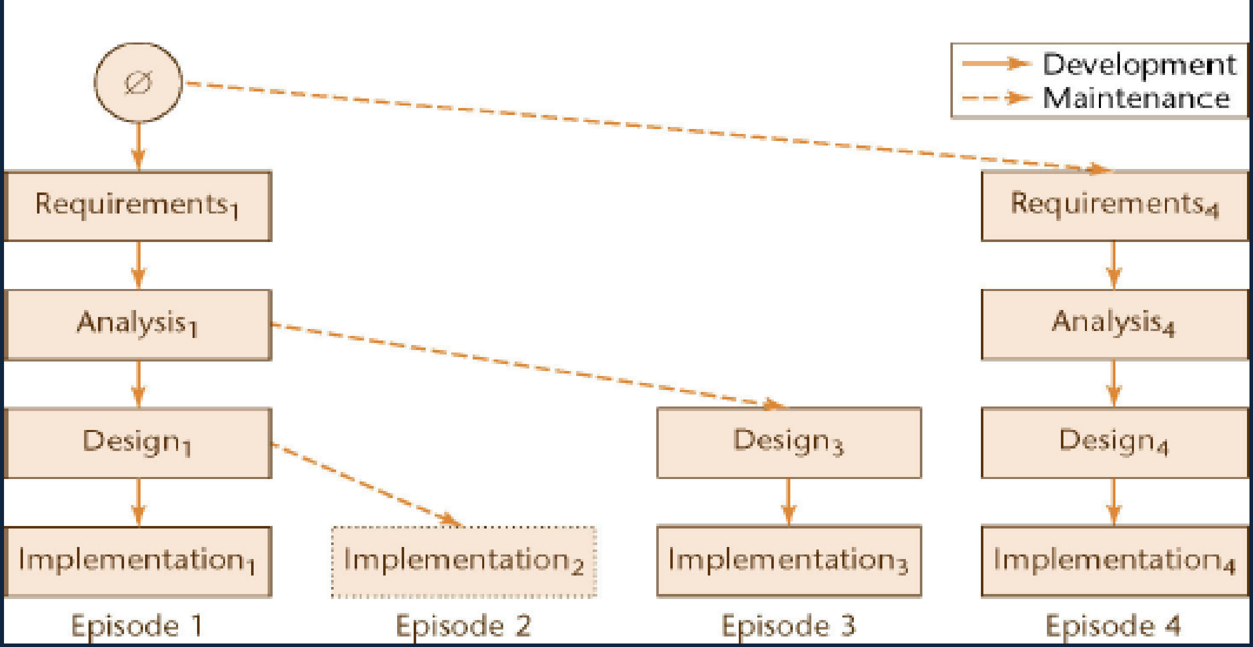
- Iteration & Incrementation are intrinsic aspect of SE
- Iteration = continuous improvement (of classical life cyle) such that each successive version is intended to be closer to its target than its predecessor
- Increment = piece by piece construction of software product. Each increment goes through multiple versions (iteration)

Iteration and Incrementation



Iteration and Incrementation

- Iteration and incrementation are used in conjunction with one another
- There is no single “requirements phase” or “design phase”
- Instead, there are multiple instances of each phase
- The number of increments will vary — it does not have to be four



Strengths of the Iterative & Incremental Model

- There are multiple opportunities for checking that the software product is correct
  - Every iteration incorporates the test workflow
  - Faults can be detected and corrected early → reduce overall cost
- The robustness of the architecture can be determined early in the life cycle
  - *Architecture* – check the various component modules and how they fit together
  - *Robustness* – the property of being able to handle extensions and changes without falling apart

Strengths of the Iterative & Incremental Model

- We can *mitigate* (resolve) risks early
  - Risks are invariably involved in software development and maintenance
- We have a working version of the software product from the start
  - The client and users can experiment with this version to determine what changes are needed
- Variation: Deliver partial versions to smooth the introduction of the new product in the client organization

Strengths of the Iterative & Incremental Model

- There is empirical evidence that the Iterative & Incremental life cycle model works
- The CHAOS reports of the Standish Group show that the percentage of successful products increases
- One factor associated with successful projects was the use of an iterative process

Managing Iteration & Incrementation

- The iterative-and-incremental life-cycle model is as well-organized as the waterfall model …
- … because the iterative-and-incremental life-cycle model *is* the waterfall model, applied successively
- Each increment is a waterfall mini project

Motto:
“Software is not built, it grows” - *Hans Van Vliet*

Agile and XP

- Somewhat controversial new approach
- *Stories* (features client wants)
  - Estimate duration and cost of each story
  - Select stories for next build
  - Each build is divided into tasks
  - Test cases for a task are drawn up first
- Pair programming
- Continuous integration of tasks

What Do We Mean By “Agile”?

- According to the Merriam-Webster on-line dictionary “agile” means:
  - “1: marked by ready ability to move with quick easy grace;”
  - “2: having a quick resourceful and adaptable character.”
- In agile software development, “agile” tends to mean “the ability to respond to change.”

Agile Processes

- Seventeen software developers (later dubbed the “Agile Alliance”) met at a Utah ski resort for two days in February 2001 and produced the *Manifesto for Agile Software Development*
- Recognized a need for an alternative to documentation-driven, heavyweight software development processes which doesn’t work for ALL projects
- The Agile Alliance did not prescribe a specific life-cycle model
- Instead, they laid out a group of underlying principles

Agile Processes

- Agreed on a “manifesto” of values and principles
  - Individuals and interactions over processes and tools
  - Working software over comprehensive documentation
  - Customer collaboration over contract negotiation
  - Responding to change over following a plan

Agile Processes

- A principle in the *Manifesto* is
  - Deliver working software frequently
  - Ideally every 2 or 3 weeks
- One way of achieving this is to use *timeboxing*
  - Used for many years as a time-management technique
- A specific amount of time is set aside for a task
  - Typically 3 weeks for each iteration
  - The team members then do the best job they can during that time

Agile Processes

- Agile processes are 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 Methodologies - Examples

- SCRUM
- Dynamic Systems Development Method (DSDM)
- Popular in UK; 9 practices similar to XP; being adopted by UK government
- Crystal Family & Adaptive Software Development (merged in 2001)
- Feature-Driven Development (FDD – Coad)
- Pragmatic Programming
- dX (agile form of RUP)
- **Extreme Programming (XP)**

Agile Assumptions

- The design shall be simple and the code quality must be high
- The customer can change their mind, substitute functionality, and change priorities
- Delivering the most value to the business, efficient use of resources, maximize ROI and time-to-ROI

Agile Methodologies ask…

- If design is good, why not make it everyone’s job?
- If simplicity is good, why not use the simplest design that supports the currently desired functionality?
- If architecture is good, why not have everyone work at defining and refining the architecture continuously?
- If short iterations are good, why not make iterations really short (hours and days) instead of weeks and months?
- If requirements, design, and code reviews are good, why not do it all the time?
- If testing is good, why not do it all the time… even customers?
- If integration testing is good, why not do it several times a day?

Agile Methodologies

- Seek to address these questions and the “realities” of software development
- Maintain a repeatable, quality-driven process

  Properties of an Agile Methodology:
  - Iterative development
    - Short iterations (2-6 weeks)
    - Working versions at conclusion of each iteration
    - Fully integrated and tested
  - Adaptable: can evolve with each iteration
  - People-centric: developers & management equal

Agile Processes

- It gives the client confidence to know that a new version with additional functionality will arrive every 3 weeks
- The developers know that they will have 3 weeks (but no more) to deliver a new iteration
  - Without client interference of any kind
- If it is impossible to complete the entire task in the timebox, the work may be reduced (“descoped”)
  - Agile processes demand fixed time, not fixed features

Agile Processes

- Another common feature of agile processes is *stand-up meetings*
  - Short meetings held at a regular time each day
  - Attendance is required
- Participants stand in a circle
  - They do not sit around a table
  - To ensure the meeting lasts no more than 15 minutes

Agile Processes

- At a stand-up meeting, each team member in turn answers five questions:
  - What have I done since yesterday’s meeting?
  - What am I working on today?
  - What problems are preventing me from achieving this?
  - What have we forgotten?
  - What did I learn that I would like to share with the team?

Agile Processes

- The aim of a stand-up meeting is
  - To raise problems
  - Not solve them
- Solutions are found at follow-up meetings, preferably held directly after the stand-up meeting

Agile Processes

- Stand-up meetings and timeboxing are both
  - Successful management techniques
  - Now utilized within the context of agile processes
- Both techniques are instances of two basic principles that underlie all agile methods:
  - Communication; and
  - Satisfying the client’s needs as quickly as possible

Evaluating Agile Processes

- Agile processes have 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 post-delivery 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?

What Is Extreme Programming?

- A system of practices that a community of software developers is evolving to address the problems of quickly delivering quality software, and then evolving it to meet changing business needs.
- Extreme programming is a software methodology, which has a set of simple practices to be followed
- Overall, the methodology emphasizes team work, customer participation and concentration in the essential.
- XP is a specific instantiation of an agile process

What is XP?

- Who is behind XP?
  - Kent Beck, Ward Cunningham, Ron Jeffries
- Short definition
  - lightweight process model for OO software development
- What’s in the name?
  - code is in the centre of the process
  - practices are applied extremely
- What is new in XP?
  - none of the ideas or practices in XP are new
  - the combination of practices and their extreme application is new
  - XP is not intended to be a complete framework

Emergence

- XP provides values and principles to guide team behavior
  - Team is expected to self-organize
  - XP provides specific core practices
  - Each practice is simple and self-complete
  - Combination of practices produces more complex emergent behavior
  - Synergy of practices still not fully understood

Unusual Features of XP

- XP is one of a number of new paradigms collectively referred to as *agile processes*
- 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)

Why Is It Called "Extreme"?

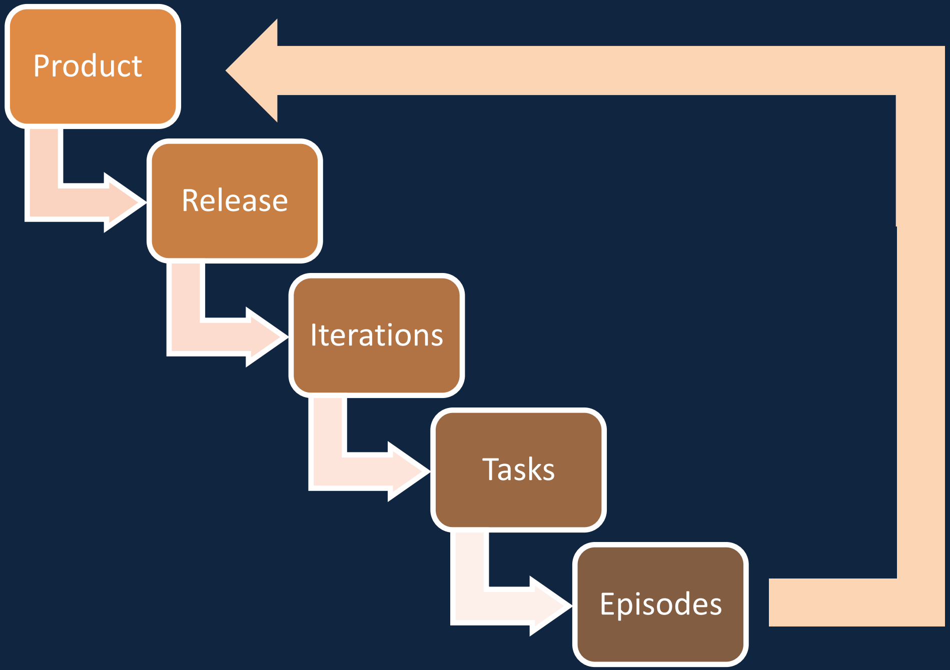
- Selected the minimal set of effective practices
- "Turned the knob up to 10" on each practice
  - Very short cycles (planning game)
  - Continuous code reviews (pair programming)
  - Extensive testing (unit testing, acceptance testing)
  - Continuous integration
  - Constant design improvement (refactoring)
  - Continuous architecture refinement (metaphor)
  - Etc…

Growing a System

- Have a running system from day 1.
- Integrate stories one by one
- Work with small releases, iterations, tasks, episodes:
  - Each iteration approx. 3 weeks
  - Split (combine) stories into tasks, estimate
  - Design, test, build, test
  - Measure progress, learn to estimate

XP Process Cycle

- XP is iterative and incremental
- XP is driven by time-boxed cycles
- The rhythm (below) of the XP process is crucial



Forward-Driving Activities

- Each level of activity provides the *minimal* materials needed for the next level
  - Product activities provide materials for release cycles – requirements and priorities
  - Release planning sessions provide materials for iteration cycles – prioritized & sized stories
  - Iteration planning sessions provide materials for task cycles – task breakdowns
  - Task development provides materials for development episodes
  - Development episodes produce product

Product

- Involves chartering, strategy planning, feature set definition and planning, investment and resource commitments…
- Tends to be organizationally context dependent
- XP does not provide specific practices for this
- XP assumes the Customer does these things
- Primary deliverable: stories

Releases

- Whole team gathers
- Review prior progress
- Customer presents stories
- Stories are discussed (analysis)
- Developer determines technical approach & risk (architecture & design)

Releases

- Developer provides first-level estimates & options
- Customer prioritizes stories
- Customer chooses target release time box
- Stories arranged into probable iterations
- Begin the next iteration
- Primary deliverable: release plan
- Releases are typically from 1 to 6 months

Planning a Release

- Release: every 2 - 6 months
  - Fixed *date*, *cost* and *quality*
  - Determine *scope*: how many stories can be done following development estimates
  - Most important user stories first
- Feedback / adjustment at *every* iteration
  - New / modified stories
  - Changed estimates

Iteration

- Each iteration is 1-3 weeks long.
- For each iteration, the customer chooses the user stories to be implemented.
- Rule: Choose more valuable first. (This way, you will focus on the most important parts.)

Iteration

- Also, choose which user stories to fix, if there are some that did not pass their acceptance tests.
- Choose such an amount of user stories that based on the velocity estimates, they will be completed within the iteration.
- Preliminary deliverable: iteration plan
- Begin the development of the tasks
- Final deliverable: a deployable system

Tasks

- Developer signs up for a task
- Developer begins episodes to implement
- Developer ensures task is complete
- If last task, Developer ensures story is complete via acceptance tests

Task planning

- Programming tasks are identified from the user stories and failed tests.
- The tasks are written down on index cards.
- Duplicates are removed.
- Developers choose tasks and estimate their duration (1-3 ideal days ie. days without interruption – shorter ones can be grouped and longer ones divided).
- After this, it is possible to evaluate how full the iteration is.

Episodes

- = daily development work
- Developer obtains a pair partner
- Pair verifies understanding of story for this task (analysis)
- Pair determines detailed implementation approach (detailed design)
- Pair begins test-driven cycle of write test, implement to pass, refactor
- At appropriate intervals, pair integrates to code base
- Pair retrospects on progress frequently
- Pair continues until pair changes or task complete

Feedback

- Pairs are constantly communicating within themselves and outward to team
- Daily “stand-up” meetings provide overall team status, resynchronization, and micro-planning
- Retrospectives provide overall status and points for process adjustment and improvement
- Development cycles may cause rethinking of tasks
- Task development may cause rethinking of stories
- Story re-estimation may cause iteration changes or recovery
- Iteration results may cause changes to release plan

Acronyms of Extreme Programming

- YAGNI (you aren’t gonna need it)
- DTSTTCPW (do the simplest thing that could possibly work)
- A principle of XP is to minimize the number of features
- There is no need to build a product that does any more than what the client actually needs

Open Source

- It can be extremely successful for infrastructure
- projects, such as
- Operating systems (Linux, OpenBSD, Mach, Darwin)
- Web browsers (Firefox, Netscape)
- Compilers (gcc)
- Web servers (Apache)
- Database management systems (MySQL)
- Check:
  - SourceForge.Net, FreshMeat.Net, GNU.Org

Open-Source Life-Cycle Model

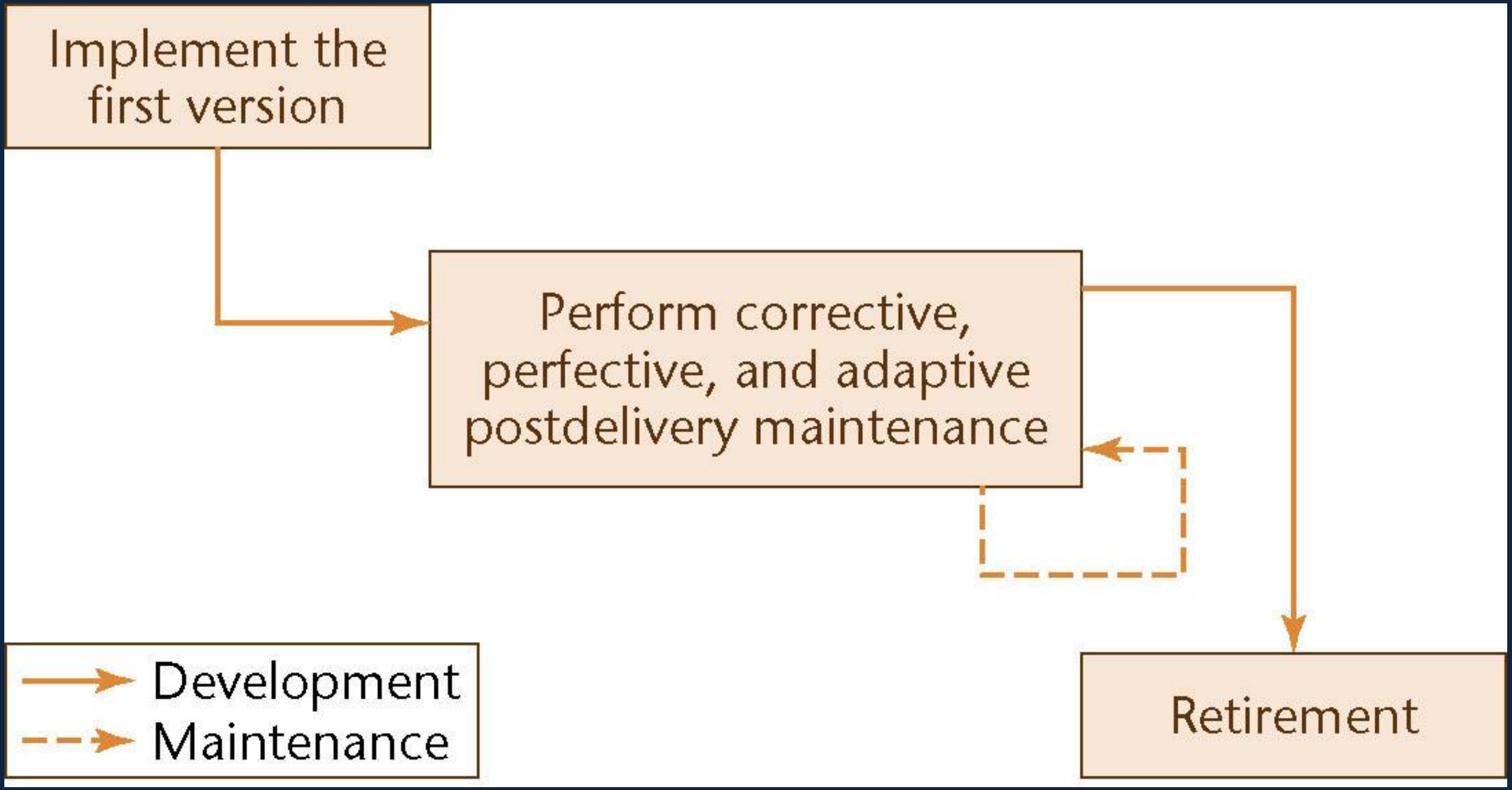
- Two informal phases
- First, one individual builds an initial version
  - Made available via the Internet (e.g., SourceForge.net)
- Then, if there is sufficient interest in the project
  - The initial version is widely downloaded
  - Users become co-developers
  - The product is extended
- Key point: Individuals generally work voluntarily (unpaid) on an open-source project in their spare time

The Activities of the Second Informal Phase

- Reporting and correcting defects
  - Corrective maintenance
- Adding additional functionality
  - Perfective maintenance
- Porting the program to a new environment
  - Adaptive maintenance
- The second informal phase consists *solely* of post-delivery maintenance
  - The word "co-developers" on the previous slide should rather be "co-maintainers"

Open-Source Life-Cycle Model

- Post-delivery maintenance life cycle model



Implement the first version

Perform corrective, perfective, and adaptive postdelivery maintenance

Retirement

Something in Common

- An initial working version is produced when using
  - The rapid-prototyping model;
  - The code-and-fix model; and
  - The open-source life-cycle model
- Then:
  - In Rapid-prototyping model
    - The initial version is discarded
  - In Code-and-fix model and open-source life-cycle model
    - The initial version becomes the target product

Open-Source Life-Cycle Model

- Consequently, in an open-source project, there are generally no specifications and no design
- How have some open-source projects been so successful without specifications or designs?

Open-Source Life-Cycle Model

- Closed-source software is maintained and tested by employees
- Users can submit failure reports = (observed incorrect behavior) but **never** fault reports = (incorrect source code and how to correct it) because the source code is not available
- Open-source software is generally maintained by unpaid volunteers
  - Users are strongly encouraged to submit defect reports, both failure reports and fault reports ➔ advantage of open source

Two groups Open-Source project

- Core group
  - = Small number of dedicated maintainers with the inclination, the time, and the necessary skills to submit fault reports (“fixes”)
  - They take responsibility for managing the project
  - They have the authority to install fixes
- Peripheral group
  - Users who choose to submit defect reports from time to time

Open-Source Life-Cycle Model

- New versions of **closed-source** software are typically released roughly once a year
  - After careful testing by the SQA group
- The core group releases a new version of an **open-source** product as soon as it is ready
  - Perhaps a month or even a day after the previous version was released
  - The core group performs minimal testing
  - Extensive testing is performed by the members of the peripheral group in the course of utilizing the software
  - “Release early and often”

Open-Source Life-Cycle Model

- Open-source software production has attracted some of the world's finest software experts
  - They can function effectively without specifications or designs
- However, eventually a point will be reached when the open-source product is no longer maintainable
- The open-source life-cycle model is restricted in its applicability (not all projects successful)

Open-Source Life-Cycle Model

- There cannot be open-source development of a software product to be used in just one commercial organization
- Key point: Members of both the core group and the periphery are users of the software being developed
- The open-source life-cycle model is inapplicable unless the target product is viewed by a wide range of users as useful to them

Open-Source Life-Cycle Model

- User’s Power:
  - To be successful, the project must be worthwhile (to be used by many users)
  - Users are also developers
  - Developers view it as a learning to gain skill & experience (to gain better position later)
- Superb developers:
  - Can propose a winning project
  - Can motivate others

Open-Source Life-Cycle Model

- About half of the open-source projects on the Web have not attracted a team to work on the project
- Even where work has started, the overwhelming predominance will never be completed
- But when the open-source model has worked, it has sometimes been incredibly successful
  - The open-source products previously listed have been utilized on a regular basis by millions of users

Synchronize-and Stabilize Model

- Microsoft’s life-cycle model based on iterative & incremental model
- Commercial software model: loose contact with users or customers
- Used in large products
  - Windows 2000: 30 million LOC, 3000 programmers, reusing Windows NT 4.0

Synchronize-and Stabilize Model

- Requirements analysis
  - Interview potential customers
  - Extract list of features of highest priority to the clients
- Draw up specifications
- Divide project into 3 or 4 builds
  - First build consist of most critical features, second build for next most critical features, and so on
- Each build is carried out by small teams working in parallel

Synchronize-and Stabilize Model

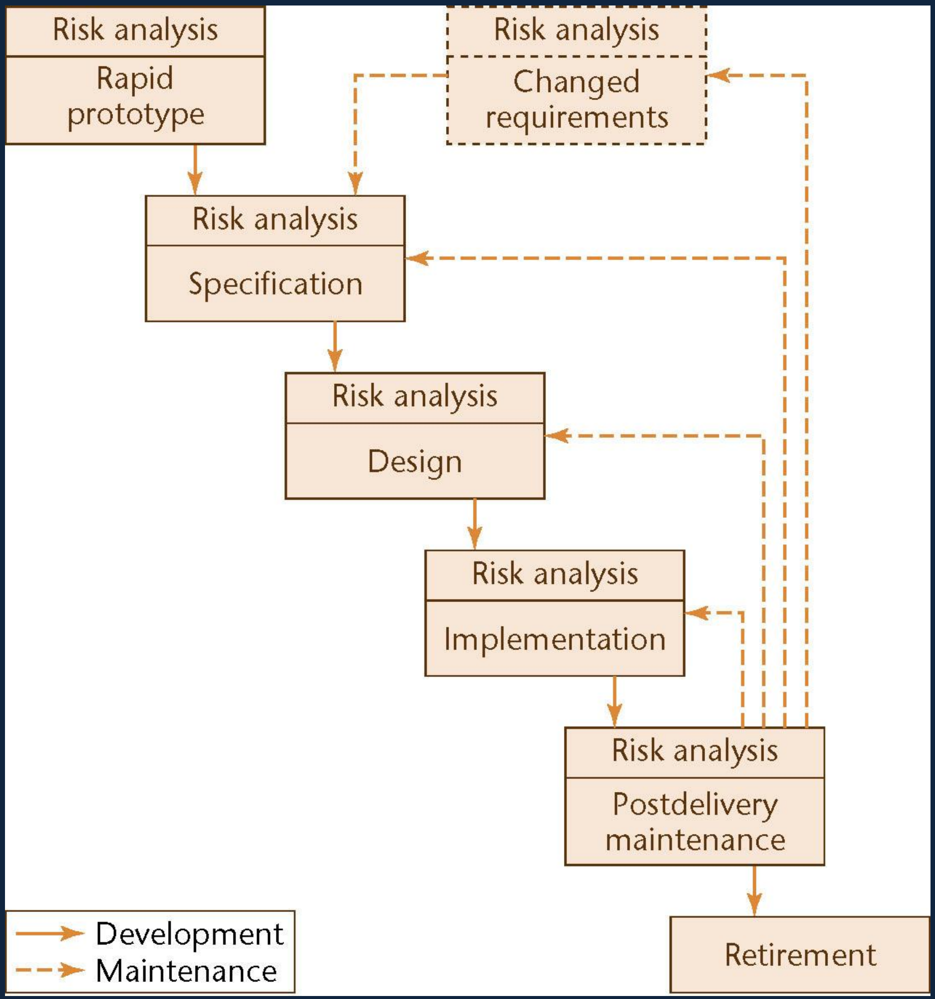
- At the end of each day — all teams *synchronize* (integrate component, test and debug)
- At the end of the build — *stabilize* (freeze the build and fixing bugs, no more change to specification)

Synchronize-and Stabilize Model

- Thus, components always work together
  - Get early insights into the operation of the product
  - Requirements can be modified during the course of a build
  - No integration problem
- Requires discipline to fix code immediately so that the rest of team can test and debug
- Needs highly talented managers and developers

Spiral

- Proposed by Boehm, 1988
- Simplified form
- Rapid prototyping model plus risk analysis preceding each phase



Risk in Software Development

- Key personnel resign before product finish
- Hardware manufacturer becomes bankrupt and software is critically dependent on that
- Before product is marketed, competitors announce first with equivalent functionality and cheaper price
- Components does not fit during integrations
- Thus, we want to minimize risk

Possible Risks

- People
  - Team not experienced
  - Team not familiar with technology
  - Unable to hire people with the right background
- Technology
  - Dependence on technology that changes
- Corporate
  - Too fast company growth
- Users
  - Lack of user acceptance
- Resources
  - Too short schedule
  - Too many users
  - Suppliers can’t deliver the product we depend on
- Market
  - Competitors
  - Not fast enough to market
  - Too fast to market
  - Market trend

Minimize some risks

- Prototype can minimize risk that delivered product does not satisfy client’s need
- Testing on Simulator or Proof-of-Concept prototype (not on actual system) reduce risk of problem if implemented on actual system
- Risk on team (resign or incompetence) are not addressed by prototype

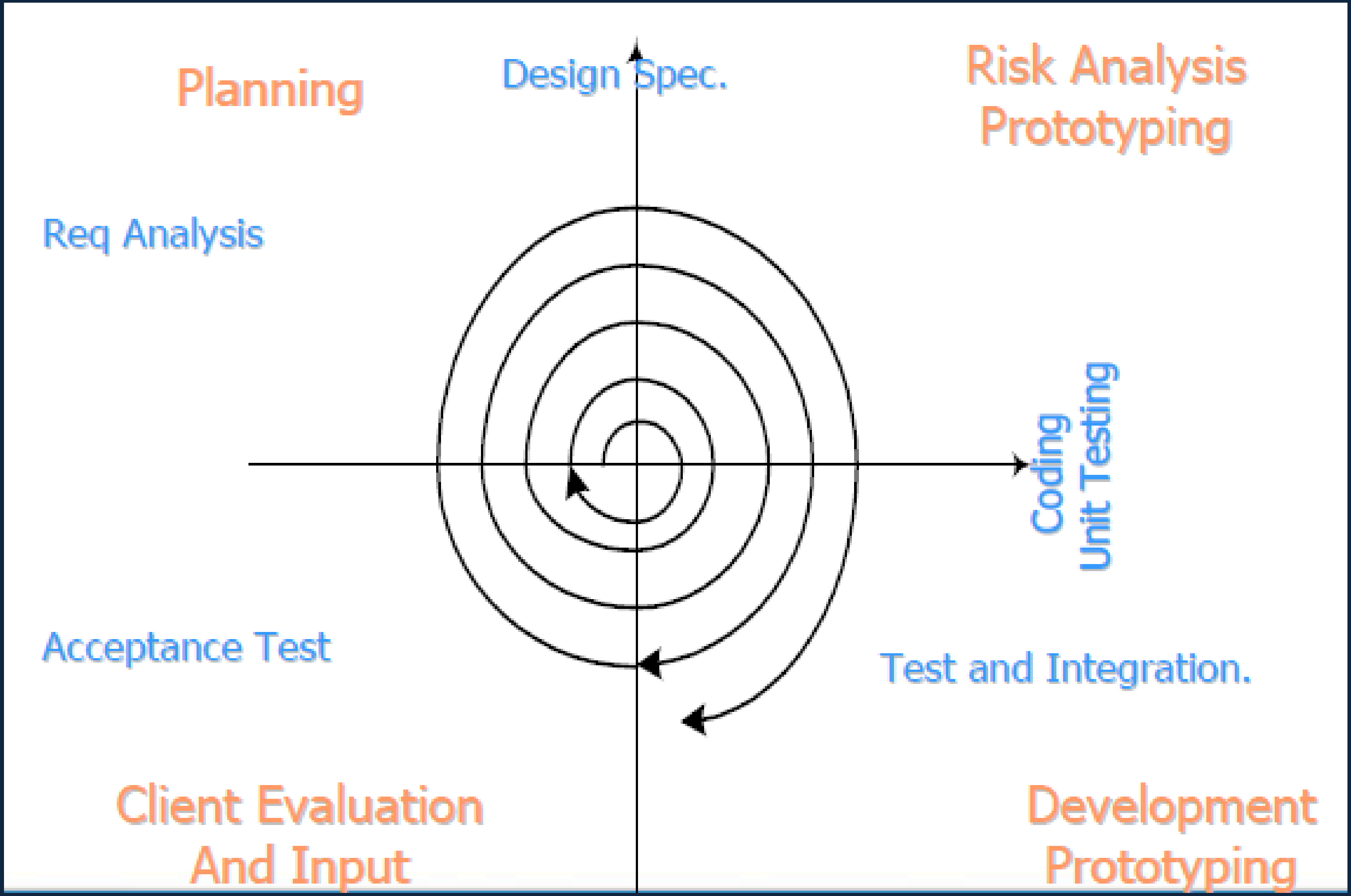
A Key Point of the Spiral Model

- It is rapid prototype & waterfall model with each phase preceded by risk analysis
- Identify the sub-problem which has the highest associated risk and find solution for that problem
- If all risks cannot be mitigated, the project is immediately terminated

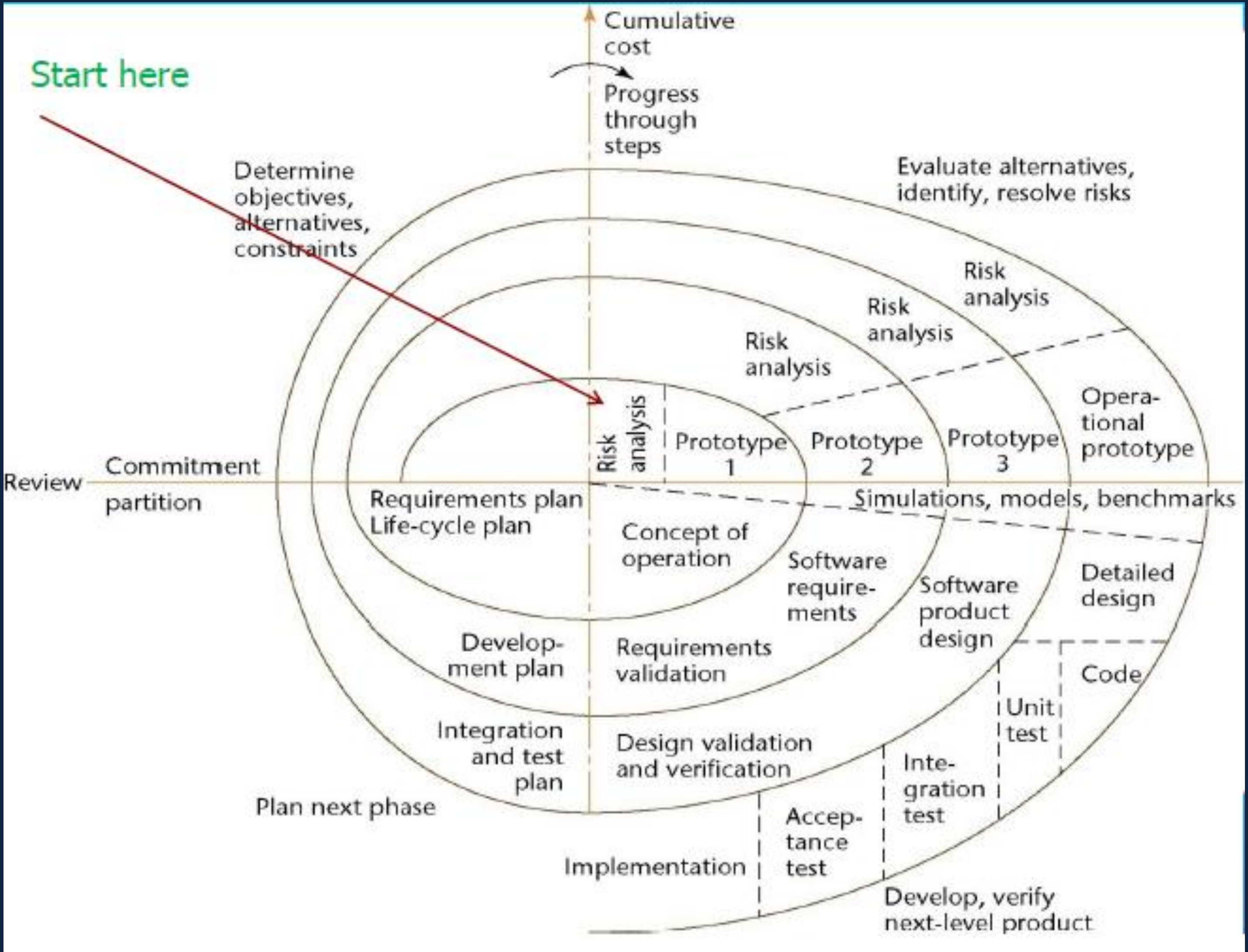
Full Spiral Model

- Incremental, track the spiral many times, once for each increment
- Precede each phase by
  - Alternatives
  - Risk analysis
- Follow each phase by
  - Evaluation
  - Planning of the next phase

Full Spiral Model



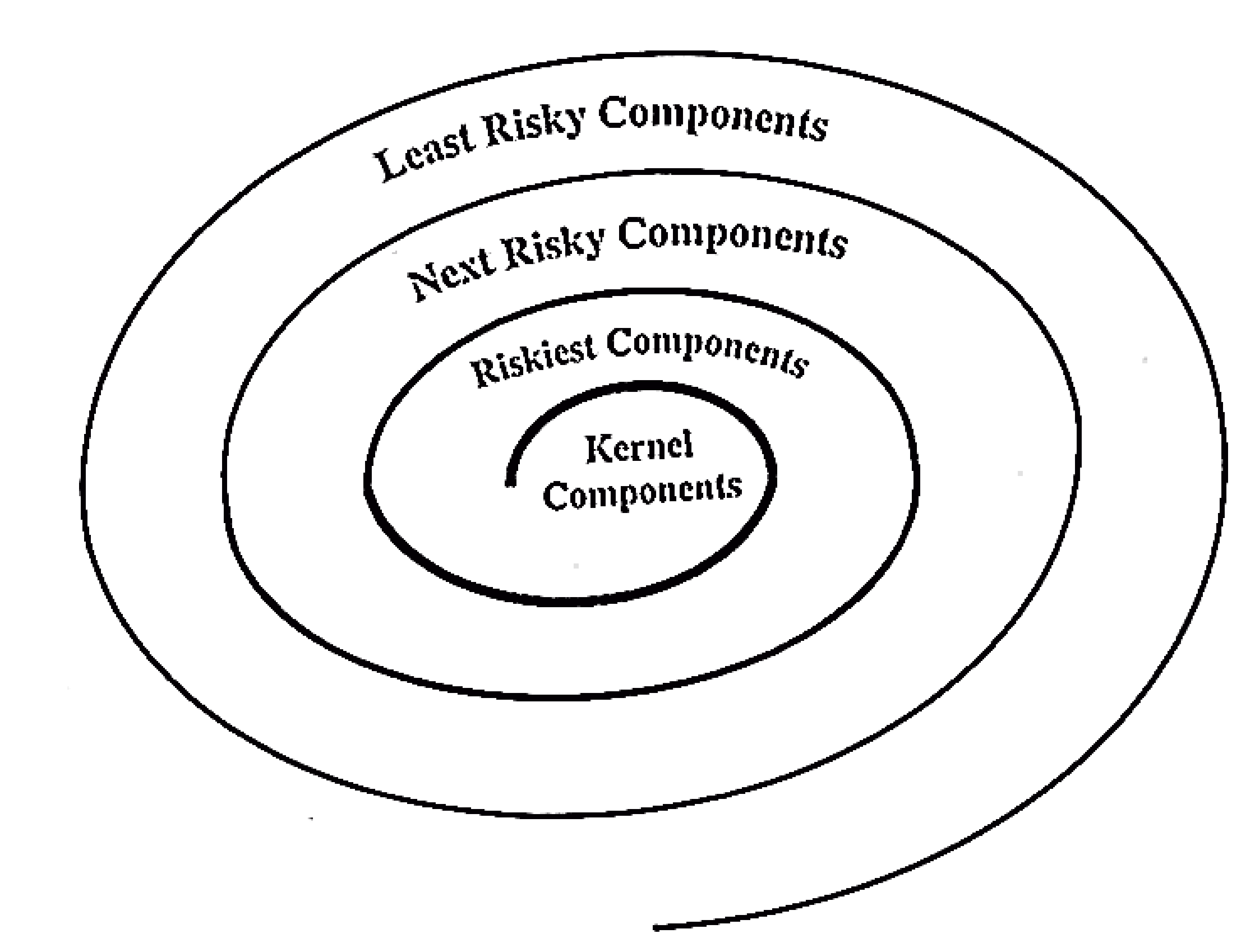
Full Spiral Model



Full Spiral Model

- Radial dimension: cumulative cost to date
- Angular dimension: progress through the spiral
- During maintenance, the reported error or changing requirement are triggers to track the spiral

Full Spiral Model



Analysis of the Spiral Model

- Strengths
  - It is easy to judge how much to test
  - No distinction is made between development and maintenance
  - Report increase 50%-100% productivity
- Weaknesses
  - For large-scale software only (cost of performing risk analysis could be higher than a small project cost)
  - For internal (in-house) software only
  - If project is terminated early, who pays?

Comparison of Life-Cycle Models

- Different life-cycle models have been presented
- Each with its own strengths and weaknesses
- Criteria for deciding on a model include:
  - The organization
  - Its management
  - The skills of the employees
  - The nature of the product
- Best suggestion
  - “Mix-and-match” life-cycle model

References