FRESNO STATE
Discovery. Diversity. Distinction.

CSCI 150
Intro to Software Engineering

September 20, 2018

Shih-Hsi “Alex” Liu

FRESNO STATE
Discovery. Diversity. Distinction.

Chapter 2 – Software Processes

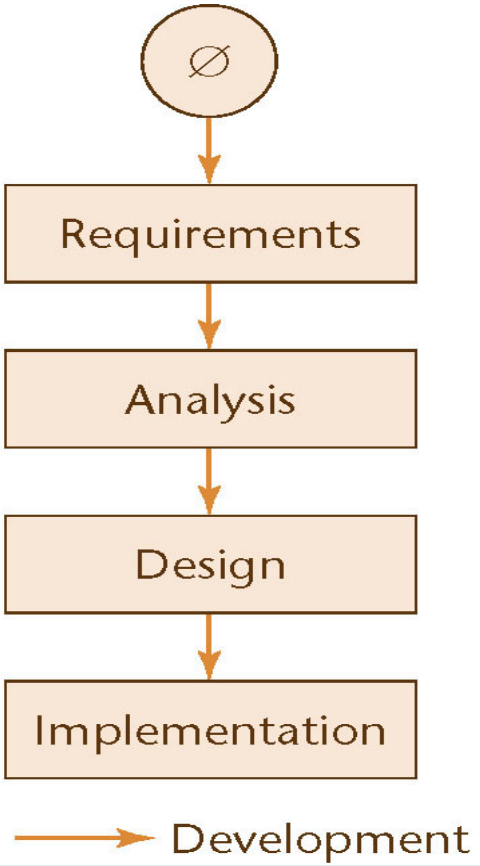
Lecture 1

Topics covered

- Software process models
- Process activities
- Coping with change
- The Rational Unified Process
  - An example of a modern software process.

Software Development in Theory

- Ideally, software is developed as described in Chapter 1
  - Linear
  - Starting from scratch



Software Development in Practice

- In the real world, software development is totally different
  - We make mistakes
  - The client's requirements change while the software product is being developed
  - Software may be built by extending and modifying existing systems or by configuring and integrating off-the-shelf software or system components.

The software process

- A structured set of activities required to develop a software system.
- Many different software processes but all involve:
  - Specification – defining what the system should do;
  - Design and implementation – defining the organization of the system and implementing the system (how);
  - Validation – checking that it does what the customer wants;
  - Evolution – changing the system in response to changing customer needs.
- Sub-activities:
  - Req. validation, architectural design, unit testing, documentation, configuration management, among others.

Software process descriptions

- When we describe and discuss processes, we usually talk about the activities in these processes such as specifying a data model, designing a user interface, etc. and the ordering of these activities.
- Process descriptions may also include:
  - Products/artifacts, which are the outcomes of a process activity;
    - SW architecture is an outcome of architectural design
  - Roles, which reflect the responsibilities of the people involved in the process;
    - Project Management, Configuration Management, programmers, Quality Engineering etc.
  - Pre- and post-conditions, which are statements that are true before and after a process activity has been enacted or a product produced.
    - Some clients request all docs being reviewed and approved before design

Choosing Software Processes

- Software processes are complex and are like intellectual and creative processes.
- No ideal process and most org. have their own processes
  - Depending on people, tools, culture, tech. etc.
  - There are no right or wrong software processes.
- E.g., Critical systems requires structured process
- E.g., Business systems requires *less formal and flexible* process due to its natural of rapid requirements change

Software process classifications: Plan-driven and agile processes

- Plan-driven processes are processes where all of the process activities are **planned in advance** and progress is measured against this plan.
- In agile processes, planning is **incremental** and it is easier to change the process to **reflect changing** customer requirements.
- In practice, most practical processes include elements of both plan-driven and agile approaches.

Improvement of Software Process

- Although there is no ‘ideal’ software process, there is scope for improving the software process in many organizations (e.g., using CMMI proposed by CMU). Processes may include outdated techniques or may not take advantages of the best practice in industrial software engineering.
- Software processes may be improved by process standardization where the diversity in software processes across an organization is reduced. This leads to improved communication and a reduction in training time and makes automated process support more economical.

Software process models

- A software process model is an abstract representation of a process. It presents a description of a process from some particular perspective.
- Each model represents a process from a particular perspective (in this chapter, we use "architectural perspective (or framework)", and thus provides only partial info about the process.
  - For example, a process activity model may not show the roles of people

Timeout

- I don’t like “A software process model is an abstract representation of a process. It presents a description of a process from some particular perspective.”
- I would prefer to use analogy of OOP concept:
  - Software process is a class and a software process model is an object of that class that comprises specific “properties” (orders of activities, and the way to perform activities and artifacts resulted from activities).

Software process models (simplified rep. of a SW process)

- The waterfall model
  - Plan-driven model. Separate and distinct phases of specification, design, implementation, testing, etc.
- Incremental development
  - Specification, development and validation are interleaved. Systems are developed as a series of increments. *May be plan-driven or agile*.
- Reuse-oriented software engineering
  - The system is assembled/integrated from existing components. May be *plan-driven or agile*.

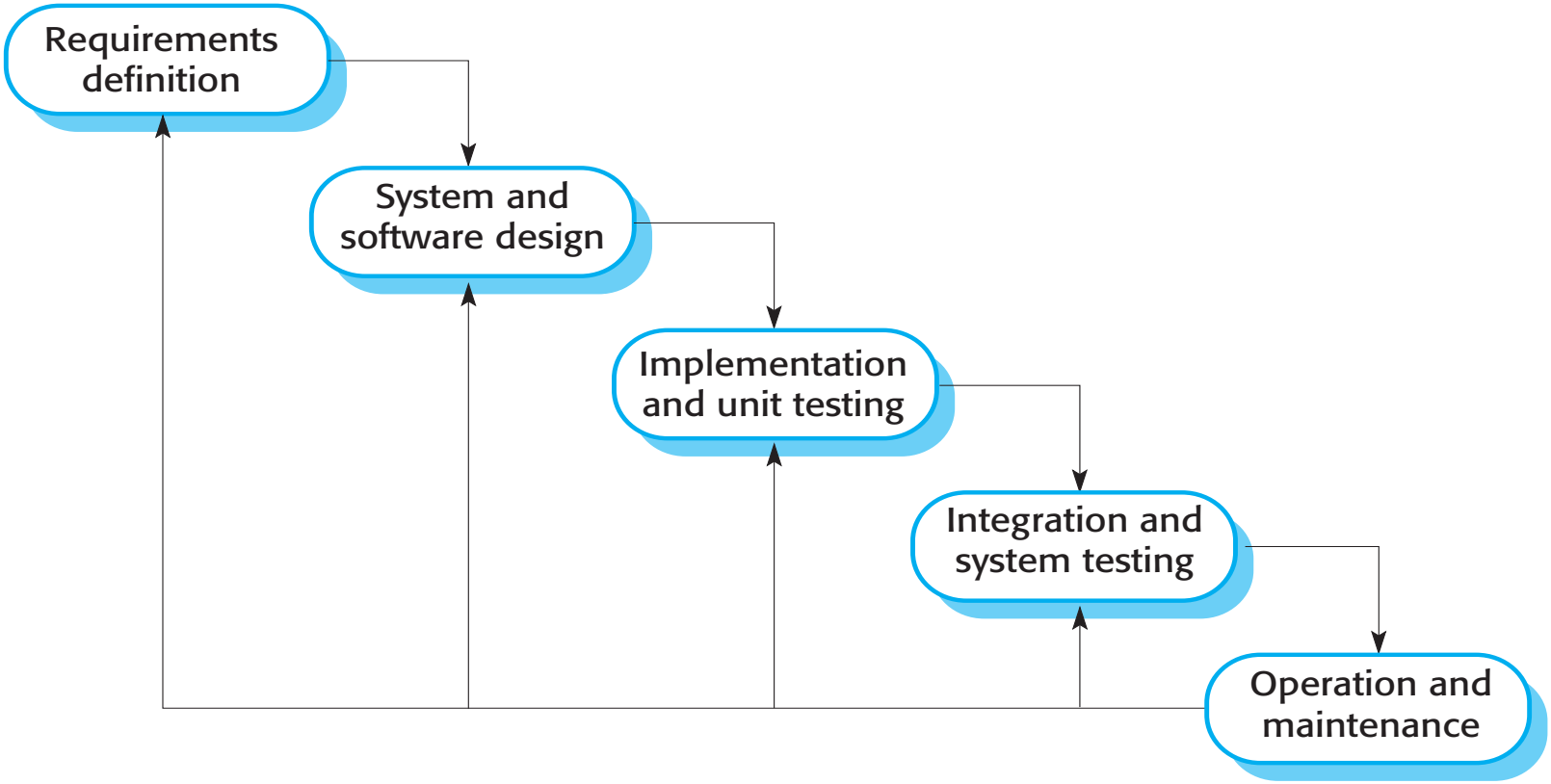
Software process models

- These models are not mutually exclusive and are often used together.
  - Some parts are **well understood** that can use waterfall
  - Others may be difficult to specify (e.g., UI) that need to develop incrementally
- In practice, most large systems are developed using a process that incorporates elements from all of these models.

1st SPML Waterfall model phases (est. 1970)

- There are separate identified phases in the waterfall model:
  - Requirements analysis and definition
    - Services, constraints, goals are defined in details and serve as specification.
  - System and software design
    - Software and System architectures are determined.
      - Allocate spec./req. to either hardware or software systems by establishing an overall system architecture.
      - Software design: Identify and describe the fundamental SW system abstractions and their relationships.
  - Implementation and unit testing
    - Programs that meet its spec.
  - Integration and system testing
  - Operation and maintenance
    - Corrective, Adaptive and Perfective maintenance

Figure 2.1 The waterfall model



- The linear life cycle model with feedback loops

  A fault is found in design phase and violates the req. => **backtrack up to top, correct and then go down**

Waterfall model characteristics

- Documentation-driven
  - Next phase is OK to proceed only if the doc of the current phase is signed off
  - In reality - non-linear model: Feedback from next phase triggers changes of previous phase. Doc is changed accordingly
- Inflexibility: Req. or design may be frozen (prematurely) to reduce cost of producing and approving documentation and cause the following inflexibilities:
  - Not accommodate to client's need and may badly structured (problems are left for later resolution, ignored, or programmed around)
  - Later changes required by clients or feedbacks may not be reflected
  - Difficulty of accommodating change after the process is underway. In principle, a phase has to be completed before moving onto the next phase.

Waterfall model problems

- Inflexible partitioning of the project into distinct stages. Commitments must be made at an early stage in the process, which makes it difficult to respond to changing customer requirements.
  - Therefore, this model is only appropriate when the requirements are well-understood and **changes will be fairly limited** during the design process.
  - Few business systems have stable requirements.
- The waterfall model is mostly used for large systems engineering projects where a system is developed at several sites.
  - In those circumstances, the plan-driven nature of the waterfall model helps **coordinate** the work.

Waterfall model characteristics

- In summary:
  - Doc driven makes the process **visible** so managers can **monitor progress** against the development plan.
  - Commitment must be made at an early stage, which makes it difficult to respond to changing customer requirements.
  - Waterfall should be used when requirements are well understood and unlikely to change radically during system development.
  - Good for formal system development (next slide)
- Good summary:
  - https://airbrake.io/blog/sdlc/waterfall-model

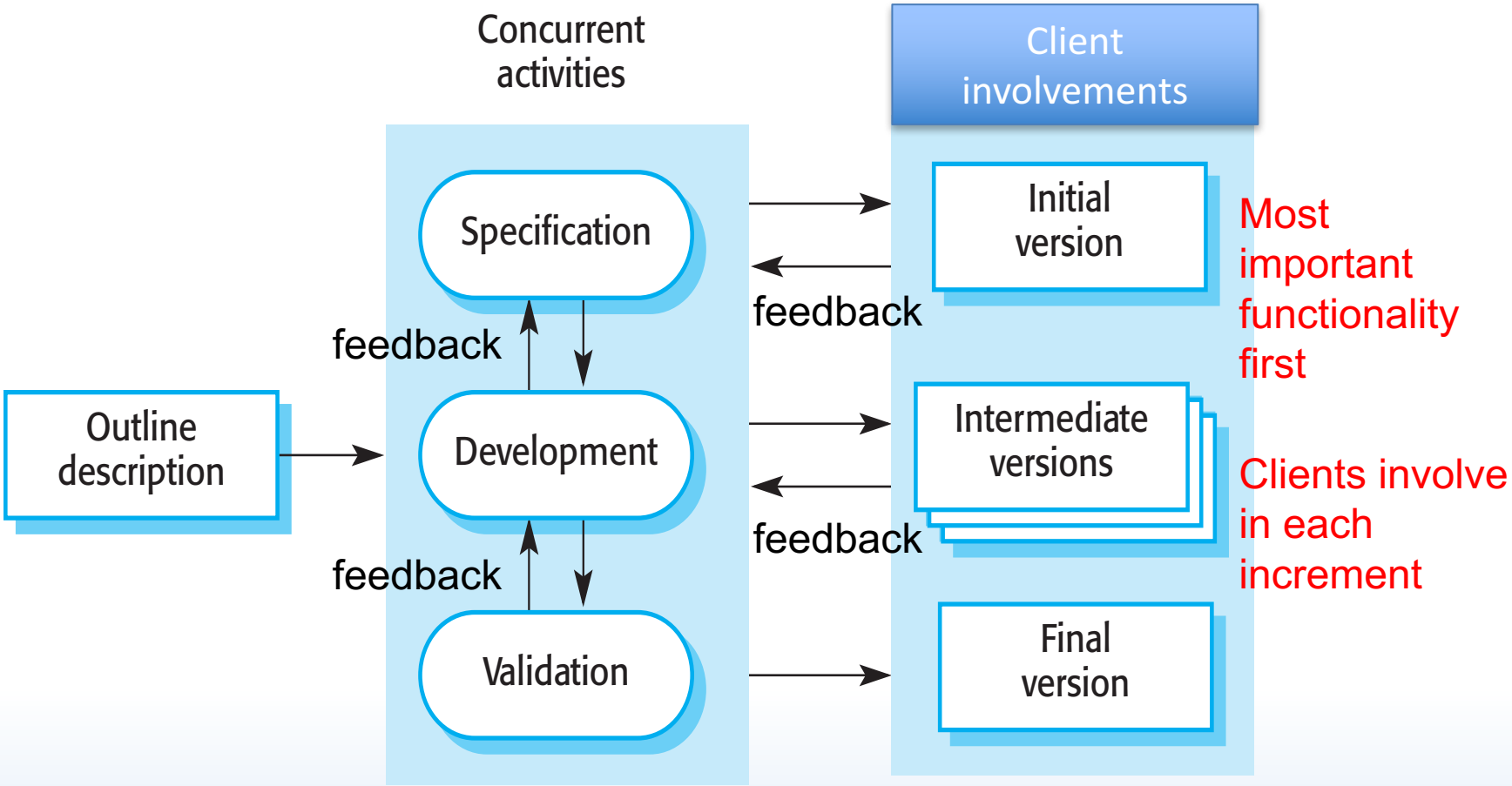
Formal Development Process

- Variation of Waterfall model
- Mathematics as spec. (formal specification)
- Refine and transform into executable
- E.g., B, Z, VDM, CSP
- Good for systems that require safety, reliability, security

2nd SPM: Incremental development

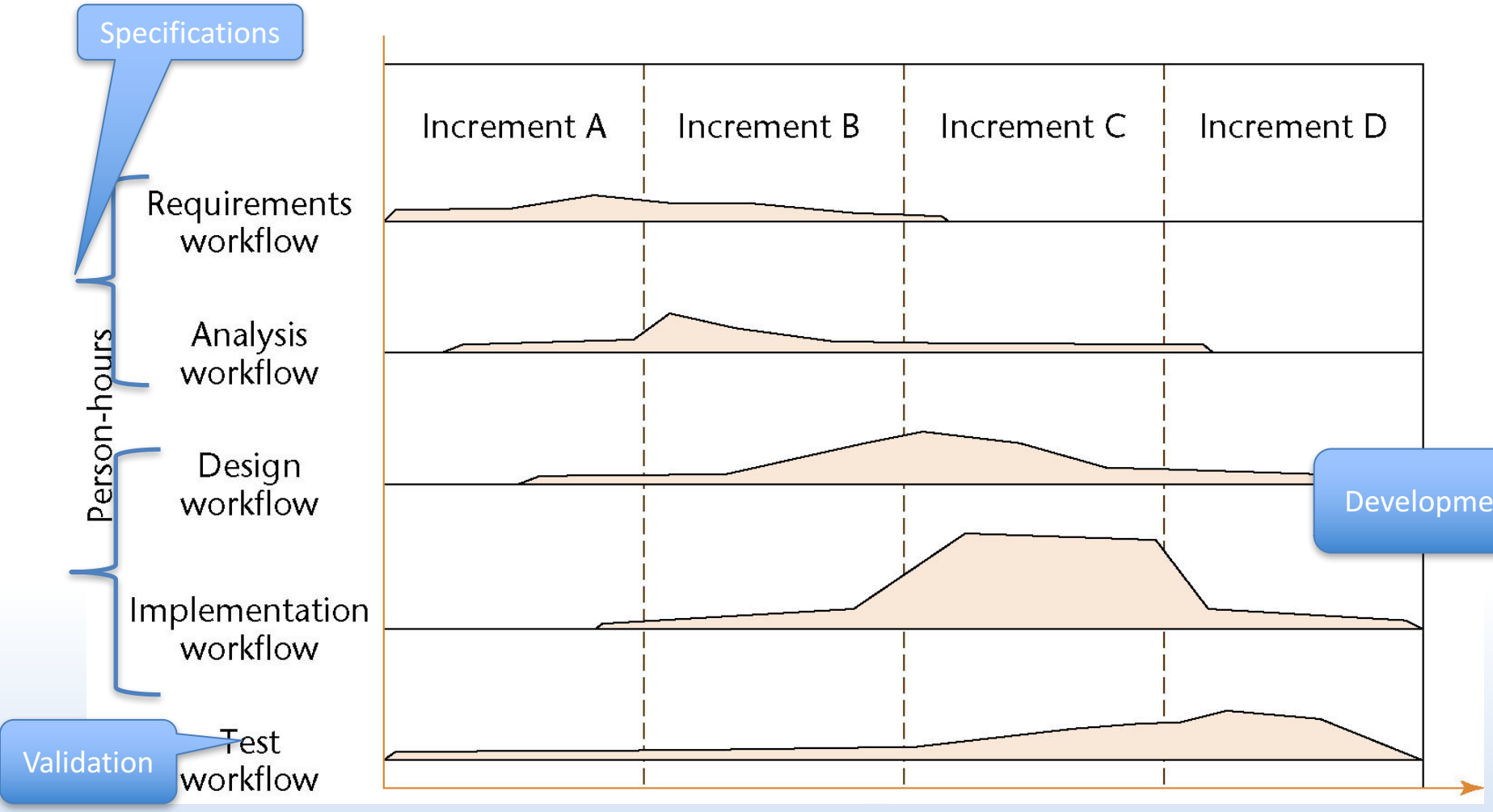
- Develop initial version and request feedbacks from clients. Evolve (increment and/or backtrack) the software (based on iterations of feedback) until done
- Spec, development, validation activities are interleaved w/ rapid feedbacks.
- Better for changing environments: business, e-commerce
- Can be plan-driven, agile, or mix.
- Closer to real-world situations.
  - We rarely work out a complete program solution in advance but move toward a solution in a series of steps, backtracking when we realize that we have made a mistake.
  - Cheaper and easier

Figure 2.2 Incremental development



Backtracking is possible in this model.

Incremental development



Strengths of the Incremental Model

- The cost of accommodating changing customer requirements is reduced.
  - The amount of analysis and documentation that has to be redone is much less than is required with the waterfall model.
- It is easier to get customer feedback on the development work that has been done.
  - Customers can comment on demonstrations of the software and see how much has been implemented. (difficult for customers to judge progress from design documents).
- More rapid delivery and deployment of useful software to the customer is possible.
  - Customers are able to use and gain value from the software earlier than is possible with a waterfall process.

Strengths of the Incremental Model (contd.)

- There are multiple opportunities for checking that the software product is correct
  - Every iteration (or you can say version in Figure 2.2, or increment figure after Figure 2.2) incorporates the **test** workflow
  - Faults can be detected and corrected early (more important features can be tested more times)
- The robustness of the architecture can be determined early in the life cycle
  - ***Architecture*** — 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 Incremental Model (contd.)

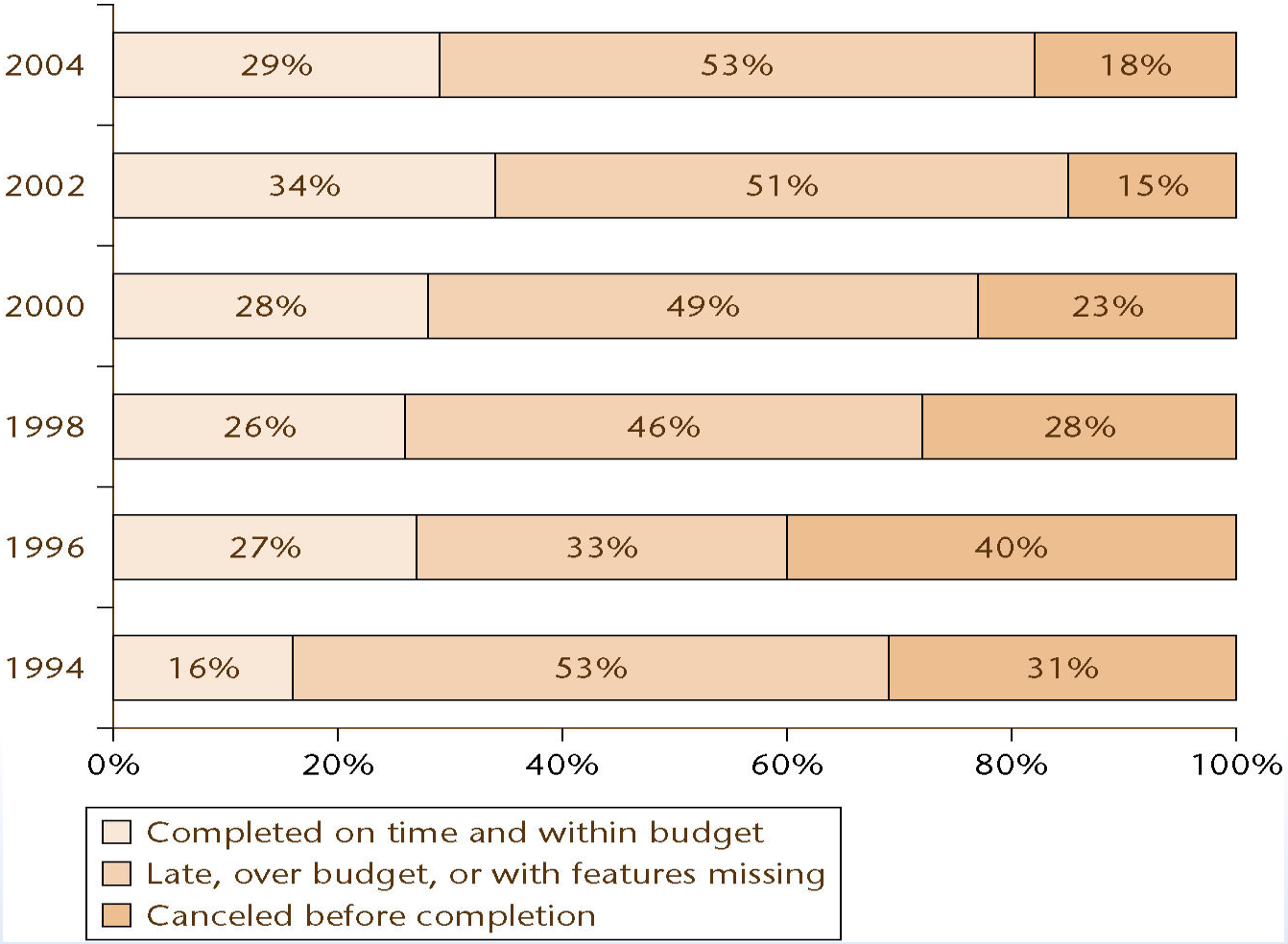
- We can *mitigate* (resolve) risks early
  - Risks are invariably involved in software development and maintenance
- We have a working version of the software product from the start
  - The client and users can experiment with this version to determine what changes are needed
- Variation: Deliver partial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

Strengths of the Incremental Model (contd.)

- There is empirical evidence that the incremental development model works
- The CHAOS reports of the Standish Group (see overleaf) show that the percentage of successful products increases

Strengths of the Incremental Model (contd)

- CHAOS reports from 1994 to 2004



Incremental development problems

- The process is not visible.
  - Managers need regular deliverables to measure progress. If systems are developed quickly, it is not cost-effective to produce documents that reflect every version of the system.
- System structure tends to degrade as new increments are added.
  - Unless time and money is spent on refactoring to improve the software, regular change tends to corrupt its structure. Incorporating further software changes becomes increasingly difficult and costly.

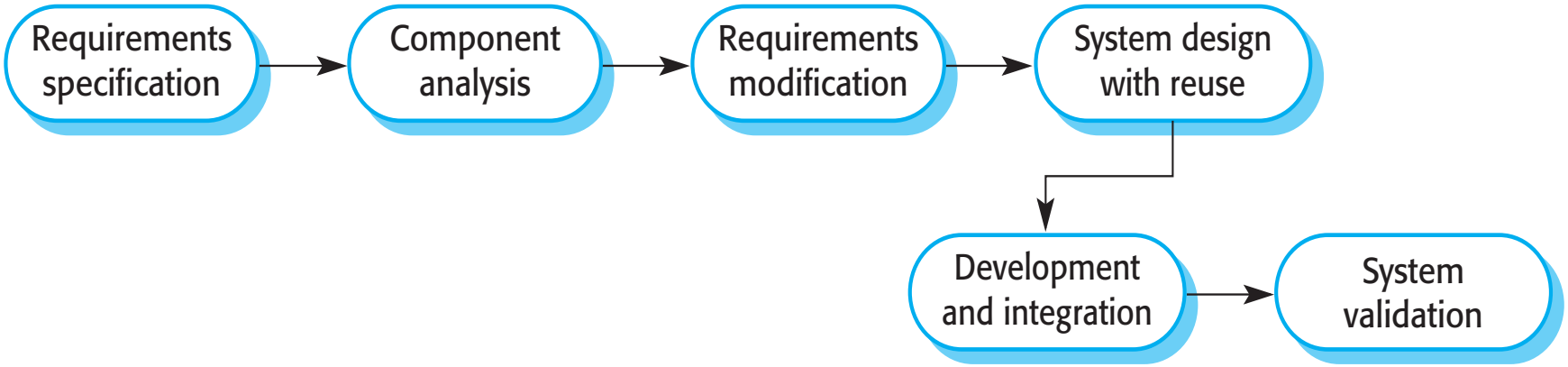
Incremental development problems

- The problems become particularly acute for large, complex, long-life systems, where different teams develop different parts of the system
- Solutions for large projects:
  - Stable framework or architecture and clear definitions of the responsibilities of different teams are needed
  - plan in advance rather than develop incrementally for large projects.

3rd SPM: Reuse-oriented software engineering

- Based on **systematic** reuse where systems are integrated from existing components or COTS (Commercial-off-the-shelf) systems.
- Process stages
  - Component analysis;
    - Search "appropriate" components
  - Requirements modification;
    - Modify req. if perfect-match components are not found
  - System design with reuse;
  - Development and integration.
    - Develop not-available/reusable components. Integrate all

Figure 2.3 Reuse-oriented software engineering



Types of software component

- Web services that are developed according to service standards and which are available for remote invocation (WSDL or REST based).
- Collections of objects that are developed as a package to be integrated with a component framework such as .NET or J2EE.
- Stand-alone software systems (COTS) that are configured for use in a particular environment.

Reuse-oriented software engineering

- ✧ Reuse is now the standard approach for building many types of business system
- ✧ Pros and Cons
  - ✧ Reduce time, cost and maybe risks
  - ✧ May not meet client's needs
  - ✧ Control over evolution is lost (e.g., some Yahoo web services are no longer available)
- Reuse covered in more depth in Chapter 16.

2.2 Process activities

- Real software processes are inter-leaved sequences of technical, collaborative and managerial activities with the overall goal of specifying, designing, implementing and testing a software system.
  - Also, tools are useful for supporting software processes
- The four basic process activities of specification, development, validation and evolution are organized differently in different development processes. In the waterfall model, they are organized in sequence, whereas in incremental development they are inter-leaved.
  - E.g., XP (eXtreme Programming)

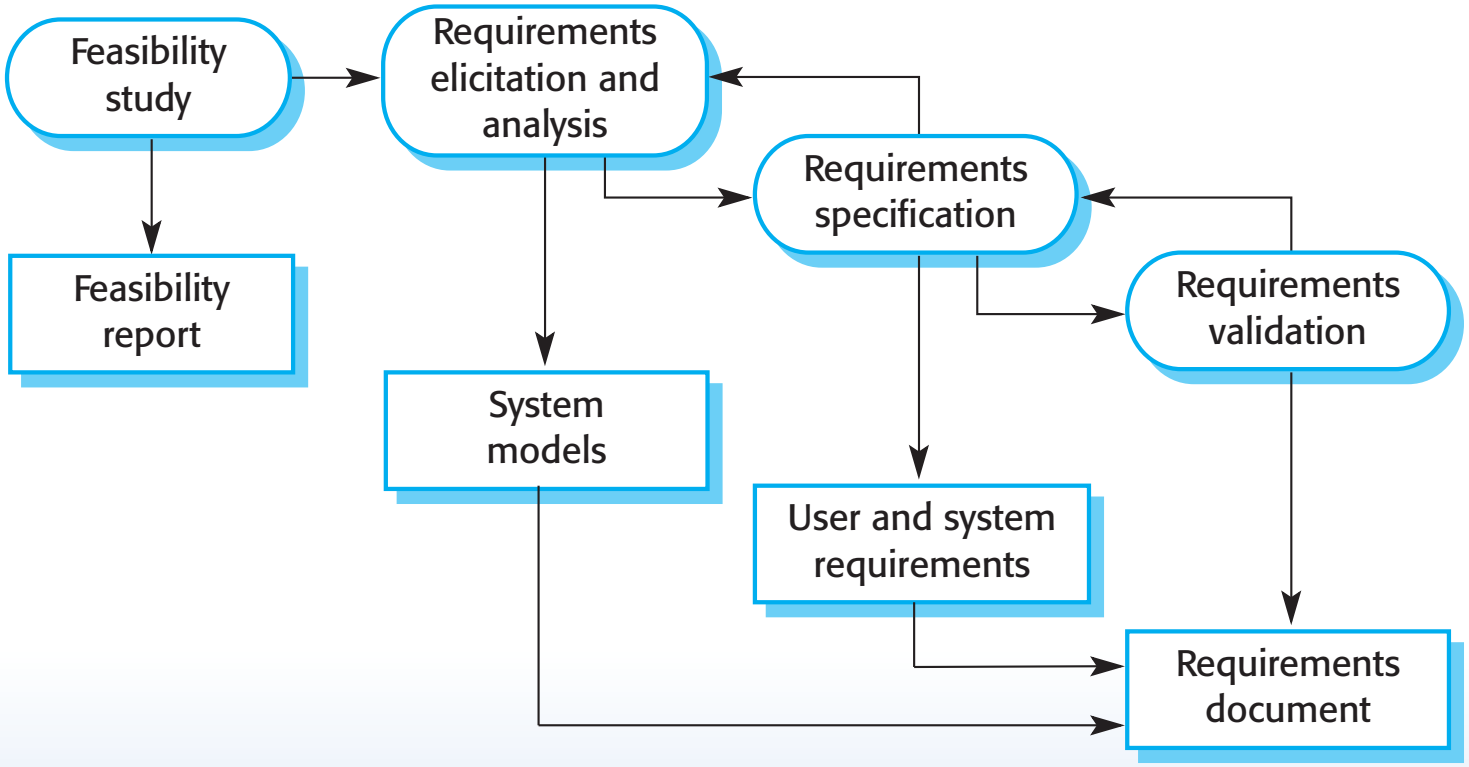
Software specification/Requirement eng.

- The process of establishing what services are required and the constraints on the system's operation and development.
  - Prior to this is understanding client's **domain** (needed before the 4 steps in next slide)
    - interview w/ clients
    - UML use case diagrams and descriptions
  - Specify a system satisfying stakeholder requirements
    - High level req. for end users and clients (User req.)
    - Detailed level req. for developers (System req.)
  - Most vital activity
    - Any error in this activity will be propagated to later phases

Software specification

- ✧ Requirements engineering process (4 steps)
  - Feasibility study
    - Is it technically (available) and financially feasible (cost-effective) to build the system?
    - http://www.businessinsider.com/10-old-apple-products-that-totally-failed-2013-11?op=1
  - Requirements elicitation and analysis
    - What do the system stakeholders require or expect from the system? (e.g., using Prototyping)
  - Requirements specification
    - Defining the requirements in details (user req. + system req.)
  - Requirements validation
    - Checking the validity of the requirements (realism, consistency, completeness, none-ambiguity)

Figure 2.4 The requirements engineering process



Precise, unambiguous, clear

- An example (from a real safety-critical system)

> The message must be triplicated. The three copies must be forwarded through three different physical channels. The receiver accepts the message on the basis of a two-out-of-three voting policy.

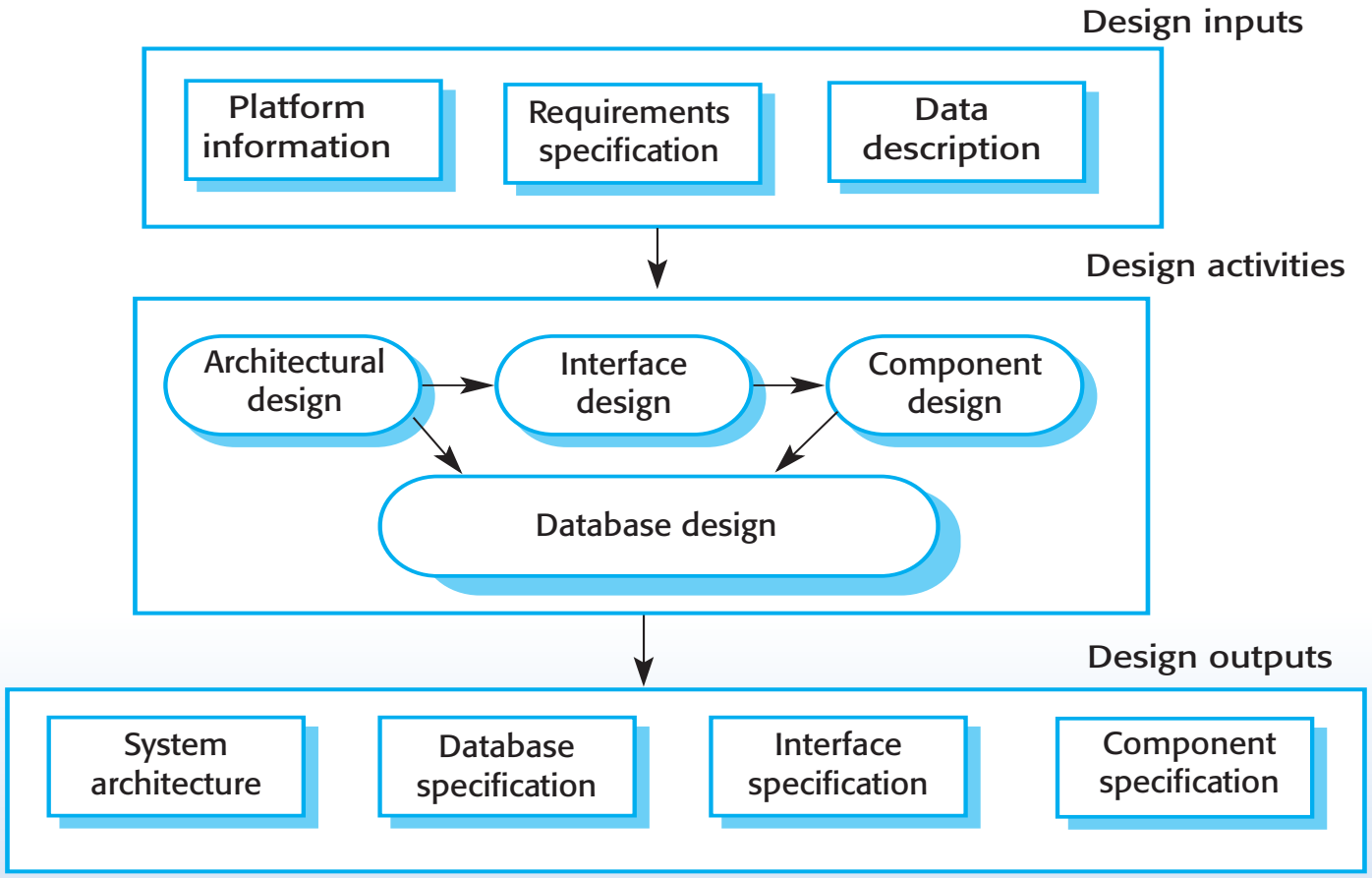
can a message be accepted as soon as we receive 2 out of 3 identical copies of message or do we need to wait for receipt of the 3rd?

CHAPTER 2
PART 2

Software design and implementation

- The process of converting the system specification into an executable system.
- Software design
  - Design a software structure that realises the specification, data models and structures used by the system, the interfaces between systems components, and algorithms used;
- Implementation
  - Translate this structure into an executable program;
- The activities of design and implementation are closely related and may be inter-leaved.

Figure 2.5 A general model of the design process



Input of Design Activities

- Software platform is the environment in which the software will execute (e.g., OS, DB, middleware, other application systems etc.)
  - Design must know how to integrate software with its environment
- Requirements Specification is a description of the functionality the software must provide and its performance and dependability requirements
- Data description defines system's data organization

Design activities

- *Architectural design,* where you identify the overall structure of the system, the principal components (sometimes called sub-systems or modules), their relationships and how they are distributed.
- *Interface design,* where you define the interfaces between system components.
  - Note this is **not user interface design**
- *Component design,* where you take each system component and design **how** it will operate.
- *Database design,* where you design the system data structures and how these are to be represented in a database.

Output of Design Activities

- Format and level of details may vary
  - Critical systems must be precise and accurate
  - Model-driven systems may be diagrams
  - Agile methods may be **code** of the program
- Programming styles vary, too
  - Some start with understandable components; others prefer difficult components first.
  - **Some start with data and then others; others leave data as the last one to implement***
  - Testing vs. Debugging
    - Testing establishes the existence of defects. Debugging is concerned with locating and correcting those defects.

*Top-down vs. Bottom-up approach

- Top-down
  - Design: It starts from the view of the entire software system and then gradually break down (decompose, refine) the system design into sub-systems until down to finest-grained element of the system (stepwise refinement)

*Top-down vs. Bottom-up approach

- Top-down design
  - Pros:
    - major design flaws show early
    - Support fault isolation
  - Cons:
    - delays testing of the ultimate functional units (“down” pieces”) of a system until significant design (“top pieces”) is complete
  - Programming analog: start from main function and then gradually decompose an entire project from class level to function level.

*Top-down vs. Bottom-up design and programming

- Bottom-up
  - Design: “the piecing together of systems to give rise to more complex systems, thus making the original systems sub-systems of the emergent system”

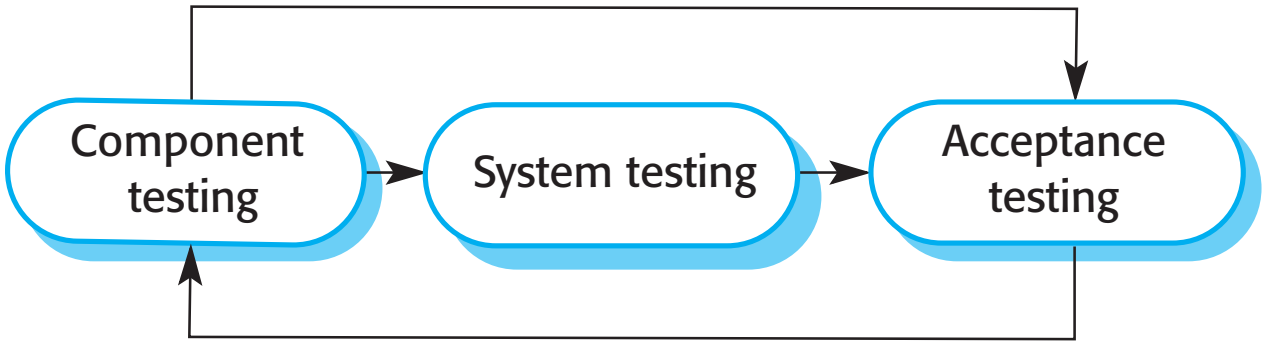
*Top-down vs. Bottom-up approach

- Bottom-up
  - Pros:
    - support fault isolation
    - Solve the con of top-down approach (functional units are not tested comprehensively)
  - Cons:
    - significant design (“up pieces”) may not be tested as completed as functional units (“bottom pieces”
    - When DB entities changes, changes may be propagated up.
  - Programming analog: start from classes and associated functions and then gradually **compose** an entire project from these classes.

Software validation

- Verification and validation (V & V) is intended to show that a system conforms to its specification and meets the requirements of the system customer.
- Involves checking and review processes and testing.
- Testing involves executing the system with test cases that are derived from the specification of the real data to be processed by the system.
- Testing is the most commonly used V & V activity.

Figure 2.6 Stages of testing



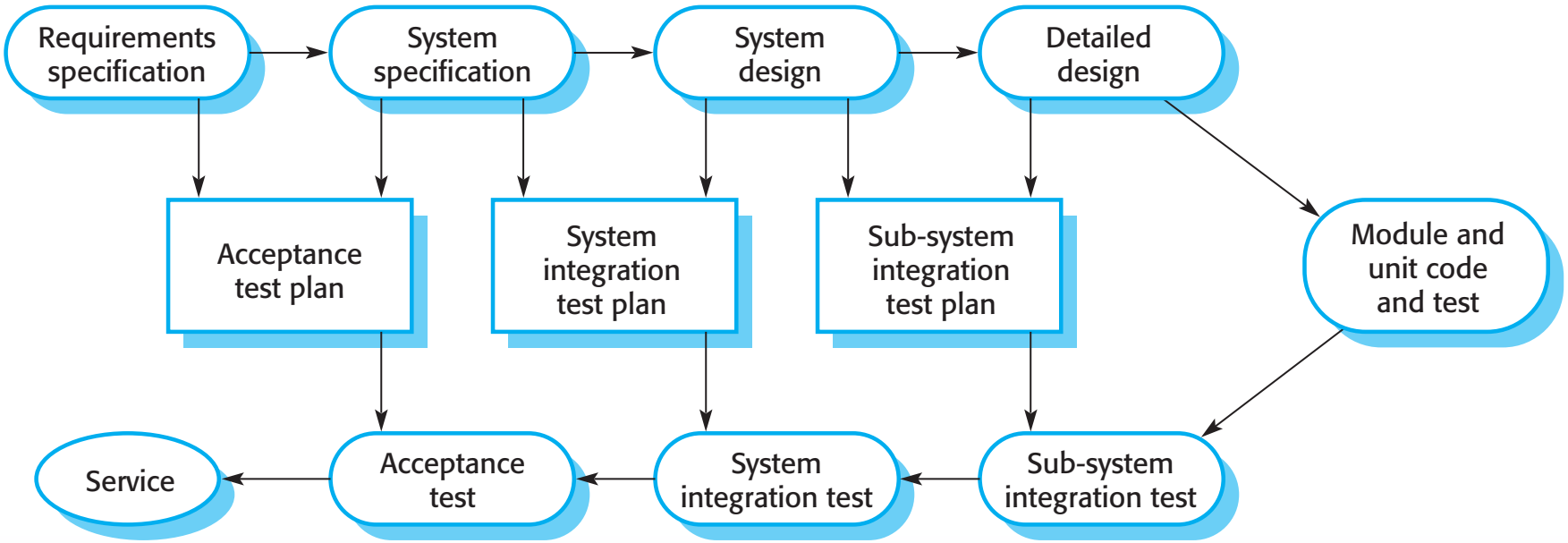
Testing stages

- ✧ Development or component testing (unit testing)
  - Individual components are tested *independently*;
  - Components may be functions or objects or coherent groupings of these entities.
  - **Don't forget integration testing**
- ✧ System testing (Product Testing)
  - Testing of the system as a whole. Testing of emergent properties is particularly important (functional and non-functional req.)
  - Artificial data is used for testing here
- ✧ Acceptance testing
  - Testing with customer data to check that the system meets the customer's needs.

Some Testing Characteristics

- Component development and Testing are interleaved
- eXtreme Programming is test driven (test is created along with req.).
  - This helps testers and developers to understand the requirements and ensures that there are no delays as test cases are created.
- Plan-driven: **V-model (next slide)**
- Alpha and Beta testing:
  - Acceptance testing also called alpha testing (single client)
  - Beta testing: software are tested by potential users

Figure 2.7 Testing phases in a plan-driven software process



Software evolution

- Software is inherently flexible and can change.
- As requirements change through changing business circumstances, the software that supports the business must also evolve and change.
- Although there has been a demarcation between development and evolution (maintenance) this is increasingly irrelevant as fewer and fewer systems are completely new.
  - Is maintenance is an easy task?
  - Should manager assigns newbies to maintenance team if it exists*
    - Note modern team organization no longer encourage an independent “maintenance team”

Key points

- Software processes are the activities involved in producing a software system. Software process models are abstract representations of these processes.
- General process models describe the organization of software processes. Examples of these general models include the ‘waterfall’ model, incremental development, and reuse-oriented development.

Key points

- Requirements engineering is the process of developing a software specification.
- Design and implementation processes are concerned with transforming a requirements specification into an executable software system.
- Software validation is the process of checking that the system conforms to its specification and that it meets the real needs of the users of the system.
- Software evolution takes place when you change existing software systems to meet new requirements. The software must evolve to remain useful.

FRESNO STATE
Discovery. Diversity. Distinction.

Chapter 2 – Software Processes

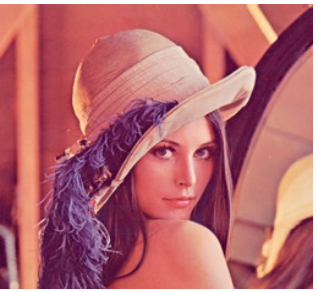
Lecture 2

Moving Target Problem

- A change in the requirements while the software product is *being developed* (the tractor case study)
- Even if the reasons for the change are *good*, the software product can be *adversely* impacted

  Dependencies will be induced

  - An image processing example
    - Add a filter function to eliminate noise of an image
  - In terms of functionality
    - The filtering algorithm might eliminate some useful information that may be required by other functions
  - In terms of quality
    - The filtering algorithm may increase the response time of the image processing system and violate the non-functional req.
  - www.lenna.org or http://www.cs.cmu.edu/~chuck/lennapg/lenna.shtml



Moving Target Problem (contd)

- 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

Moving Target Problem (contd)

- 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 perfect solution to the moving target problem

  Yet, when designing your project, make sure components/modules/classes are less independent as possible.

  Also, choose a suitable software process model

Coping with change

- Change is inevitable in all large software projects.
  - Business changes lead to new and changed system requirements (adaptive maintenance)
  - New technologies open up new possibilities for improving implementations (perfective maintenance)
  - Changing platforms require application changes (adaptive maintenance)
  - Diagnose and fix errors (corrective maintenance)
  - Whatever software process model is used, it is essential that it can accommodate changes to the SW being developed
- Change leads to rework so the costs of change include both rework (e.g. re-analyzing requirements) as well as the costs of implementing new functionality (or fix defects)

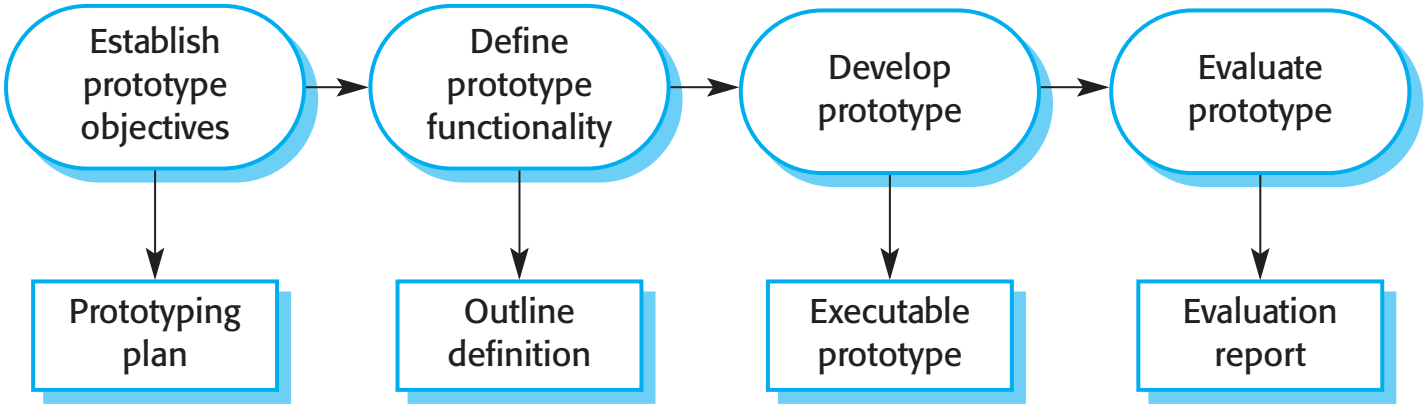
Reducing the costs of rework

- Change avoidance, where the software process includes activities that can **anticipate** possible changes **before** significant rework is required.
  - For example, a prototype system may be developed to show some key features of the system to customers.
- Change tolerance, where the **process** is designed so that changes can be accommodated at relatively low cost.
  - This normally involves some form of incremental development. Proposed changes may be implemented in increments that have not yet been developed. If this is impossible, then only a single increment (a small part of the system) may have be altered to incorporate the change.

1. Software prototyping

- A prototype is an initial version of a system used to demonstrate concepts and try out design options.
- A prototype can be used in:
  - The requirements engineering process to help with requirements elicitation and validation;
    - May help reveal errors and omissions
  - In design processes to explore design options, and develop a UI design;
    - E.g., DB prototype checks data access efficiency

Figure 2.9 The process of prototype development



Prototype development

- ✧ May be based on rapid prototyping languages or tools
- ✧ May involve leaving out functionality (Step 2 of the previous slide)
  - Prototype should focus on areas of the product that are not well-understood; don't attempt to introduce a prototype achieving too many objectives (Step 1)
  - Error checking and recovery may not be included in the prototype;
  - Focus on functional rather than non-functional requirements such as reliability and security

Prototype development

- Prototype evaluation should be against its objective.
  - Users should get trained to use prototype
  - Once getting comfortable, users may find requirements errors and omissions.

Throw-away prototypes

- Prototypes should be discarded after development as they are not a good basis for a production system:
  - It may be impossible to tune the system to meet non-functional requirements;
  - Prototypes are normally undocumented;
  - The prototype structure is usually degraded through rapid change;
  - The prototype probably will not meet normal organizational quality standards.

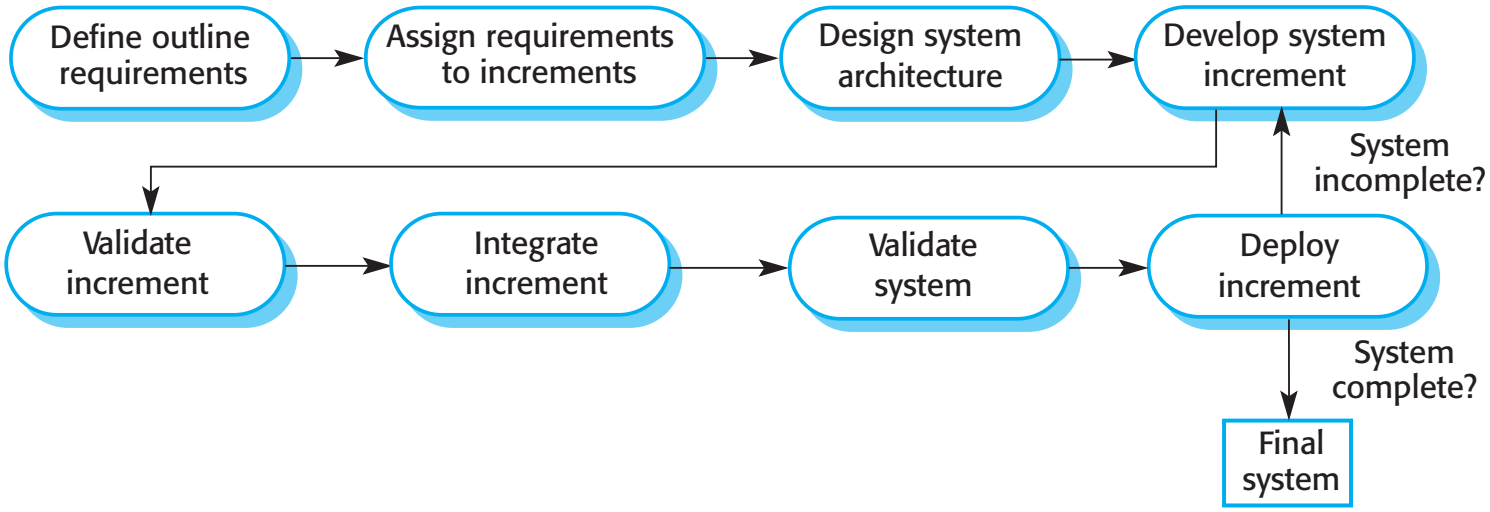
Summary: Benefits of prototyping

- Improved system usability.
- A closer match to users' real needs. (req. engineering)
- Improved design quality. (find weakness and strength)
  - E.g., database prototype for checking the performance of data access
- Improved maintainability. (reveal errors and omissions in the req.)
- Reduced development effort.

2. Incremental delivery

- Rather than deliver the system as a single delivery, the **development and delivery** is broken down into increments with each increment delivering part of the required functionality.
- User requirements are prioritised and the highest priority requirements are included in early increments.
- Once the development of an increment is started, the requirements of **current** increment are **frozen,** though requirements for later increments can continue to evolve.

Figure 2.10 Incremental delivery



Incremental development and delivery

- Incremental development
  - Develop the system in increments and evaluate each increment before proceeding to the development of the next increment;
  - Normal approach used in agile methods;
  - Evaluation done by user/customer proxy.
- Incremental delivery
  - **Deploy** an increment for use by end-users;
  - More realistic evaluation about practical use of software;
  - Difficult to implement for replacement systems as increments have less functionality than the system being replaced.

Incremental delivery advantages

- Early increments act as a prototype (but still not prototype) to help elicit requirements for later increments.
- Customers do not have to wait until the entire system is delivered before they can gain value from it.
- The process maintains the benefits of incremental development in that it should be relatively easy to incorporate changes into the system.
- **The highest priority system services tend to receive the most testing. Lower risk of overall project failure.**

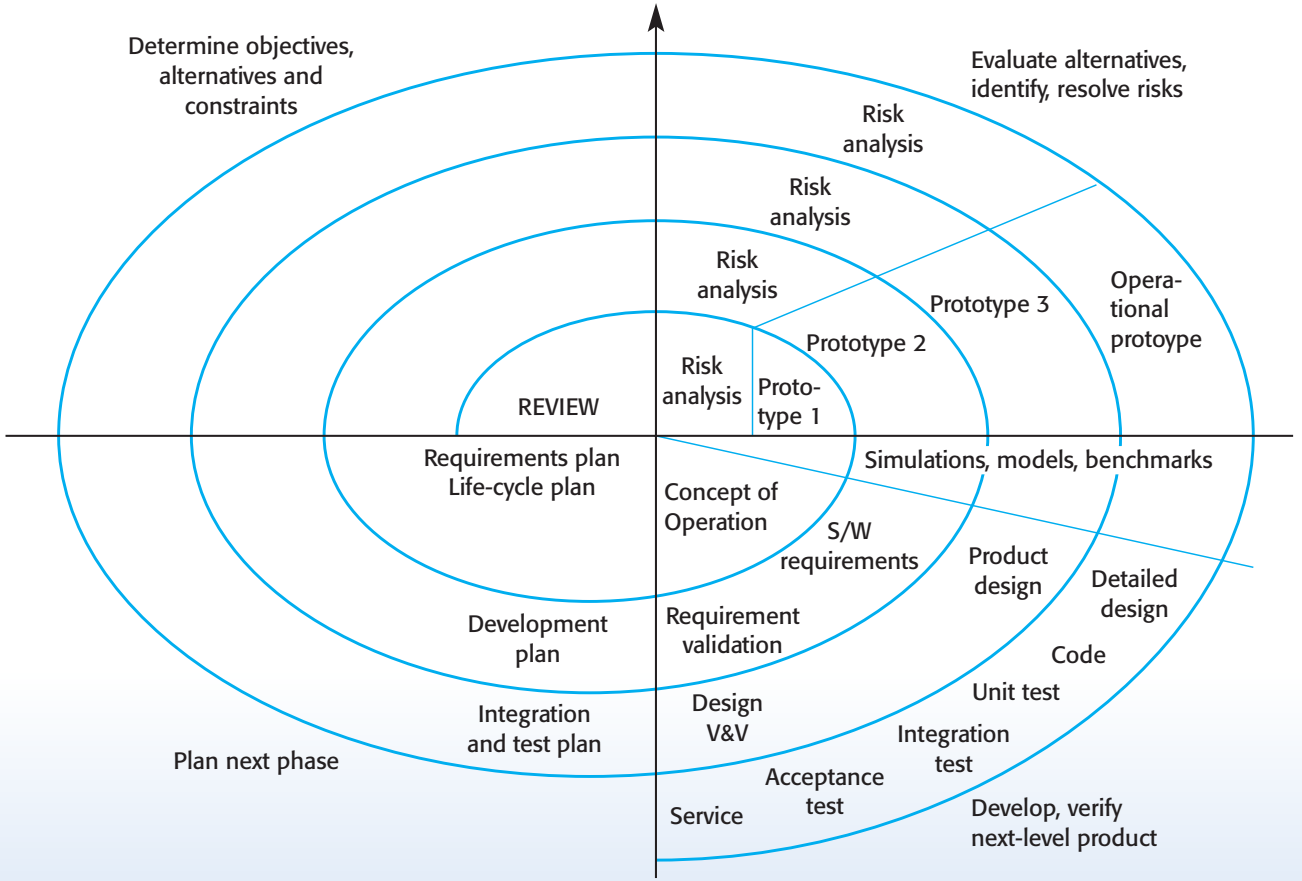
Incremental delivery **problems**

- Most systems require a set of basic facilities that are used by different parts of the system.
  - As requirements are *not defined in detail* until an increment is to be implemented, it can be **hard to identify common facilities** that are needed by all increments.
- If a replacement is being developed, feedback from users will be difficult because users usually tend to use a "complete" system than an incomplete one
- The essence of iterative processes is that the specification is developed in conjunction with the software.
  - However, this conflicts with the procurement model of many organizations, where the complete system specification is part of the system development contract.

3. Boehm's spiral model (Risk driven)

- *Combines* change avoidance and change tolerance
- Process is represented as a spiral rather than as a sequence of activities with backtracking.
- Each loop in the spiral represents a phase in the process.
- No fixed phases such as specification or design - loops in the spiral are chosen depending on what is required.
- Risks are explicitly assessed and resolved throughout the process.
  - Combines change avoidance and change tolerance
  - Changes are a result of project risks and the model includes explicit **risk management activities** to reduce these risks.

Figure 2.11 Boehm's spiral model of the software process



Spiral model sectors

- Objective setting
  - Specific objectives for the phase are identified. Constraints, risks and **alternatives** are also identified.
- **Risk assessment and reduction**
  - Risks are assessed and analysed. And activities put in place to reduce the key risks (e.g., introduce **prototype** for potential inappropriate req.). If reduction is not possible, terminate projects.
- Development and validation
  - A development model for the system is chosen which can be any of the generic models (e.g., prototyping, waterfall, formal etc).
- Planning
  - The project is reviewed and the next phase of the spiral is planned.

Spiral model usage

- Spiral model has been very influential in helping people think about iteration in software processes and introducing the risk-driven approach to development.
- In practice, however, the model is rarely used as published for practical software development.

Analysis of the Spiral Model

- Strengths
  - Support reuse of existing software (via alternative strategies)
  - It is easy to judge how much to test
    - Too much or too little is never good
  - No distinction is made between development and maintenance
    - But…it is very much like waterfall model
  - Stats show that at least 50% productivity increase by all projects (100% increase by most projects)
- Weaknesses
  - For large-scale software only
  - For internal (in-house) software only
  - Risk driven: Depend on developers' experience

CHAPTER 2
PART 3

4. The Rational Unified Process

- Created by IBM
- RUP is a specific implementation of the Unified Process (using UML).
- Combining the experience base of companies led to the articulation of six *best practices* for modern software engineering:
  - Develop iteratively, with risk as the primary iteration driver
  - Manage requirements
  - Employ a component-based architecture
  - Model software visually
  - Continuously verify quality
  - Control changes
- IBM created a subset of RUP that is tailored for the delivery of agile projects, called OpenUP.

The Rational Unified Process

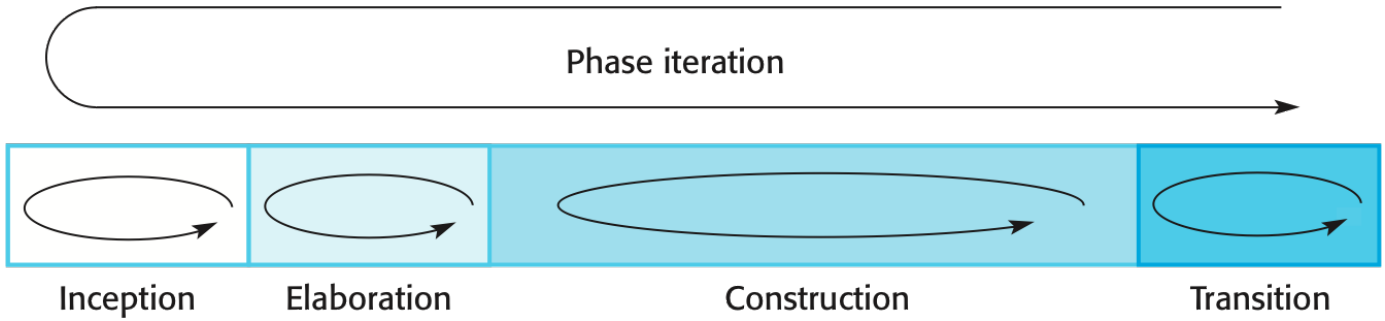
- Pros: (1) Brings together aspects of the 3 generic process models discussed previously (Waterfall, incremental, reuse-oriented); (2) Illustrate good practice in specification and design; and (3) support prototyping and incremental delivery.

✧ Normally described from 3 perspectives:

  - A dynamic perspective that shows phases over time;
  - A static perspective that shows process activities;
  - A practice perspective that suggests good practice.

- Phases here is related to **business** more, instead of the ones that describe process activities (technical concerns) in waterfall

Figure 2.12 Phases in the Rational Unified Process



Phases in the Rational Unified Process*

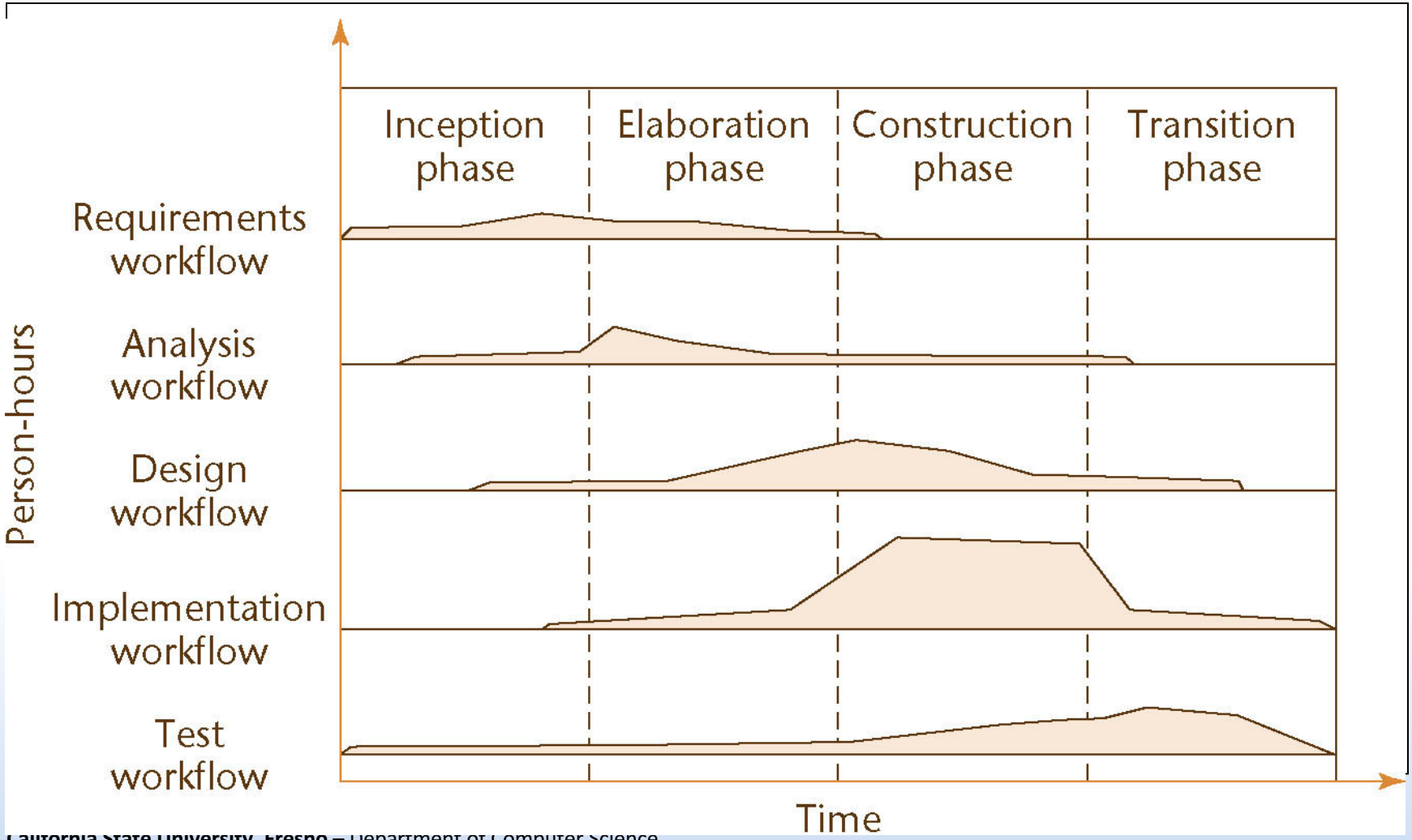


Figure 3.1

RUP phases (related to business/time)

- Inception
  - Understand the domain. Determine scope. Establish the business case for the system. (identify all external entities (people and systems) that will interact with the system and define these interactions.)
- Elaboration
  - Develop an understanding of the problem domain (use case) and the system architecture. Refine the architecture, identify and monitor the risks and refine their priorities, refine the business case, produce the project management plan)
- Construction
  - System design, programming and testing. i.e., produce the first operational-quality version of the software product (beta-release)
- Transition
  - Ensure that the client's requirements have indeed been met. Deploy the system in its operating environment.

More details can be found at http://en.wikipedia.org/wiki/RUP

The Rational Unified Process

- Within each iteration, the tasks are categorized into nine disciplines (the two-dimensional figure above with *):
  - Six "engineering disciplines" (Static view, also called workflows)
    - Business Modeling
    - Requirements
    - Analysis and Design
    - Implementation
    - Test
    - Deployment
  - Three supporting disciplines
    - Configuration and Change Management
    - Project Management
    - Environment

Static workflows in the Rational Unified Process

Workflow	Description
Business modelling	The business processes are modelled using business cases (e.g., business context, success factors, financial forecast, project plan, risk assessment, project description).
Requirements	Actors who interact with the system are identified and use cases are developed to model the system requirements.
Analysis and design	A design model is created and documented using architectural models, component models, object models and sequence models.
Implementation	The components in the system are implemented and structured into implementation sub-systems. Automatic code generation from design models helps accelerate this process.

Static workflows in the Rational Unified Process

Workflow	Description
Testing	Testing is an iterative process that is carried out in conjunction with implementation. System testing follows the completion of the implementation.
Deployment	A product release is created, distributed to users and installed in their workplace.
Configuration and change management	This supporting workflow managed changes to the system (see Chapter 25).
Project management	This supporting workflow manages the system development (see Chapters 22 and 23).
Environment	This workflow is concerned with making appropriate software tools available to the software development team.

RUP iteration

- In-phase iteration
  - Each phase is iterative with results developed incrementally.
- Cross-phase iteration
  - As shown by the loop in the RUP model, the whole set of phases may be enacted incrementally.

RUP phases

- In theory, there could be any number of increments
  - In practice, development seems to consist of four increments
- Every step performed in the Unified Process falls into
  - One of the five core workflows (slide with *) and *also*
  - One of the four phases (slide with *)
- Why does each step have to be considered twice (i.e., in two dimensions)?

RUP good practice

- ✧ Develop software iteratively
  - Plan increments based on customer priorities and deliver highest priority increments first.
- ✧ Manage requirements
  - Explicitly document customer requirements and keep track of changes to these requirements.
  - Analyze the impact of changes before accepting them.
- ✧ Use component-based architectures
  - Organize the system architecture as a set of reusable components.

RUP good practice

- Visually model software
  - Use graphical UML models to present static and dynamic views of the software.
- Verify software quality
  - Ensure that the software meet's organizational quality standards.
- Control changes to software
  - Manage software changes using a change management system and configuration management tools.

Additional SP models not covered in text

Synchronize-and Stabilize Model

- Microsoft's life-cycle model
- Requirements analysis — interview potential customers
- Draw up specifications
- Divide project into 3 or 4 builds
- Each build is carried out by small teams working in parallel

Synchronize-and Stabilize Model (contd)

- At the end of the day — *synchronize* (test and debug)
- At the end of the build — *stabilize* (freeze the build)
- Components always work together
  - Get early insights into the operation of the product
    - Which model is like this?

Synchronize-and-Stabilize Teams

- Used by Microsoft
- Products consist of 3 or 4 sequential builds
- Small parallel teams
  - 3 to 8 developers
  - 3 to 8 testers (work one-to-one with developers)
  - The team is given the overall task specification
  - They may design the task as they wish

Synchronize-and-Stabilize Teams (contd)

- Why this does not degenerate into hacker-induced chaos?
  - Daily synchronization step
  - Individual components always work together

Synchronize-and-Stabilize Teams (contd)

- Rules
  - Programmers must adhere to the time for entering the code into the database for that day's synchronization
- Analogy
  - Letting children do what they like all day…
  - … but with a 9 P.M. bedtime

Synchronize-and-Stabilize Teams (contd)

- Will this work in all companies?
  - Perhaps if the software professionals are as good as those at Microsoft
- Alternate viewpoint
  - The synchronize-and-stabilize model is simply a way of allowing a group of hackers to develop large products
  - Microsoft's success is due to superb marketing rather than quality software

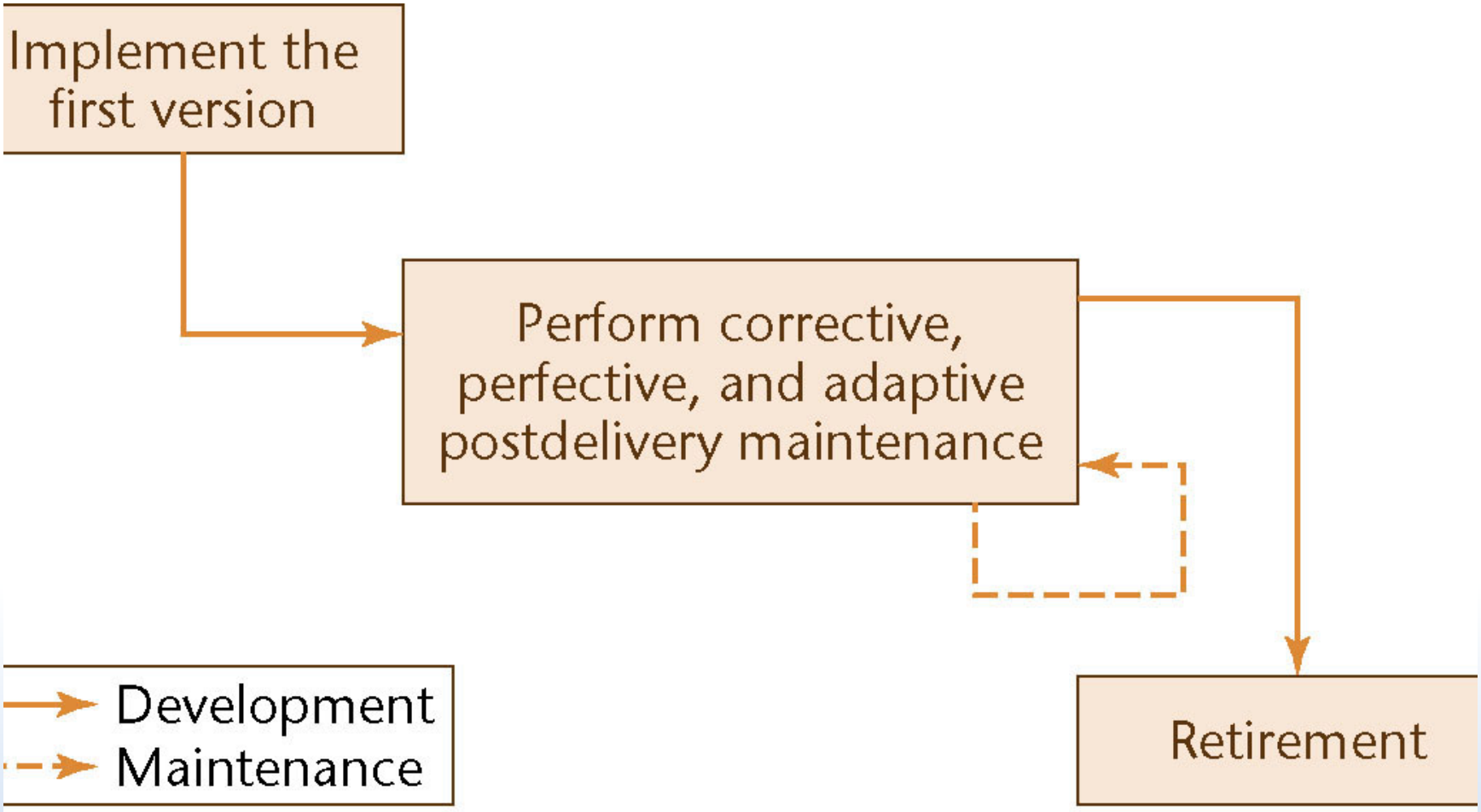
Open-Source Model

- 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 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 postdelivery maintenance
  - The word “co-developers” on the previous slide should rather be “co-maintainers”

Open-Source Life-Cycle Model (contd)



Open-Source Model (contd)

- Closed-source software is maintained and tested by employees
  - Users can submit failure reports but never fault reports (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

Open-Source Model (contd)

- 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 Model (contd)

- 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 Model

- 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:
  - Rapid-prototyping model
    - The initial version is discarded
  - Code-and-fix model and open-source life-cycle model
    - The initial version becomes the target product

Open-Source Model (contd)

- Consequently, in an open-source project, there are generally no specifications and no design

  Actually, when wiki becomes popular, many open source projects have rich documentation
- How have some open-source projects been so successful without specifications or designs?

Open-Source Model (contd)

- 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

Open-Source Model (contd)

- The open-source model is restricted in its applicability
- 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)

Open-Source Model (contd)

- There cannot be open-source development of a software product to be used in just one commercial organization
  - Members of both the core group and the periphery are invariably 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 (contd)

- 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 preponderance 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

Comparison of Life-Cycle Models (contd)

Iterative-and-incremental life-cycle model (Section 2.5)	Closely models real-world software production Underlies the Unified Process	
Code-and-fix life-cycle model (Section 2.9.1)	Fine for short programs that require no maintenance	Totally unsatisfactory for nontrivial programs
Waterfall life-cycle model (Section 2.9.2)	Disciplined approach Document driven	Delivered product may not meet client's needs
Rapid-prototyping life-cycle model (Section 2.9.3)	Ensures that the delivered product meets the client's needs	Not yet proven beyond all doubt
Open-source life-cycle model (Section 2.9.4)	Has worked extremely well in a small number of instances	Limited applicability Usually does not work
Agile processes (Section 2.9.5)	Works well when the client's requirements are vague	Appears to work on only small-scale projects
Synchronize-and-stabilize life-cycle model (Section 2.9.6)	Future users' needs are met Ensures that components can be successfully integrated	Has not been widely used other than at Microsoft
Spiral life-cycle model	Risk-driven	Can be used for only

Key points

- Processes should include activities to cope with change. This may involve a prototyping phase that helps avoid poor decisions on requirements and design.
- Processes may be structured for iterative development and delivery so that changes may be made without disrupting the system as a whole.
- The Rational Unified Process is a modern generic process model that is organized into phases (inception, elaboration, construction and transition) but separates activities (requirements, analysis and design, etc.) from these phases.