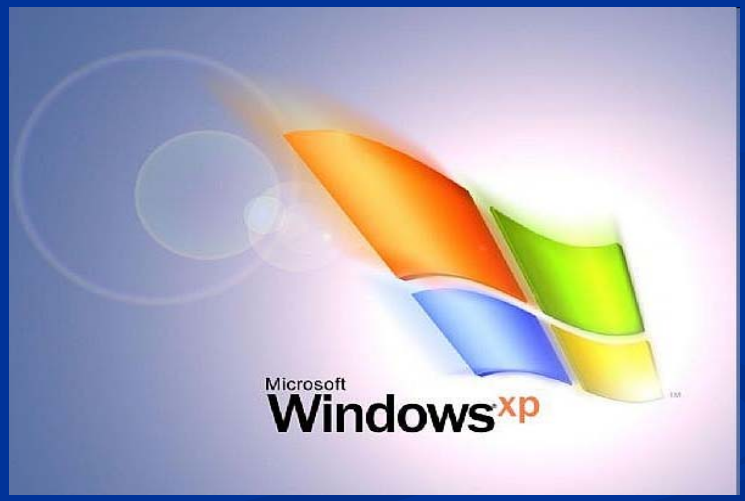
- **The Agenda for this lecture:**

  - Real Software Products are Engineered
  - Software Engineering Practices
  - Software Life-Cycle Models
  - UML
  - System Engineering
  - Requirements Engineering

Real Software Products are Engineered

Software Product Size

- **Windows Vista is said to have over 50 million lines of code, whereas XP was said to have around 40 million**



Software Product Size

- **Red Hat Linux version 7.1 (released April 2001) contained over 30 million SLOC**

Software Product Size

- **Mac OS X 10.4 has about 86 million SLOC**

Software Product Size

- **Debian 5.0 /GNU Linux has about 324 million SLOC**

Software Product Size

- **The Android operating system consists of 12 million lines of code SLOC**

Software Product Size

- **The Navigation system in the current S-class Mercedes-Benz requires over 20 million lines of code**

Software Product Size

- The avionics system in the F-22 Raptor, the current U.S. Air Force frontline jet fighter, consists of about 1.7 million lines of software code.
- The F-35 Joint Strike Fighter requires about 5.7 million lines of code to operate its onboard systems.
- Boeing's new 787 Dreamliner requires about 6.5 million lines of software code to operate its avionics and onboard support systems.

Software Engineering Practice: A Generic View

What is “Practice”?

- Practice is a broad array of concepts, principles, methods, and tools that you must consider as software is planned and developed.
- It represents the details—the technical considerations and how to’s—that are below the surface of the software process—the things that you’ll need to actually build high-quality computer software.

The Essence of Practice

- George Polya, in a book written in 1945 (!), describes the essence of software engineering practice …
  - *Understand the problem* (communication and analysis).
  - *Plan a solution* (modeling and software design).
  - *Carry out the plan* (code generation).
  - *Examine the result for accuracy* (testing and quality assurance).
- At its core, good practice is common-sense problem solving

Core Software Engineering Principles

- Provide value to the customer and the user
- KIS—keep it simple!
- Maintain the product and project “vision”
- What you produce, others will consume
- Be open to the future
- Plan ahead for reuse
- Think!

Software Engineering Practices

- Consider the generic process framework
  - Communication
  - Planning
  - Modeling
  - Construction
  - Deployment
- Here, we'll identify
  - Underlying principles
  - How to initiate the practice
  - An abbreviated task set

Communication Practices

- Principles
  - Listen
  - Prepare before you communicate
  - Facilitate the communication
  - Face-to-face is best
  - Take notes and document decisions
  - Collaborate with the customer
  - Stay focused
  - Draw pictures when things are unclear
  - Move on …
  - Negotiation works best when both parties win.

Communication Practices

- Initiation
  - The parties should be physically close to one another
  - Make sure communication is interactive
  - Create solid team “ecosystems”
  - Use the right team structure
- An abbreviated task set
  - Identify who it is you need to speak with
  - Define the best mechanism for communication
  - Establish overall goals and objectives and define the scope
  - Get more detailed
    - Have stakeholders define scenarios for usage
    - Extract major functions/features
  - Review the results with all stakeholders

Planning Practices

- Principles
  - Understand the project scope
  - Involve the customer (and other stakeholders)
  - Recognize that planning is iterative
  - Estimate based on what you know
  - Consider risk
  - Be realistic
  - Adjust granularity as you plan
  - Define how quality will be achieved
  - Define how you'll accommodate changes
  - Track what you've planned

Planning Practices

- Initiation
  - Ask Boehm's questions
    - Why is the system begin developed?
    - What will be done?
    - When will it be accomplished?
    - Who is responsible?
    - Where are they located (organizationally)?
    - How will the job be done technically and managerially?
    - How much of each resource is needed?

Planning Practices

- An abbreviated task set
  - Re-assess project scope
  - Assess risks
  - Evaluate functions/features
  - Consider infrastructure functions/features
  - Create a coarse granularity plan
    - Number of software increments
    - Overall schedule
    - Delivery dates for increments
  - Create fine granularity plan for first increment
  - Track progress

Modeling Practices

- We create models to gain a better understanding of the actual entity to be built
- *Analysis models* represent the customer requirements by depicting the software in three different domains: the information domain, the functional domain, and the behavioral domain.
- *Design models* represent characteristics of the software that help practitioners to construct it effectively: the architecture, the user interface, and component-level detail.

Analysis Modeling Practices

- Analysis modeling principles
  - Represent the information domain
  - Represent software functions
  - Represent software behavior
  - Partition these representations
  - Move from essence toward implementation
- Elements of the analysis model
  - Data model
  - Flow model
  - Class model
  - Behavior model

Design Modeling Practices

- Principles
  - Design must be traceable to the analysis model
  - Always consider architecture
  - Focus on the design of data
  - Interfaces (both user and internal) must be designed
  - Components should exhibit functional independence
  - Components should be loosely coupled
  - Design representation should be easily understood
  - The design model should be developed iteratively
- Elements of the design model
  - Data design
  - Architectural design
  - Component design
  - Interface design

Construction Practices

- Preparation principles: *Before you write one line of code, be sure you:*
  - Understand the problem you're trying to solve (see communication and modeling)
  - Understand basic design principles and concepts.
  - Pick a programming language that meets the needs of the software to be built and the environment in which it will operate.
  - Select a programming environment that provides tools that will make your work easier.
  - Create a set of unit tests that will be applied once the component you code is completed.

Construction Practices

- Coding principles: *As you begin writing code, be sure you:*
  - Constrain your algorithms by following structured programming [BOH00] practice.
  - Select data structures that will meet the needs of the design.
  - Understand the software architecture and create interfaces that are consistent with it.
  - Keep conditional logic as simple as possible.
  - Create nested loops in a way that makes them easily testable.
  - Select meaningful variable names and follow other local coding standards.
  - Write code that is self-documenting.
  - Create a visual layout (e.g., indentation and blank lines) that aids understanding.

Construction Practices

- Validation Principles: *After you've completed your first coding pass, be sure you:*
  - Conduct a code walkthrough when appropriate.
  - Perform unit tests and correct errors you've uncovered.
  - Refactor the code.

Construction Practices

- Testing Principles
  - All tests should be traceable to requirements
  - Tests should be planned
  - Testing begins “in the small” and moves toward “in the large”
  - Exhaustive testing is not possible

Deployment Practices

- Principles
  - Manage customer expectations for each increment
  - A complete delivery package should be assembled and tested
  - Instructional materials must be provided to end-users
  - Buggy software should be fixed first, delivered later

Software Engineering:
Classical and Object-Oriented

- Is there a relationship between classical software engineering and Object-oriented software Engineering?
- Structured analysis vs. object-oriented analysis

Ask yourself the following questions:

- Where do I start?
- When do I need to talk to the customer?
- What requirements do I need to gather?
- Do I need to think about Analysis?
- Do I need to think about Design (high level/low level)?
- How can I capture the requirements?
  - Informal (Natural Language)
  - Formal (Specification and Modeling Languages)
- There are so many things involved, how I prioritize?
- How can I cope with complexity?

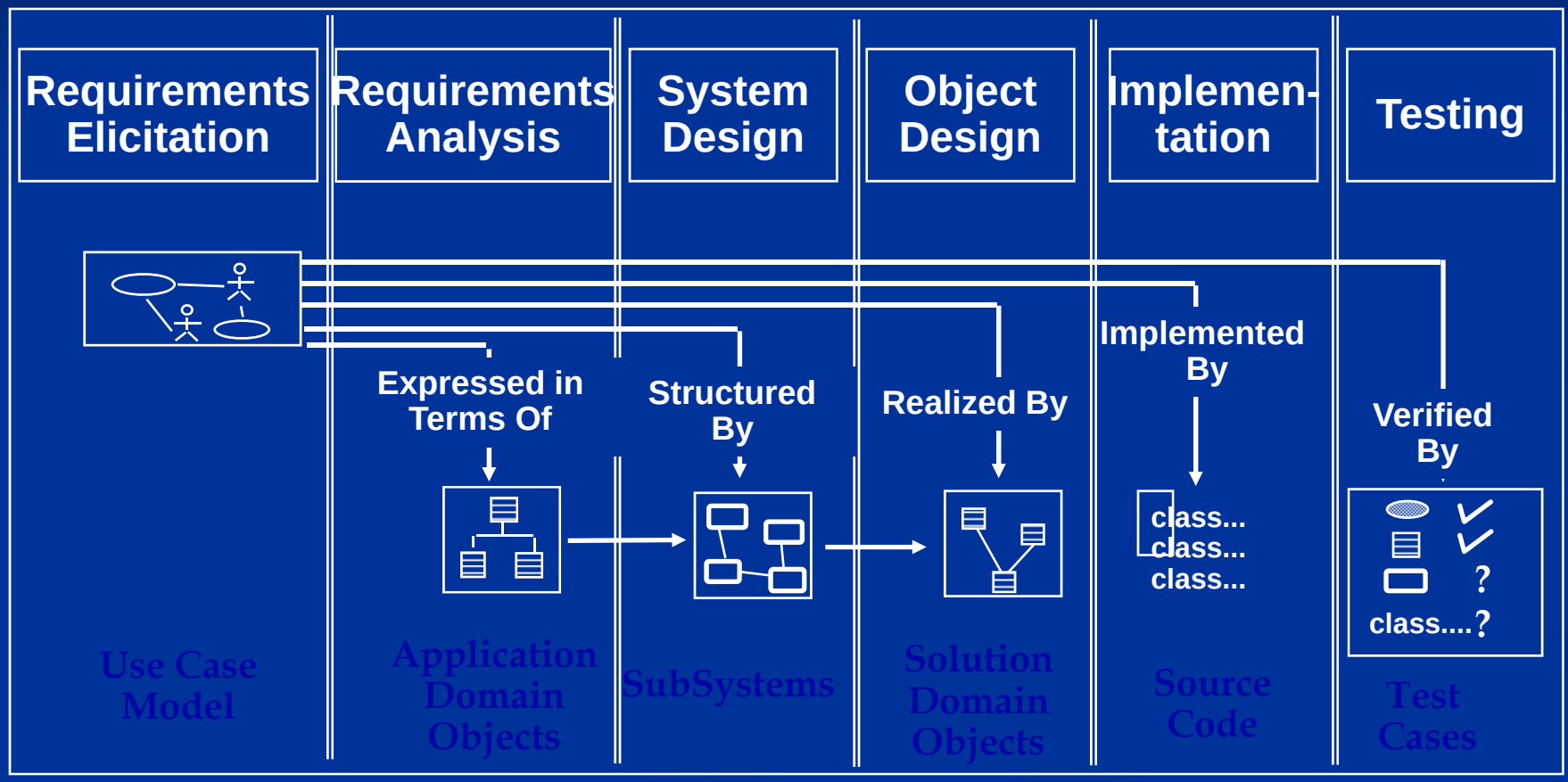
Where are we right now?

- Three ways to deal with complexity:
  - Abstraction
  - Decomposition (Technique: Divide and conquer)
  - Hierarchy (Technique: Layering)
- Two ways to deal with decomposition:
  - Object-orientation and functional decomposition
  - Functional decomposition leads to unmaintainable code
  - Depending on the purpose of the system, different objects can be found
- What is the right way?
  - Start with a description of the functionality (Use case model). Then proceed by finding objects (object model).
- What activities and models are needed?
  - This leads us to the software lifecycle used in OO paradigm …

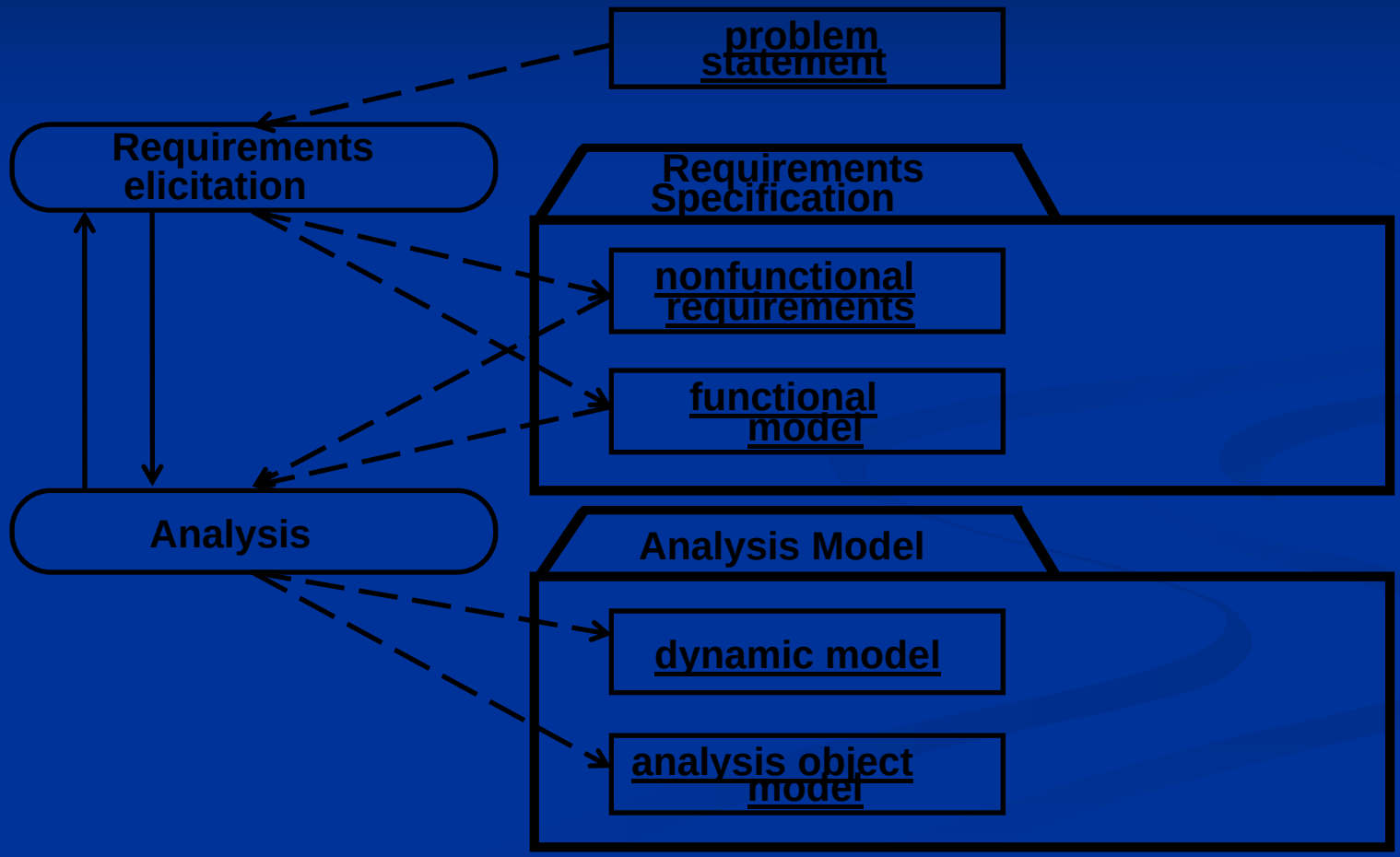
Software Lifecycle Definition

- Software lifecycle:
  - Set of activities and their relationships to each other to support the development of a software system
- Typical Lifecycle questions:
  - Which activities should I select for the software project?
  - What are the dependencies between activities?
  - How should I schedule the activities?
  - What is the result of an activity

Software Lifecycle Activities



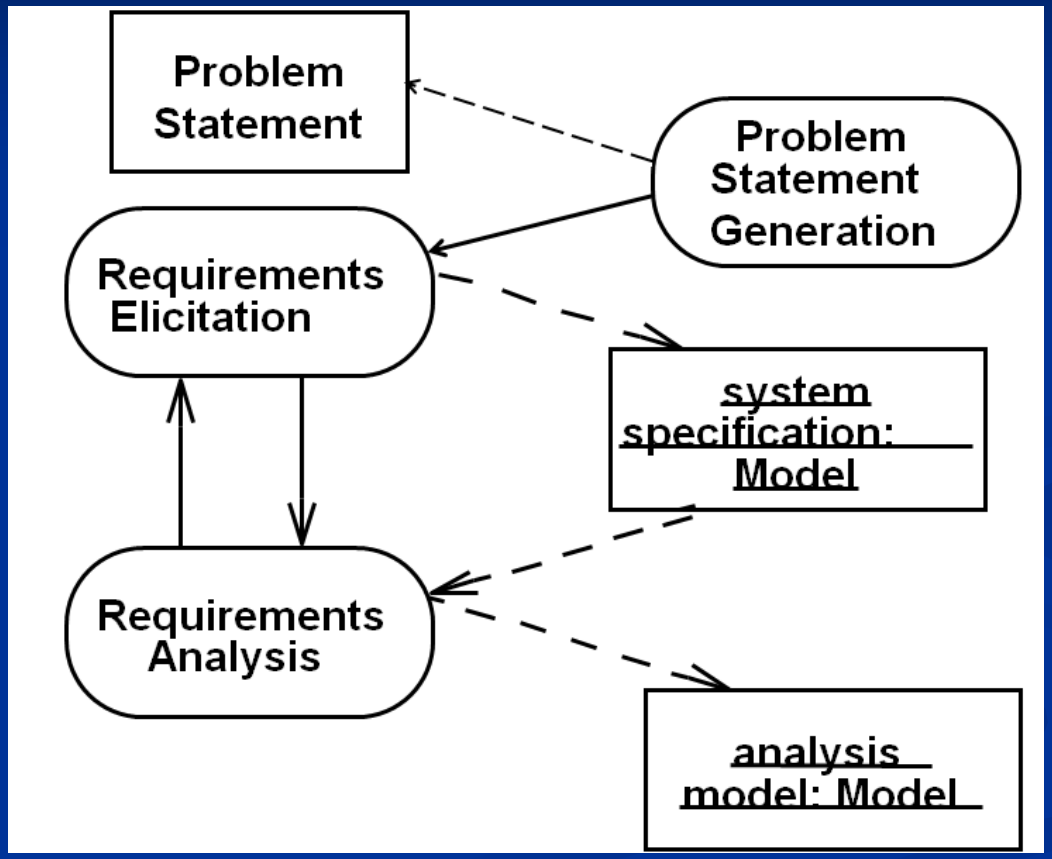
Products of requirements elicitation and analysis.



First Step in Establishing the Requirements: System Identification

- The development of a system is not just done by taking a snapshot of a scene (domain)
- Two questions need to be answered:
  - How can we identify the purpose of a system?
  - Crucial is the definition of the system boundary: What is inside, what is outside the system?
- These two questions are answered in the requirements process
- The requirements process consists of two activities:
  - Requirements Elicitation:
    - Definition of the system in terms understood by the customer ("Problem Description")
  - Requirements Analysis:
    - Technical specification of the system in terms understood by the developer ("Problem Specification")

Products of Requirements Process



Requirements Elicitation

- Very challenging activity
- Requires collaboration of people with different backgrounds
  - Users with application domain knowledge
  - Developer with solution domain knowledge (design knowledge, implementation knowledge)
- Bridging the gap between user and developer:
  - *Scenarios:* Example of the use of the system in terms of a series of interactions with between the user and the system
  - *Use cases:* Abstraction that describes a class of scenarios

System Specification vs Analysis Model

- Both models focus on the requirements from the user's view of the system.
- ***System specification*** uses natural language (derived from the *problem statement*)
- The ***analysis model*** uses formal or semi-formal notation (for example, a graphical language like UML)

- The starting point is the problem statement

Problem Statement

- The problem statement is developed by the client as a description of the problem addressed by the system
- Other words for problem statement:
  - Statement of Work
- A good problem statement describes
  - The current situation
  - The functionality the new system should support
  - The environment in which the system will be deployed
  - Deliverables expected by the client
  - Delivery dates
  - A set of acceptance criteria

Ingredients of a Problem Statement

- Current situation: The Problem to be solved
- Description of one or more scenarios
- Requirements
  - Functional and Nonfunctional requirements
  - Constraints (“pseudo requirements”)
- Project Schedule
  - Major milestones that involve interaction with the client including deadline for delivery of the system
- Target environment
  - The environment in which the delivered system has to perform a specified set of system tests
- Client Acceptance Criteria
  - Criteria for the system tests

SOFTWARE LIFE-CYCLE MODELS

SOFTWARE LIFE-CYCLE MODELS

- Why you need the software development process?
- Perspective: Software Engineer and Project Management
- Software development in theory
- Iteration and incrementation
- Risks and other aspects of iteration and incrementation
- Managing iteration and incrementation
- Other life-cycle models
- Comparison of life-cycle models
- Rational Unified Process

Why Software development processes?

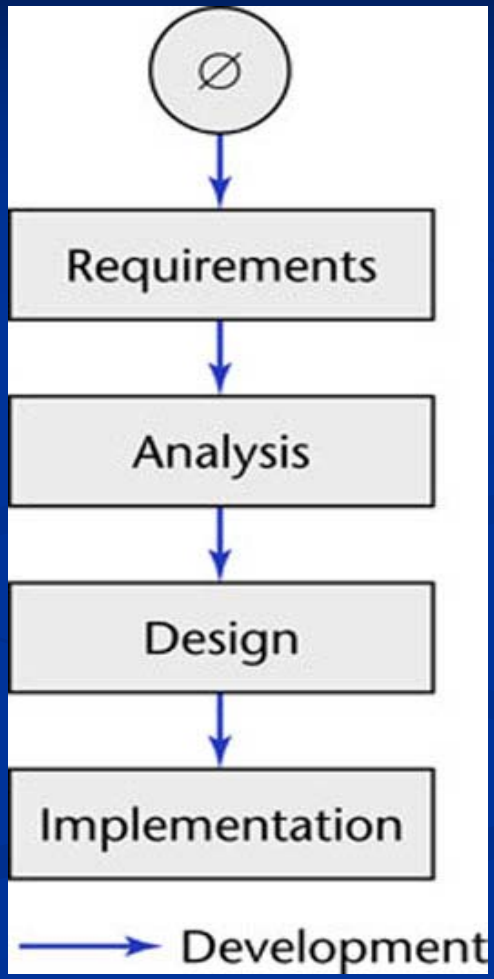
- Software Development Processes:
  - Improve Software quality
  - Improve software development cost estimates and reduce budget overruns
  - Improve software development schedule estimates and reduce schedule overruns
- In Software Project Management:
  - Triple-Constraints
    - Quality
    - Budget
    - Schedule

Perspective: Software Engineer and Project Management

- What is Software Engineer?
- What is Software Project Management?
- Why to study Software Development Processes?
- Software Quality:
  - Quality of Product
  - Quality of Process

Software Development in Theory

- Ideally, software is developed Linear



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

Root Causes of Software Development Problems

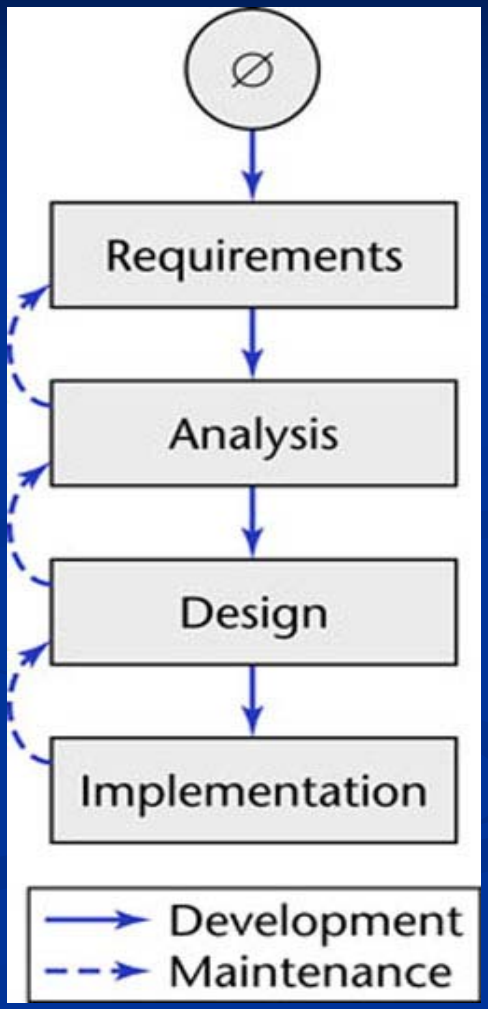
- Symptoms that characterise failed projects
  - Inaccurate understanding of end-user needs
  - Inability to deal with changing requirements
  - Modules that do not fit together
  - Software that is hard to maintain or extend
  - Late discovery of serious project flaws
  - Poor software quality
  - unacceptable software performance
  - untrustworthy build-and-release processes

Root Causes of Software Development Problems

- Many projects fail because there is
  - Ad hoc requirements management
  - Ambiguous and imprecise communication
  - Brittle architectures
  - Overwhelming complexity
  - Undetected inconsistencies in requirements, designs and implementations
  - Insufficient testing
  - Subjective assessment of project status

Waterfall Model

- The linear life cycle model with feedback loops



Moving Target Problem

- A change in the requirements while the software product is being developed
- Even if the reasons for the change are good, the software product can be adversely impacted

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

- Change is inevitable
  - Growing companies are always going to change
- If there are too many changes
  - The entire product may have to be redesigned and reimplemented

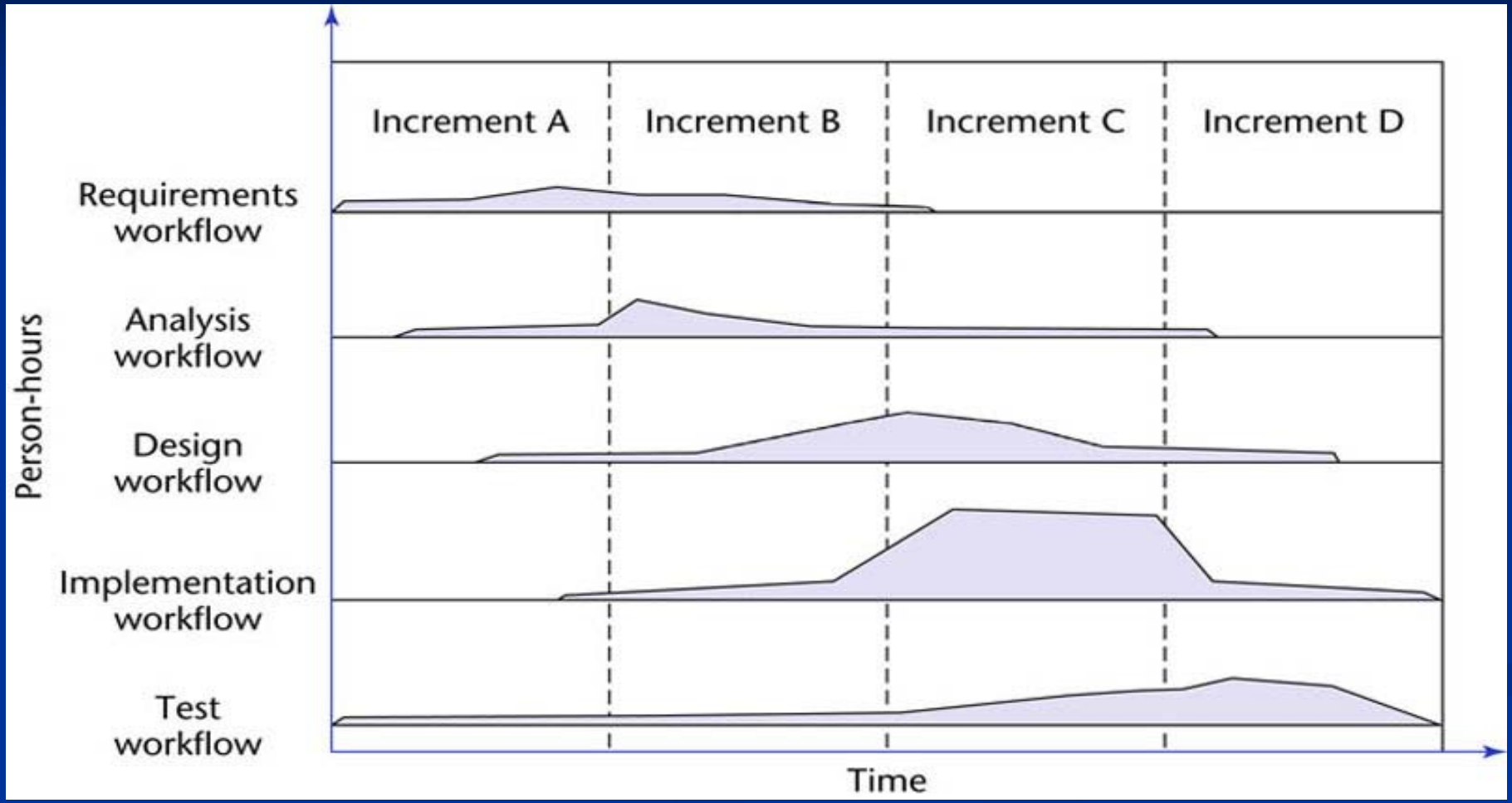
Iteration and Incrementation

- In real life, 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

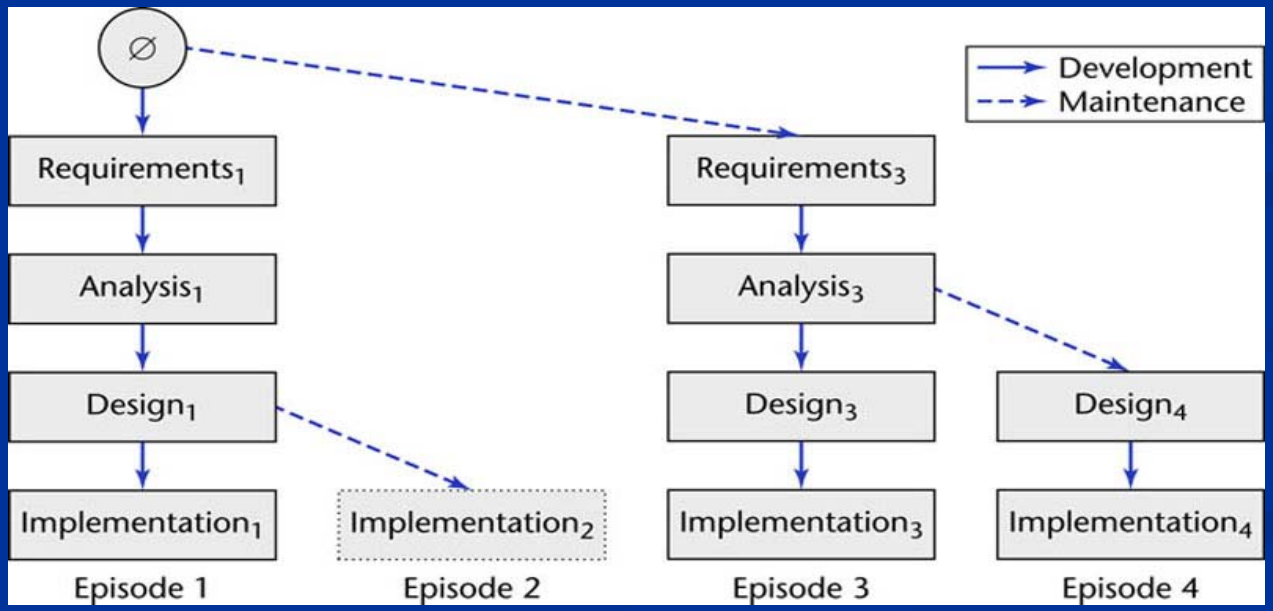
- At any one time, we can concentrate on only approximately seven *chunks* (units of information)
- To handle larger amounts of information, use *stepwise refinement*
  - Concentrate on the aspects that are currently the most important
  - Postpone aspects that are currently less critical
  - Every aspect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

Iteration and Incrementation



Iteration and Incrementation (contd)

- 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



Iteration and Incrementation (contd)

- The number of increments will vary—it does not have to be four

Classical Phases versus Workflows

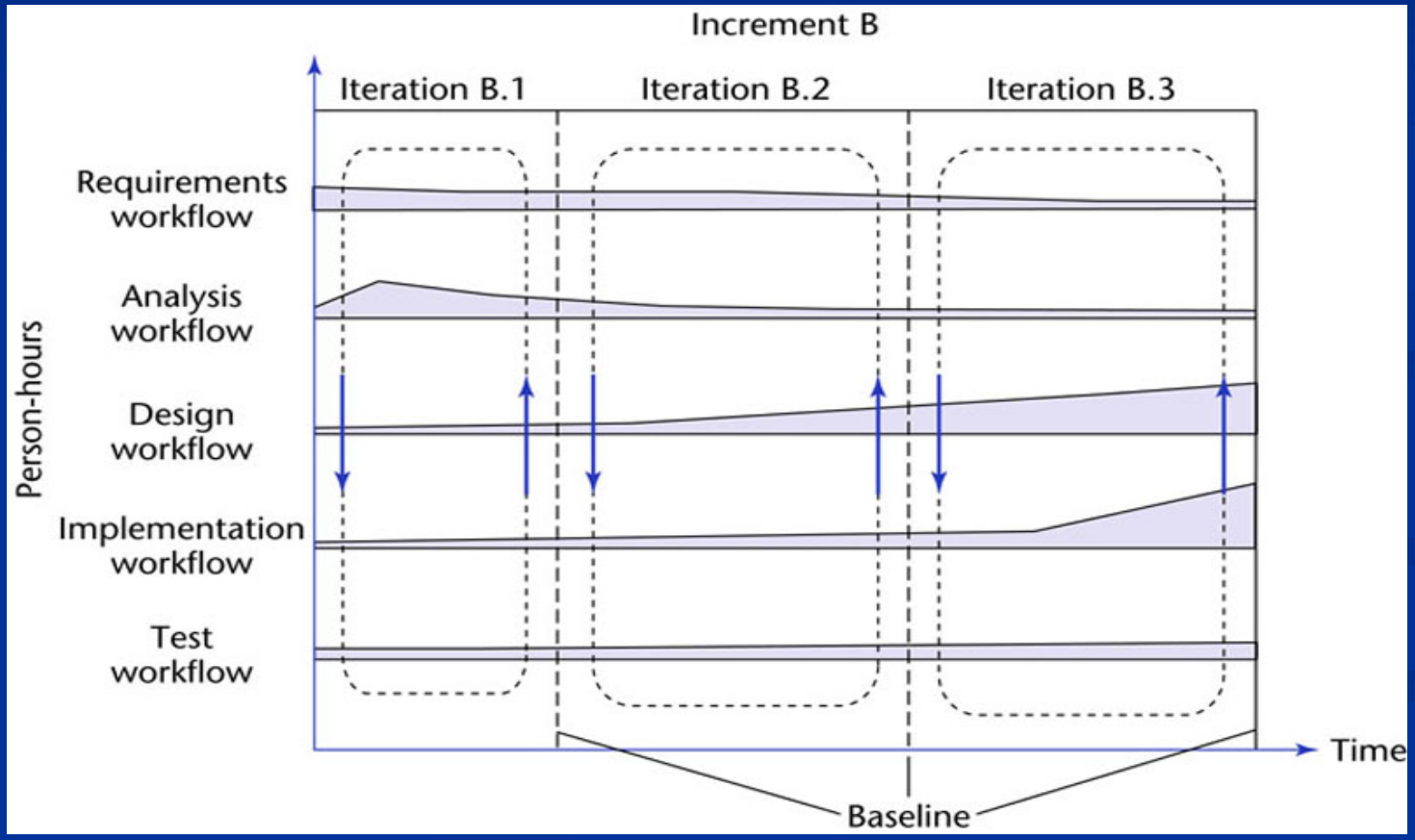
- Sequential phases do not exist in the real world
- Instead, the five core workflows (activities) are performed over the entire life cycle
  - Requirements workflow
  - Analysis workflow
  - Design workflow
  - Implementation workflow
  - Test workflow

Workflows

- All five core workflows are performed over the entire life cycle
- However, at most times one workflow predominates
- Examples:
  - At the beginning of the life cycle
    - The requirements workflow predominates
  - At the end of the life cycle
    - The implementation and test workflows predominate
- Planning and documentation activities are performed throughout the life cycle

Iteration and Incrementation (contd)

- Iteration is performed during each incrementation



Iteration and Incrementation (contd)

- Again, the number of iterations will vary—it is not always three

More on Incrementation (cont.)

- Each episode corresponds to an increment
- Not every increment includes every workflow
- Dashed lines denote maintenance
  - Corrective maintenance, in all three instances

Risks and Other Aspects of Iteration and Incrementation

- We can consider the project as a whole as a set of mini projects (increments)
- Each mini project extends the
  - Requirements artifacts
  - Analysis artifacts
  - Design artifacts
  - Implementation artifacts
  - Testing artifacts
- The final set of artifacts is the complete product

Risks and Other Aspects of Iter. and Increm. (contd)

- During each mini project we
  - Extend the artifacts (**incrementation**);
  - Check the artifacts (test workflow); and
  - If necessary, change the relevant artifacts (iteration)

Risks and Other Aspects of Iter. and Increm. (contd)

- Each iteration can be viewed as a small but complete waterfall life-cycle model
- During each iteration we select a portion of the software product
- On that portion we perform the
  - Classical requirements phase
  - Classical analysis phase
  - Classical design phase
  - Classical implementation phase

Strengths of the Iterative-and-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
- 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
  - What is the difference between flexibility and extensibility.

Strengths of the Iterative-and-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
  - Variation: Deliver partial versions to smooth the introduction of the new product in the client organization

Managing Iteration and Incrementation

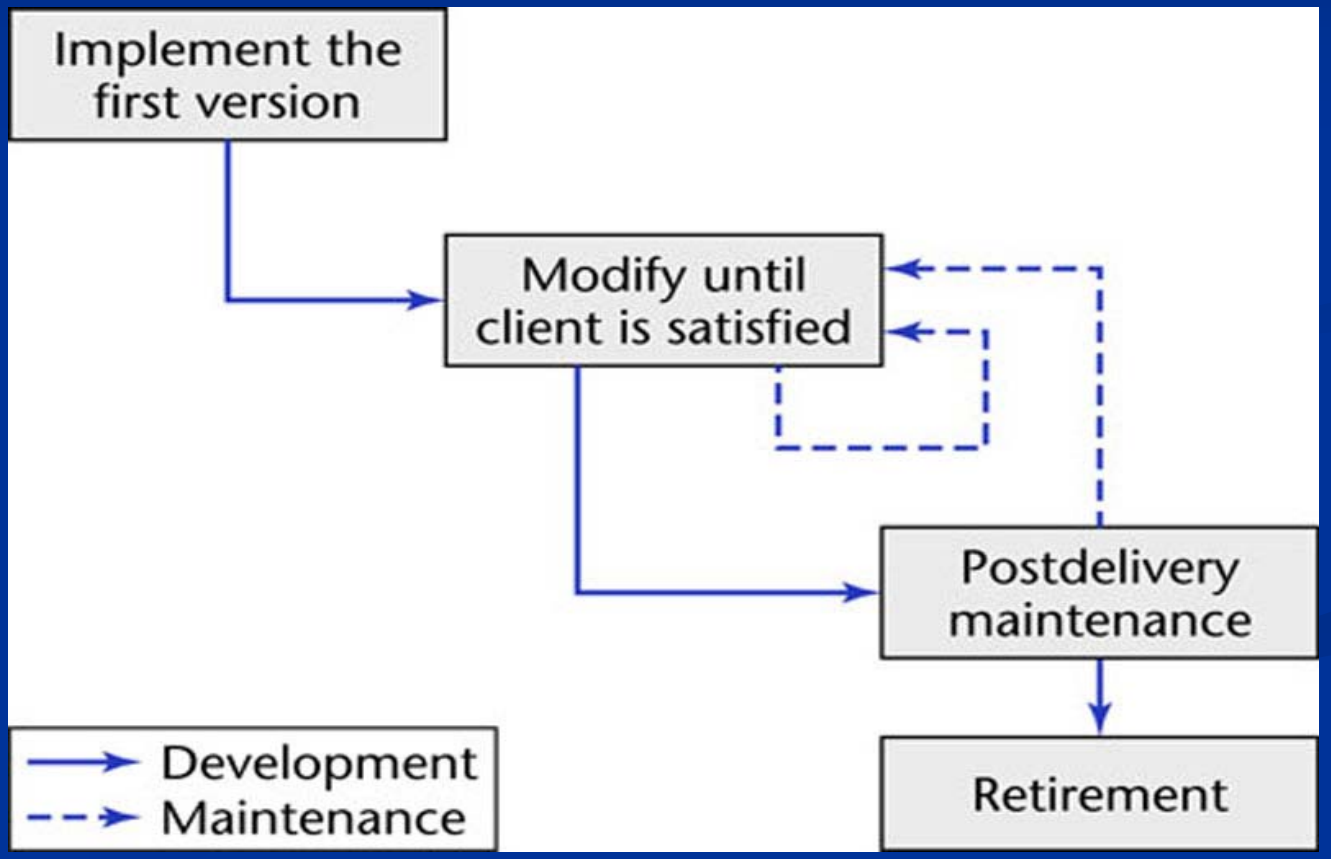
- The iterative-and-incremental life-cycle model *is* the waterfall model, applied successively
- Each increment is a waterfall mini project

Other Life-Cycle Models

- Now, we look into the different life-cycle models and give a comparative analysis at the end regard their weaknesses and strengths:
  - Code-and-fix life-cycle model
  - Waterfall life-cycle model
  - Rapid prototyping life-cycle model
  - Extreme programming and agile processes
  - Synchronize-and-stabilize life-cycle model
  - Spiral life-cycle model

Code-and-Fix Model

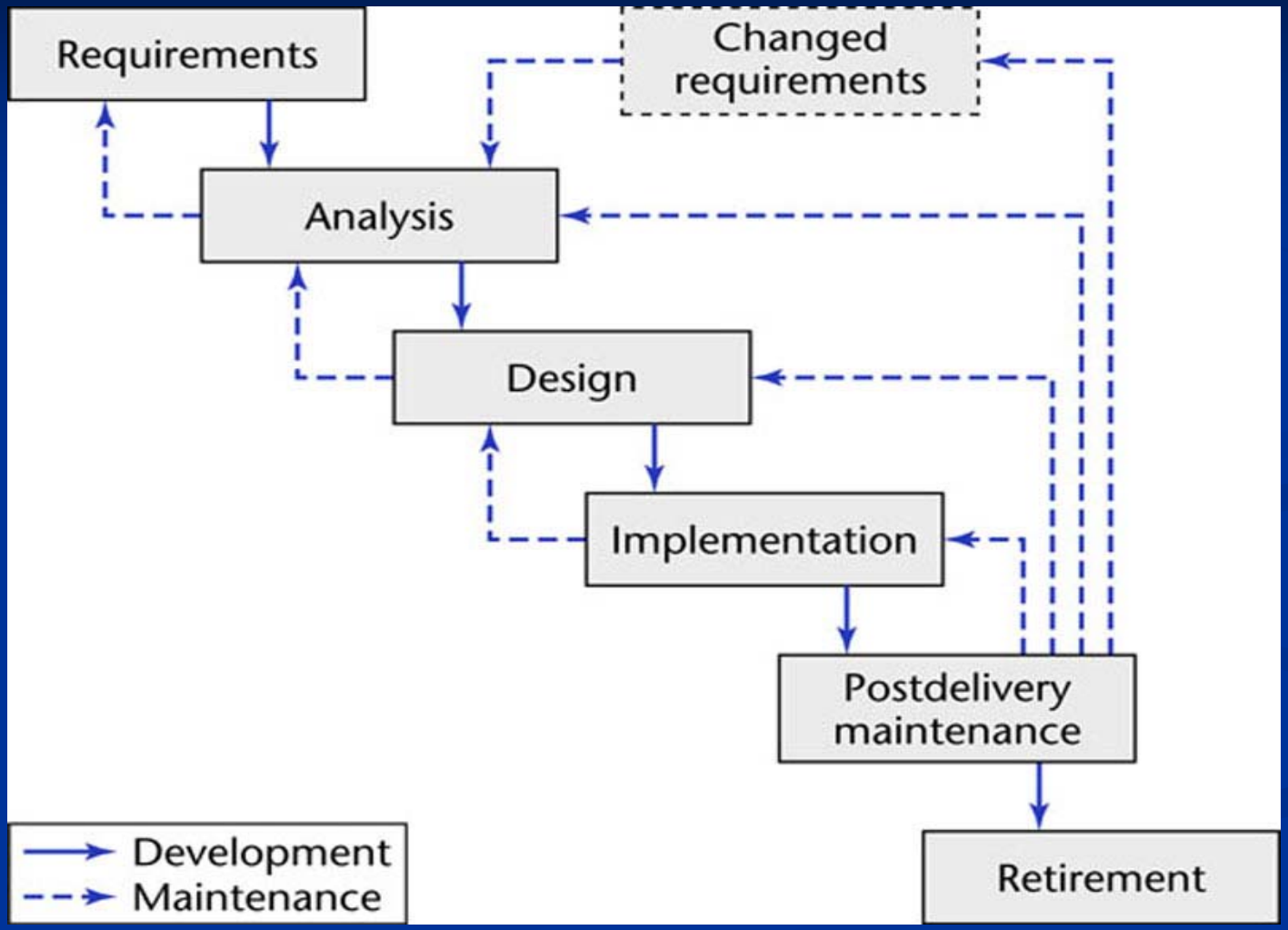
- No design
- No specifications
  - Maintenance nightmare



Code-and-Fix Model (contd)

- The easiest way to develop software
- The most expensive way

Waterfall Model

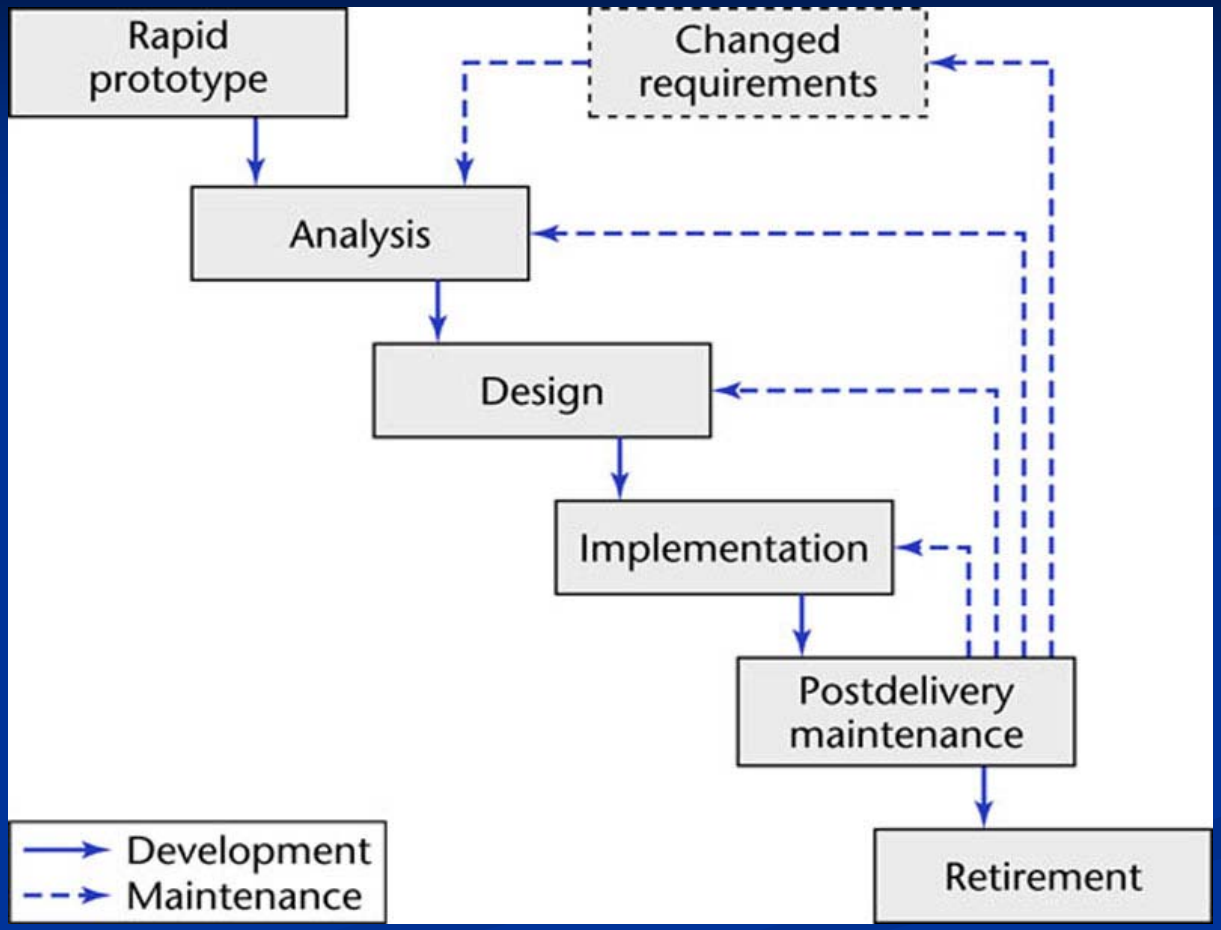


Waterfall Model (contd)

- Characterized by
  - Feedback loops
  - Documentation-driven
- Advantages
  - Documentation and quality records after every phase; SQA team will be the dog-watch
  - Maintenance is easier
- Disadvantages
  - Specification document ambiguous, not complete, written in an informal language (natural language)

Rapid Prototyping Model

- Linear model
- The Key is "Rapid Prototype"
- The sole purpose of the Rapid Prototype is to determine what the client's real needs are



Extreme Programming and Agile Processes

- 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
- All XP members work on analysis, design, coding, and testing
- There is no overall design document before various builds, instead the design is being modified while the product is being built; this procedure is termed **refactoring**.
- Pair programming
- Continuous integration of tasks

Extreme Programming and Agile Processes

- XP is one example of a number of new models that are collectively referred to as Agile Processes
- So what is the Agile Process?

Agile Processes

- 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

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

Synchronize-and Stabilize Model

- Microsoft's life-cycle model
- Requirements analysis — interview potential customers; extract a list of feature of highest priority to the clients
- Draw up specifications
- Divide project into 3 or 4 builds
  - 1st build consist of most critical features
  - 2nd build next most critical features
  - And so on
- Each build is carried out by small teams working in parallel

Synchronize-and Stabilize Model (contd)

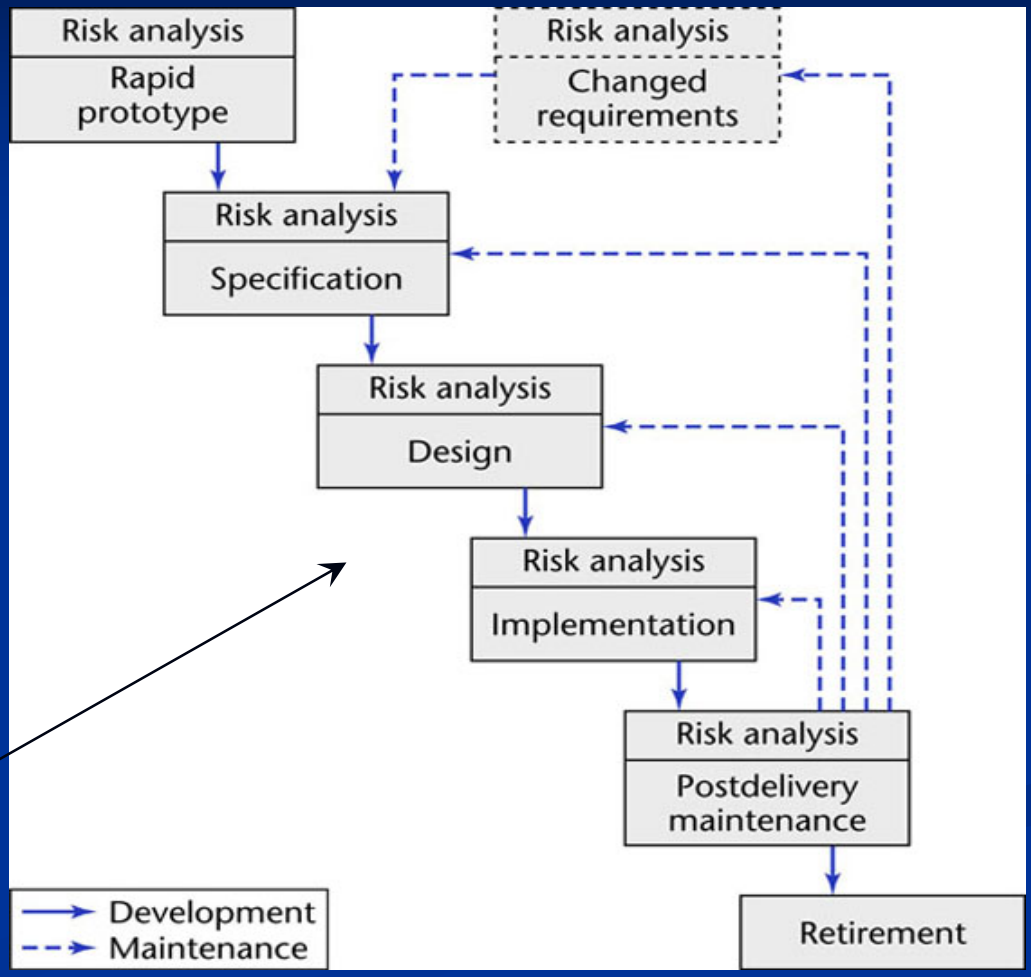
- At the end of the day — *synchronize* (test and debug)
- At the end of the build — *stabilize* (freeze the build)
- The repeated synchronization step ensures that the various components always work together
  - Get early insights into the operation of the product
- Clearly in this model, integration, interoperability , and stability are very crucial

Spiral Model

- The idea on minimizing risk via the use of prototypes is the underlying principle of the spiral model paper published by [Barry Boehm 1988], IEEE Computer.

- Rapid prototyping model plus risk analysis preceding each phase

A Simplified version of the Spiral Model



A Key Point of the Spiral Model

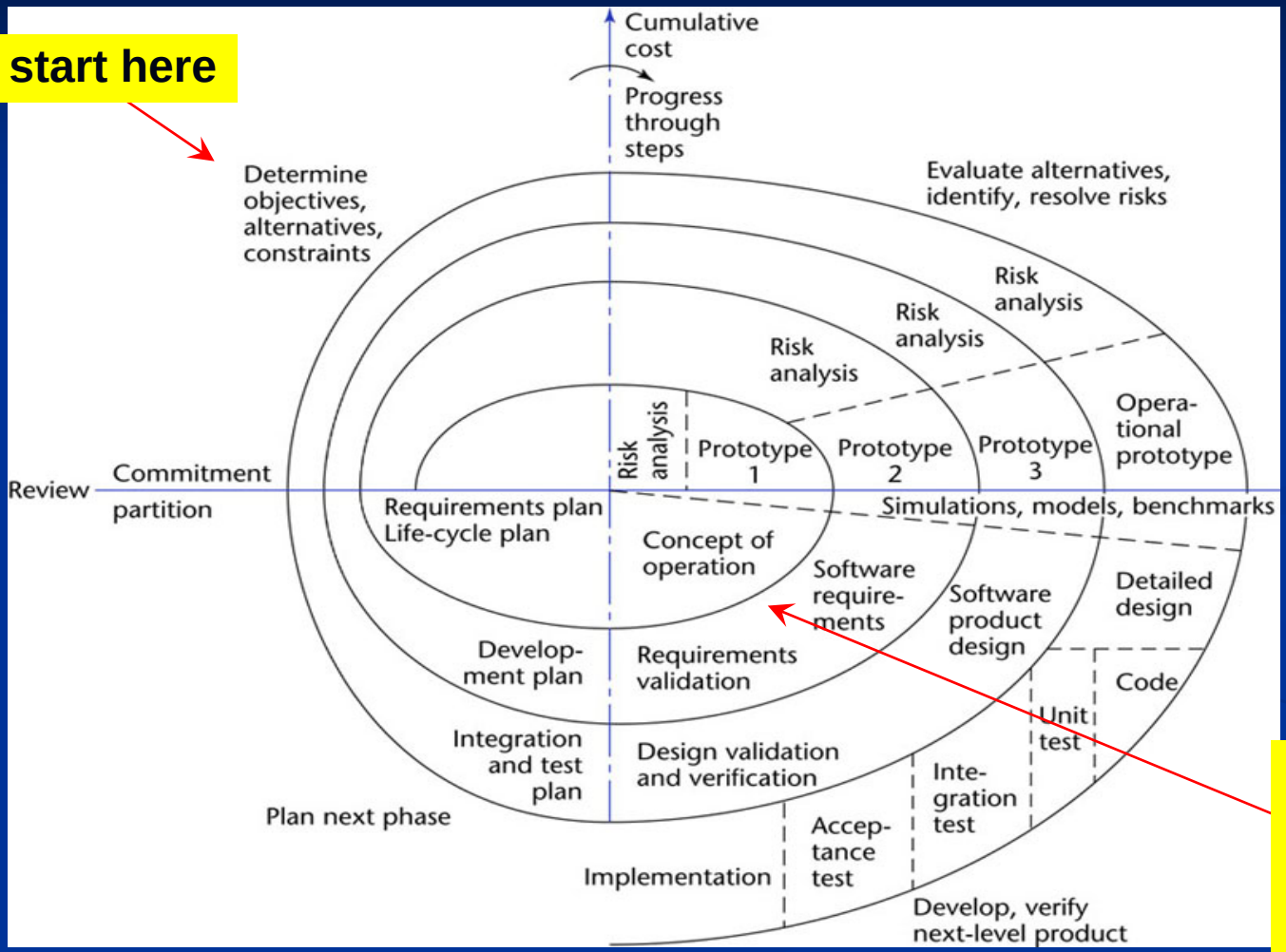
- If all risks cannot be mitigated, the project is immediately terminated

- Two kinds of software projects:
  - R&D Projects
  - Contracted Projects

Full Spiral Model

- Each Cycle of the spiral corresponds to a phase
  - Requirements, Design, Implementation, etc.
- Each Phase starts (begins in the top left quadrant) by determining objectives of that phase
- Precede each phase by
  - Alternatives
  - Risk analysis
- Follow each phase by
  - Evaluation
  - Planning of the next phase
- Radial dimension: cumulative cost to date
- Angular dimension: progress through the spiral

Full Spiral Model (contd)



This quadrant corresponds to classical waterfall model

Analysis of the Spiral Model

- Strengths
  - It is easy to judge how much to test in terms of the risks that would be incurred by not doing enough testing or too much of testing
    - Too much testing, expensive and products will be unduly delayed
    - Too little testing, may result in too many defects discovered by the customer
  - No distinction is made between development and maintenance. Postdelivery maintenance is simply another cycle of the spiral.

- Weaknesses
  - For large-scale software only
  - For internal (in-house) software only
  - Not for external customers since there is always a contract, and terminating the contract from either side can lead to a breach-of-contract lawsuit

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

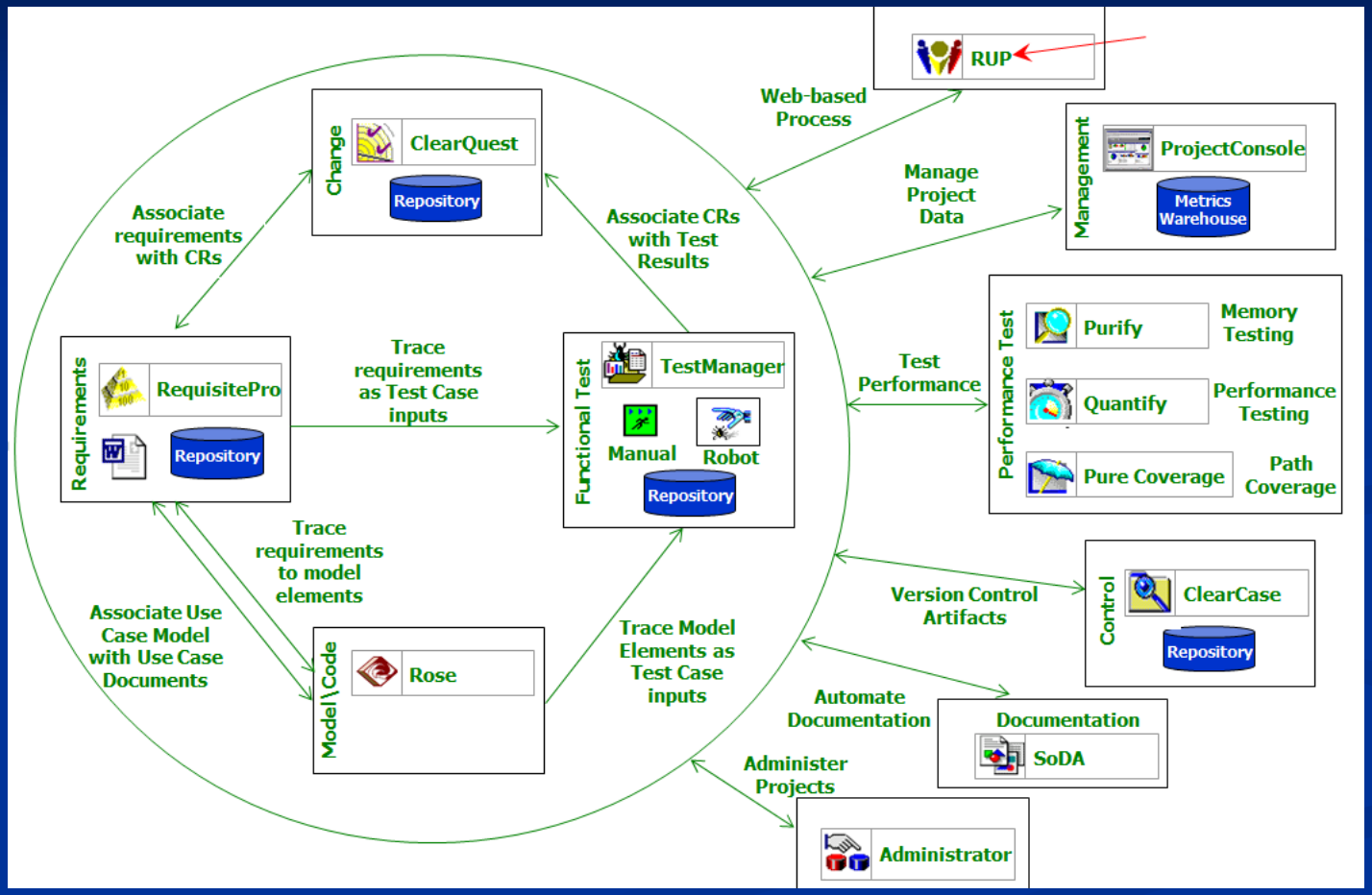
Comparison of Life-Cycle Models (contd)

Life-Cycle Model	Strengths	Weaknesses
Iterative-and-incremental Model	Closely models real-world software production Underlies the Unified Process	
Code-and-fix Model	Fine for short programs that require no maintenance	Totally unsatisfactory for nontrivial programs
Waterfall Model	Disciplined approach Document Driven (quality records for all artifacts)	Delivered product may not meet client's needs
Rapid-prototyping Model	Ensures that the delivered product meets the client's needs	Not yet proved beyond all doubts
Extreme Programming Model	Works well when the customer's requirements are vague	Appears to work on only small-scale project
Synchronize-and-stabilize Model	Future user's needs are met Ensures that components can be successfully integrated	Has not been widely used other than Microsoft
Spiral Model	Risk driven	Can be used only for large-scale, in-house products Developers have to be competent in risk analysis and risk resolution

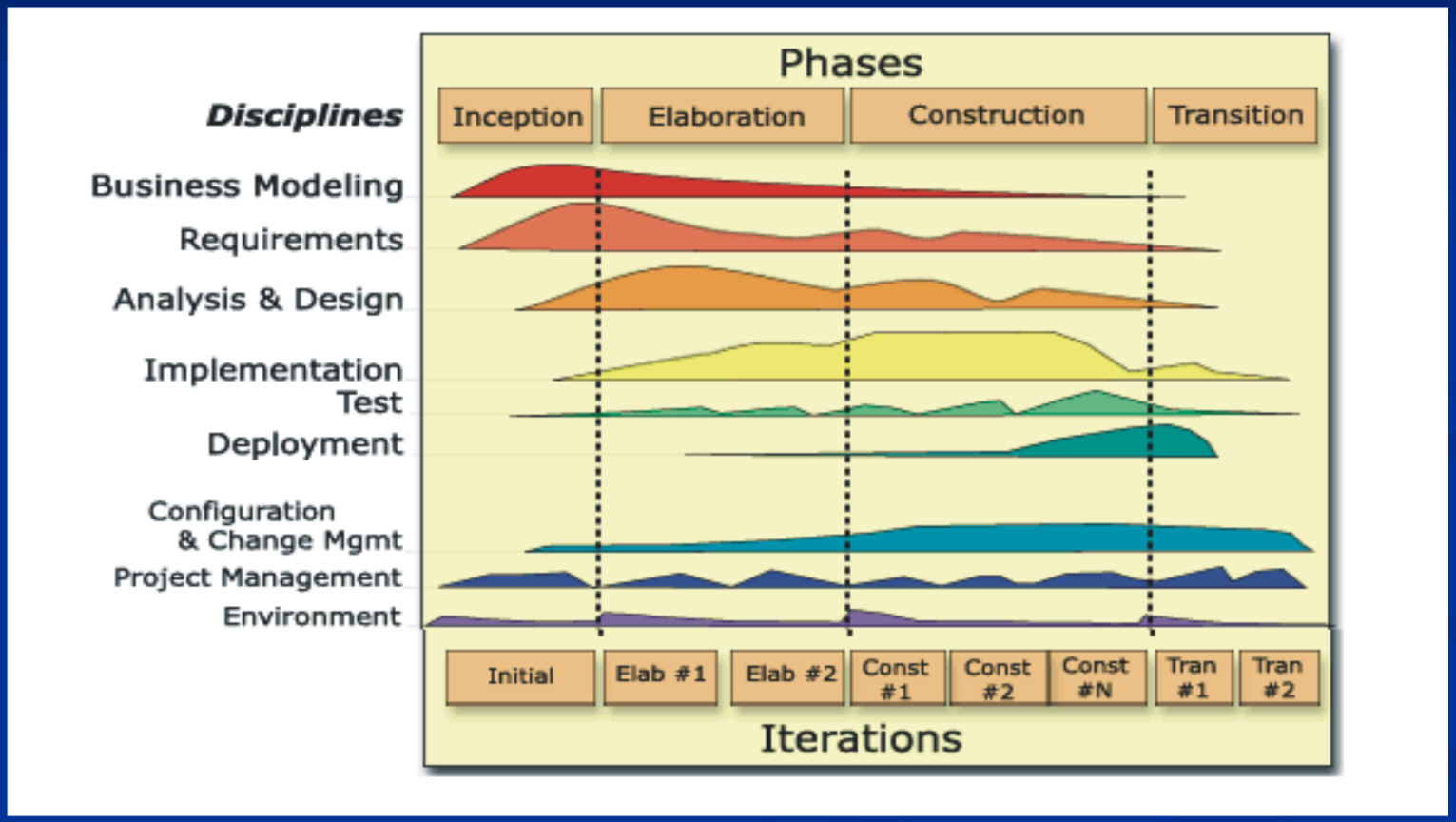
The Rational Unified Process

- A modern process model derived from the work on the UML and associated process.
- Normally described from 3 perspectives
  - A dynamic perspective that shows phases over time;
  - A static perspective that shows process activities;
  - A proactive perspective that suggests good practice.
- Although in theory the development of a software product could be performed in any number of increments, development in practice usually seems to consist of four increments (or phases):
  - Inception
  - Elaboration
  - Construction
  - Transition

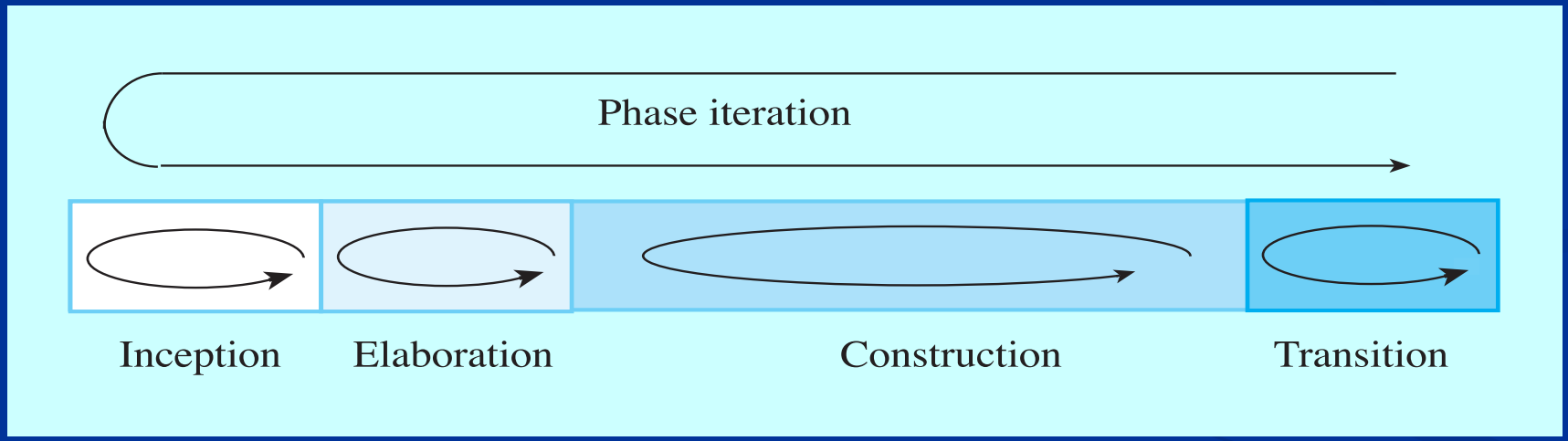
Rational Integration Overview



The Rational Unified Process



RUP phase model



RUP phases

- Inception
  - Establish the business case for the system.
- Elaboration
  - Develop an understanding of the problem domain and the system architecture.
- Construction
  - System design, programming and testing.
- Transition
  - Deploy the system in its operating environment.

RUP good practices

- Develop software iteratively
- Manage requirements
- Use component-based architectures
- Visually model software
- Verify software quality
- Control changes to software

Larman's Process

- Similar to the Rational "Unified Process"
- Provides sequence of steps to take
  - Use cases (and CRC cards)
  - Conceptual Model
  - System Sequence Diagram
  - System Contracts
  - Collaboration Diagram
  - Class Diagram
  - Code

UML:
Unified Modeling Language

Unified Modeling Language (UML)

A general-purpose visual modeling language designed to specify, visualize, construct and document the artifacts of a software system.

- A visual modeling language that depicts the structure of the code at a level just above the code itself.
- Independent of any particular programming language and development process
- With extensibility and specialization mechanisms
- That provides a formal basis for understanding the model

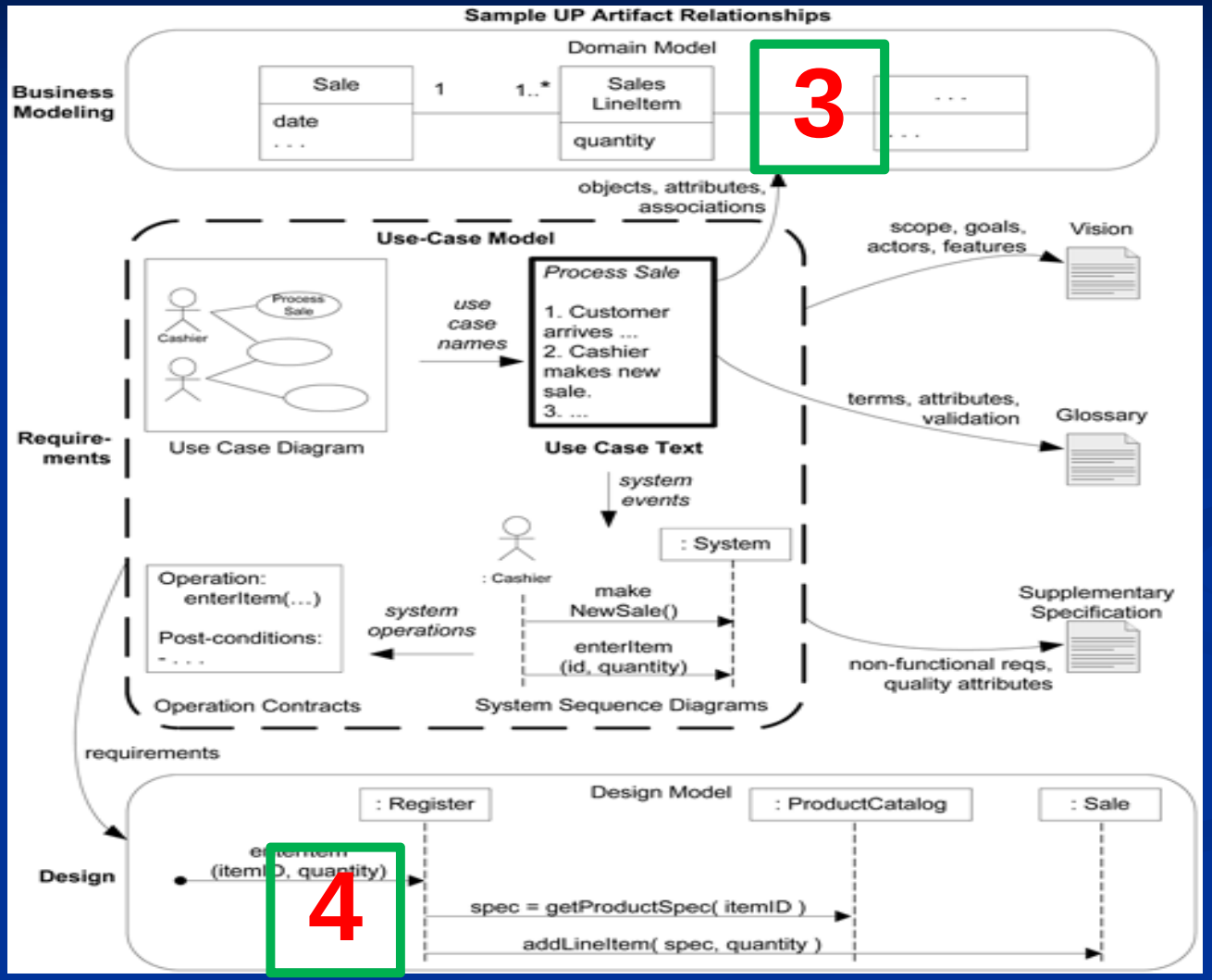
What is the UML

- Three ways to apply UML
  - UML as sketch Informal and incomplete diagrams (often hand sketched on whiteboards) created to explore difficult parts of the problem or solution space, exploiting the power of visual languages.
  - UML as blueprint Relatively detailed design diagrams used either for 1) reverse engineering to visualize and better understanding existing code in UML diagrams, or for 2) code generation (forward engineering).
  - UML as programming language . Complete executable specification of a software system in UML

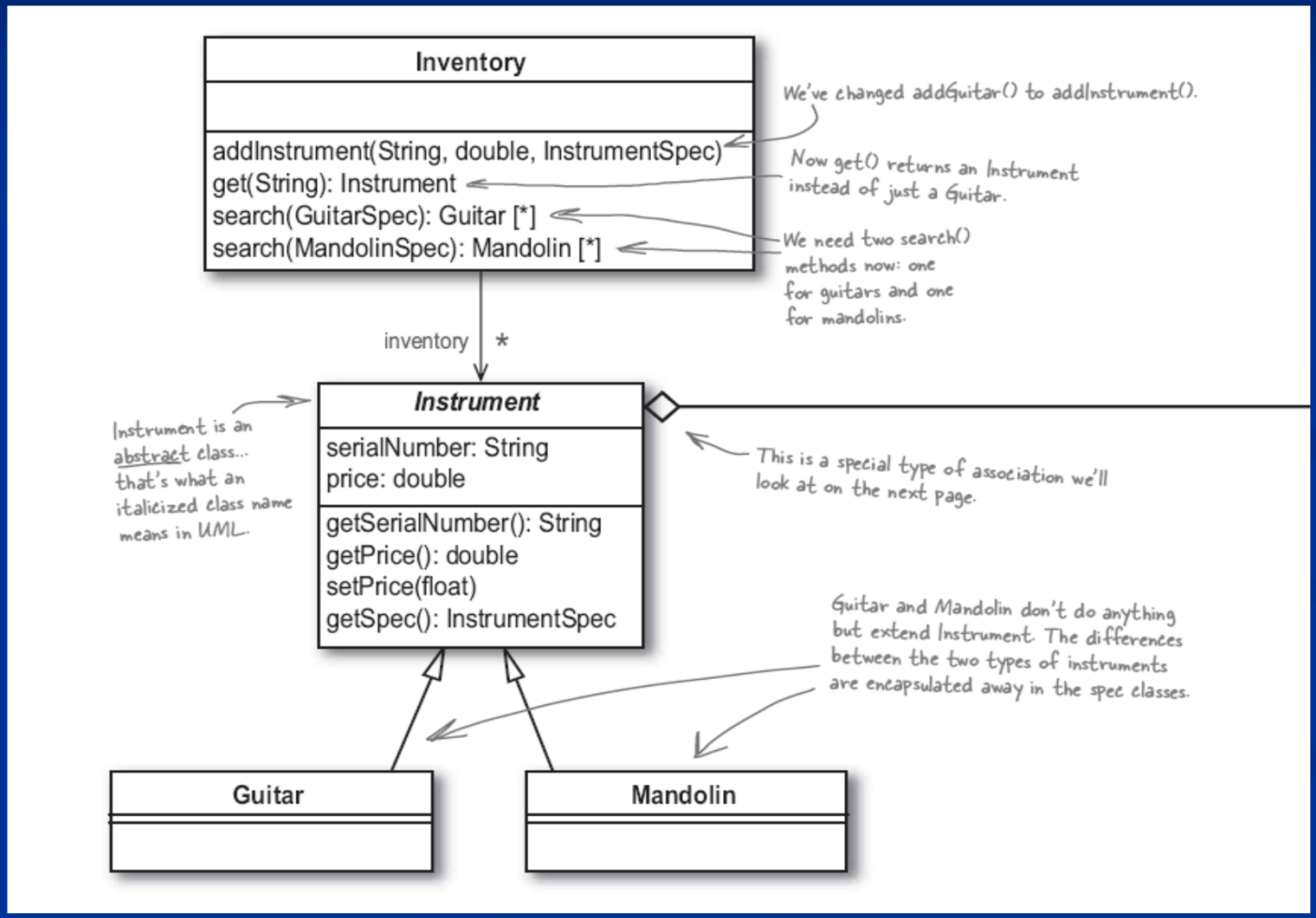
Where to a use/apply UML?

- For us in this class, we will use UML to document and construct the following artifacts:
  - Requirements (Use-Case Model)
  - Analysis (Analysis Model)
  - Design (Design Model)

Where to a use/apply UML?



Unified Modeling Language (UML)



Unified Modeling Language

What does UML contribute?

- A widely-accepted notational system for modeling object-oriented concepts.
- Unified Modeling Language is intended to describe both the real-world and software

Analysis	Design
What	How
Investigate the problem	Create a solution

Unified Modeling Language

UML Cheat Sheet

What we call it in Java	What we call it in UML	How we show it in UML
Abstract Class	Abstract Class	*Italicized Class Name*
Relationship	Association	
Inheritance	Generalization	
Aggregation	Aggregation	

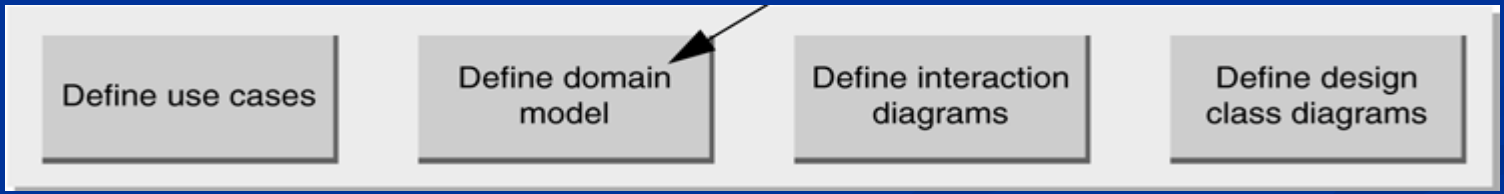
Define Use Cases

- Requirements analysis may include stories or scenarios of how people use the application; these can be written as use cases.

- Use cases are not an object-oriented artifact—they are simply written stories. However, they are a popular tool in requirements analysis. For example, here is a brief version of the Play a Dice Game use case:
  - Play a Dice Game: **Player requests to roll the dice. System presents results: If the dice face value totals seven, player wins; otherwise, player loses.**

Define Domain Model

- Object-oriented analysis is concerned with creating a description of the domain from the perspective of objects. The result can be expressed in a domain model that shows the noteworthy domain concepts or objects.



- Note that a domain model is not a description of software objects; it is a visualization of the concepts or mental models of a real-world domain. Thus, it has also been called a conceptual object model.

Partial domain model of a Dice Game

- This model illustrates the noteworthy concepts Player, Die, and DiceGame, with their associations and attributes.

Assign Object Responsibilities and Draw Interaction Diagrams

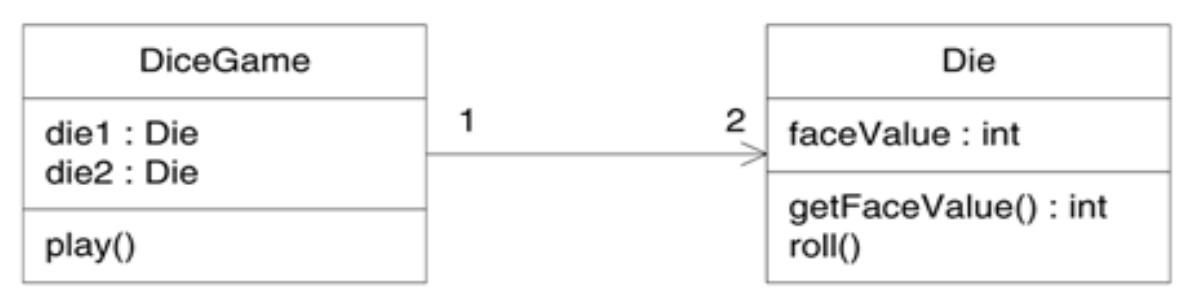
- Object-oriented design is concerned with defining software objects, their responsibilities and collaborations. A common notation to illustrate these collaborations is the sequence diagram (a kind of UML interaction diagram). It shows the flow of messages between software objects, and thus the invocation of methods.

- Software object designs and programs do take some inspiration from real-world domains, but they are not direct models or simulations of the real world.

Define Design Class Diagrams

- A static view of the class definitions is usefully shown with a design class diagram. This illustrates the attributes and methods of the classes.

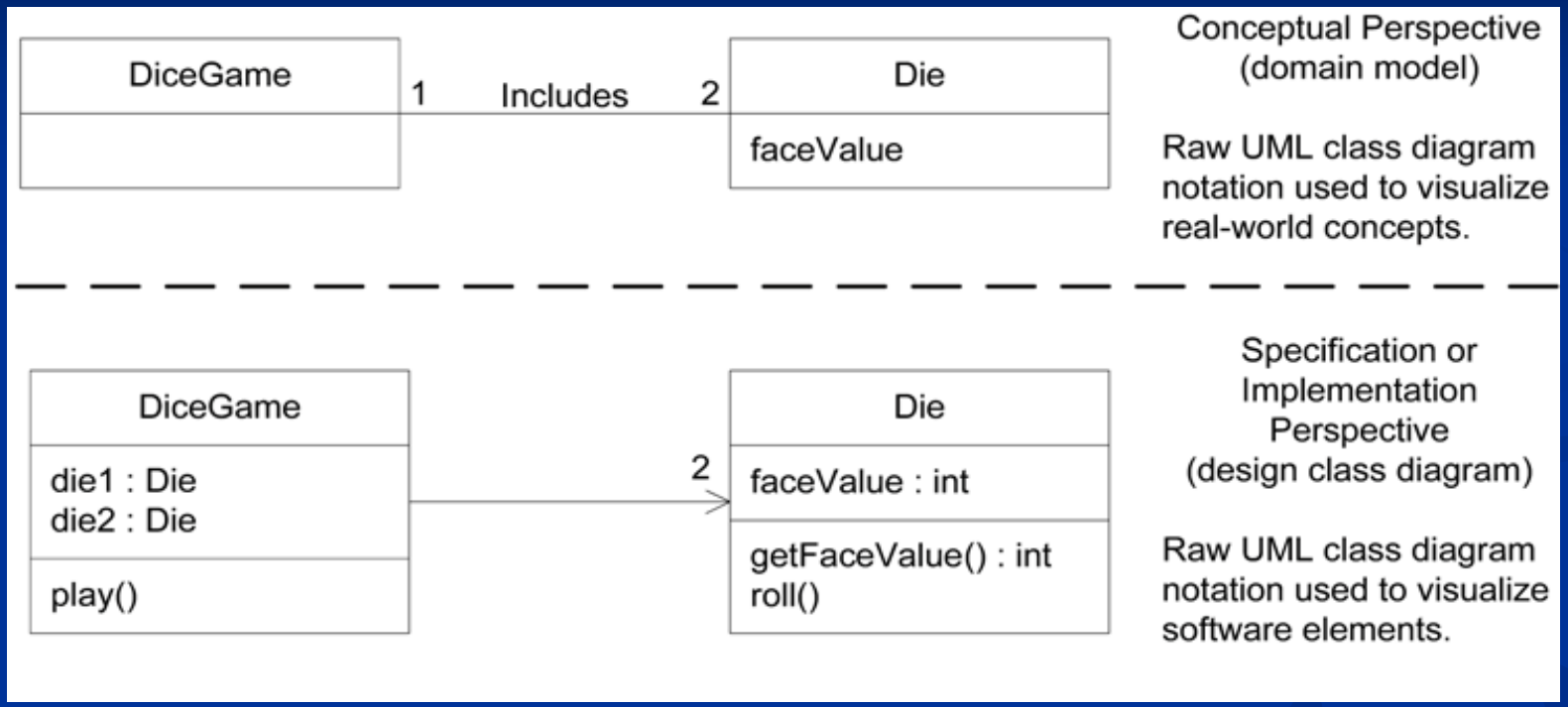
- Partial design class diagram



- In this way OO designs and languages can support a lower representational gap between the software components and our mental models of a domain.

- Three perspectives to apply UML:
  - Conceptual perspective the diagrams are interpreted as describing things in a situation of the real world or domain of interest.
  - Specification (software) perspective the diagrams (using the same notation as in the conceptual perspective) describe software abstractions or components with specifications and interfaces, but no commitment to a particular implementation (for example, not specifically a class in C# or Java).
  - Implementation (software) perspective the diagrams describe software implementations in a particular technology (such as Java).

Different perspectives with UML

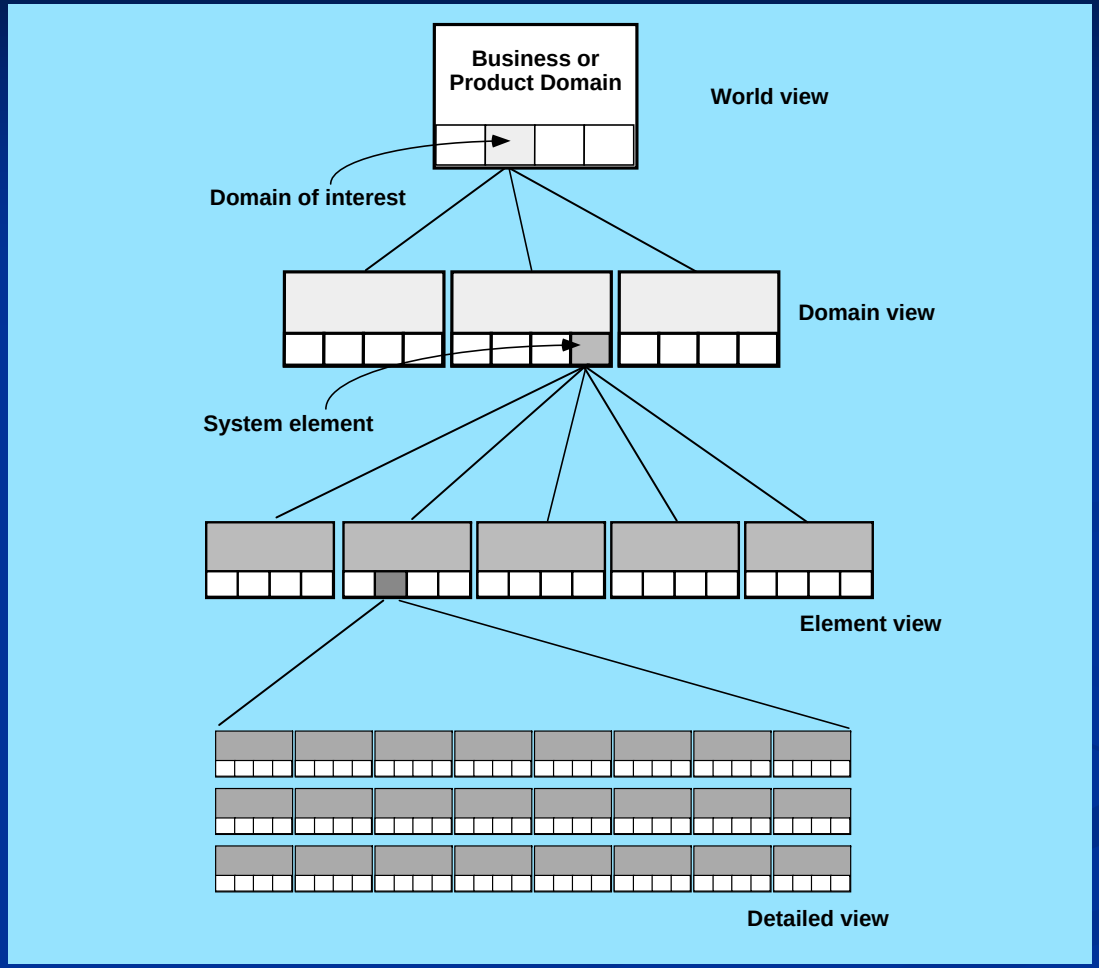


System Engineering

System Engineering

- Elements of a computer-based system
  - Software
  - Hardware
  - People
  - Database
  - Documentation
  - Procedures
- Systems
  - A hierarchy of macro-elements

The Hierarchy



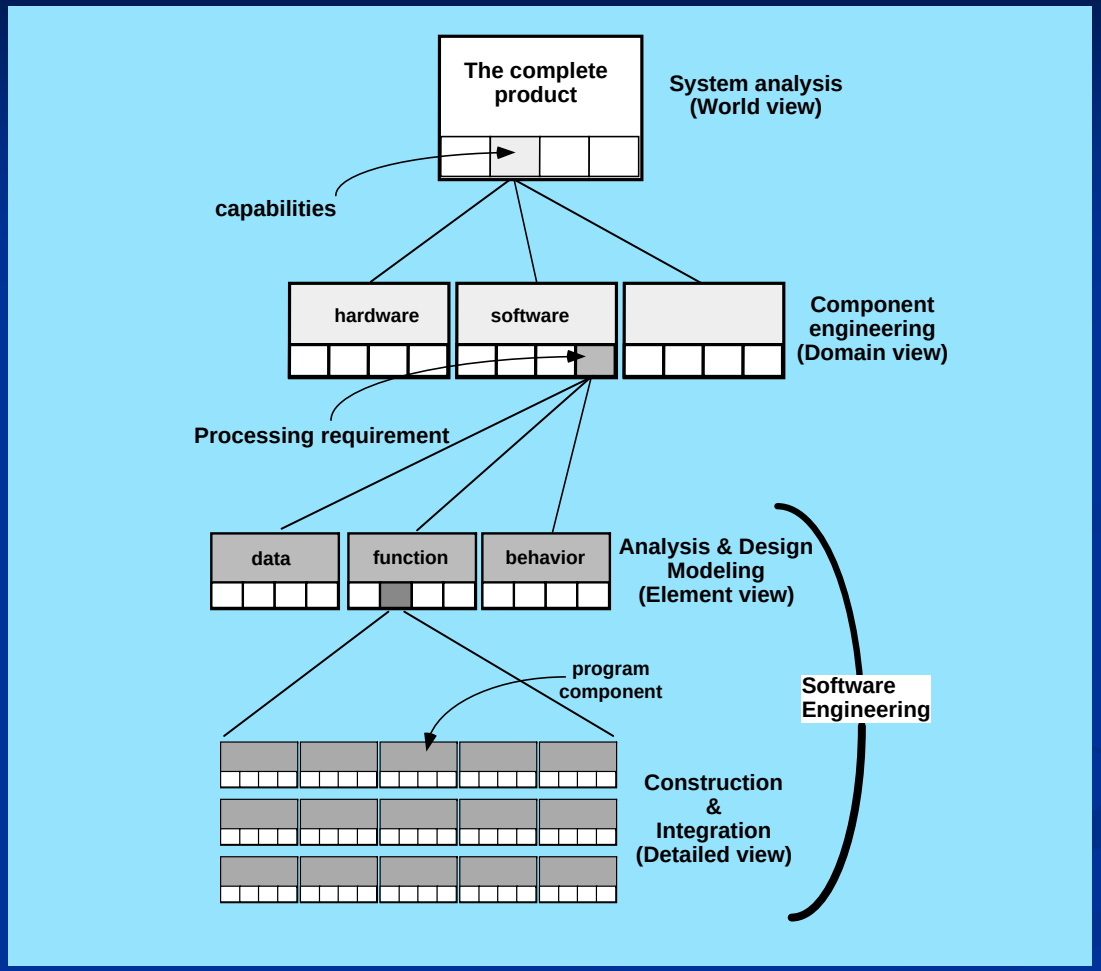
System Modeling

- define the processes that serve the needs of the view under consideration.
- represent the behavior of the processes and the assumptions on which the behavior is based.
- represent all linkages (including output) that will enable the engineer to better understand the view.

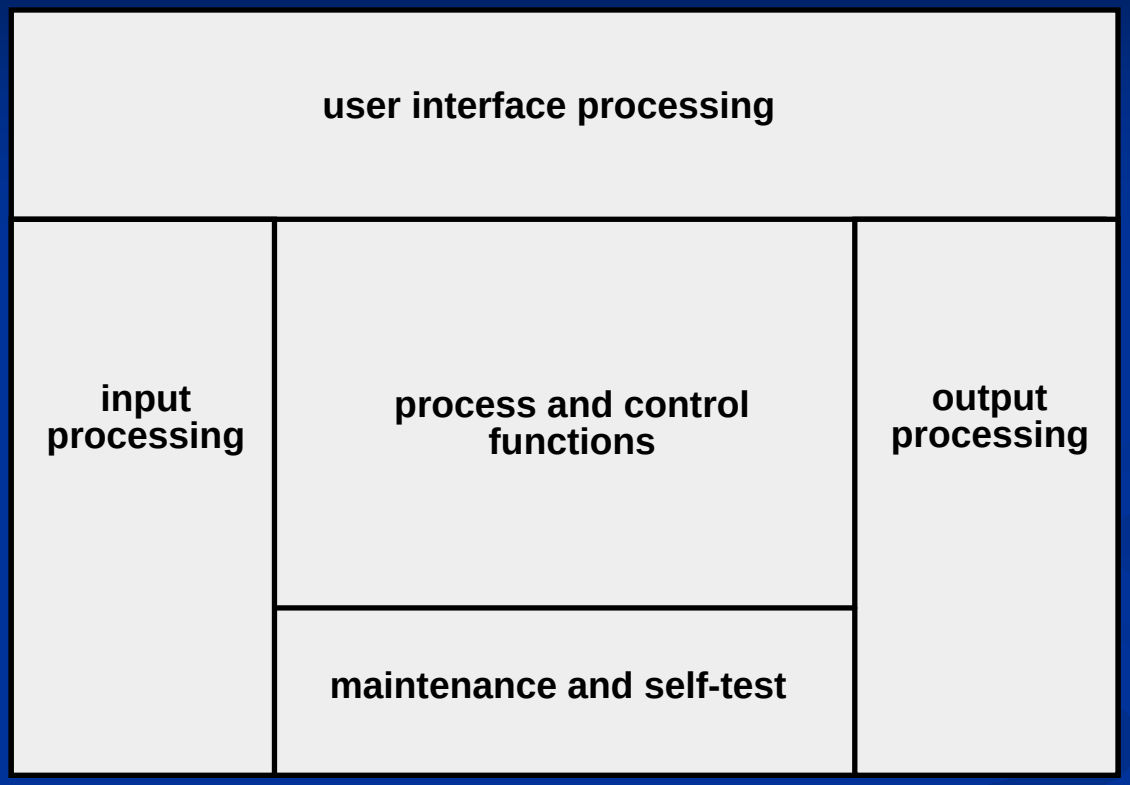
System Architectures

- Three different architectures must be analyzed and designed within the context of business objectives and goals:
  - data architecture
  - applications architecture
  - technology infrastructure
- *data architecture* provides a framework for the information needs of a business or business function
- *application architecture* encompasses those elements of a system that transform objects within the data architecture for some business purpose
- *technology infrastructure* provides the foundation for the data and application architectures

Product Engineering



Product Architecture Template



Requirements Engineering

Requirements Engineering-I

- Requirement Engineering Process is achieved through the execution of the following **7 tasks**:

1. Inception—ask a set of questions that establish …
   - basic understanding of the problem
   - the people who want a solution
   - the nature of the solution that is desired, and
   - the effectiveness of preliminary communication and collaboration between the customer and the developer
2. Elicitation—elicit requirements from all stakeholders
3. Elaboration—create an analysis model that identifies data, function and behavioral requirements
4. Negotiation—agree on a deliverable system that is realistic for developers and customers

Requirements Engineering-II

5. Specification—can be any one (or more) of the following:
   - A written document
   - A set of models
   - A formal mathematical
   - A collection of user scenarios (use-cases)
   - A prototype
6. Validation—a review mechanism that looks for
   - errors in content or interpretation
   - areas where clarification may be required
   - missing information
   - inconsistencies (a major problem when large products or systems are engineered)
   - conflicting or unrealistic (unachievable) requirements.
7. Requirements management

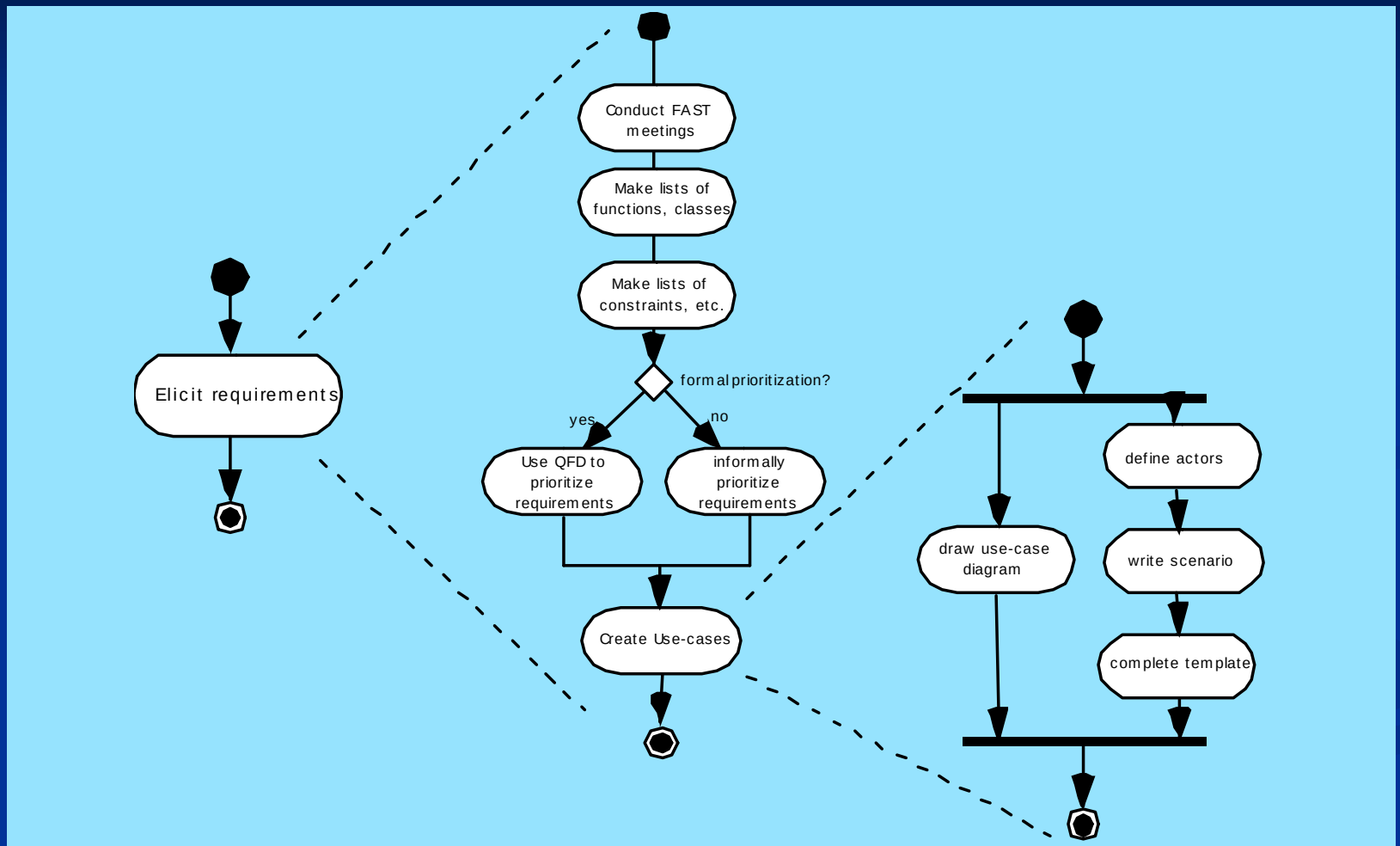
Inception

- Identify stakeholders
  - “who else do you think I should talk to?”
- Recognize multiple points of view
- Work toward collaboration
- The first questions
  - Who is behind the request for this work?
  - Who will use the solution?
  - What will be the economic benefit of a successful solution
  - Is there another source for the solution that you need?

Eliciting Requirements

- meetings are conducted and attended by both software engineers and customers
- rules for preparation and participation are established
- an agenda is suggested
- a "facilitator" (can be a customer, a developer, or an outsider) controls the meeting
- a "definition mechanism" (can be work sheets, flip charts, or wall stickers or an electronic bulletin board, chat room or virtual forum) is used
- the goal is
  - to identify the problem
  - propose elements of the solution
  - negotiate different approaches, and
  - specify a preliminary set of solution requirements

Activity Diagram for Eliciting Requirements



Quality Function Deployment

- Function deployment determines the “value” (as perceived by the customer) of each function required of the system
- Information deployment identifies data objects and events
- Task deployment examines the behavior of the system
- Value analysis determines the relative priority of requirements

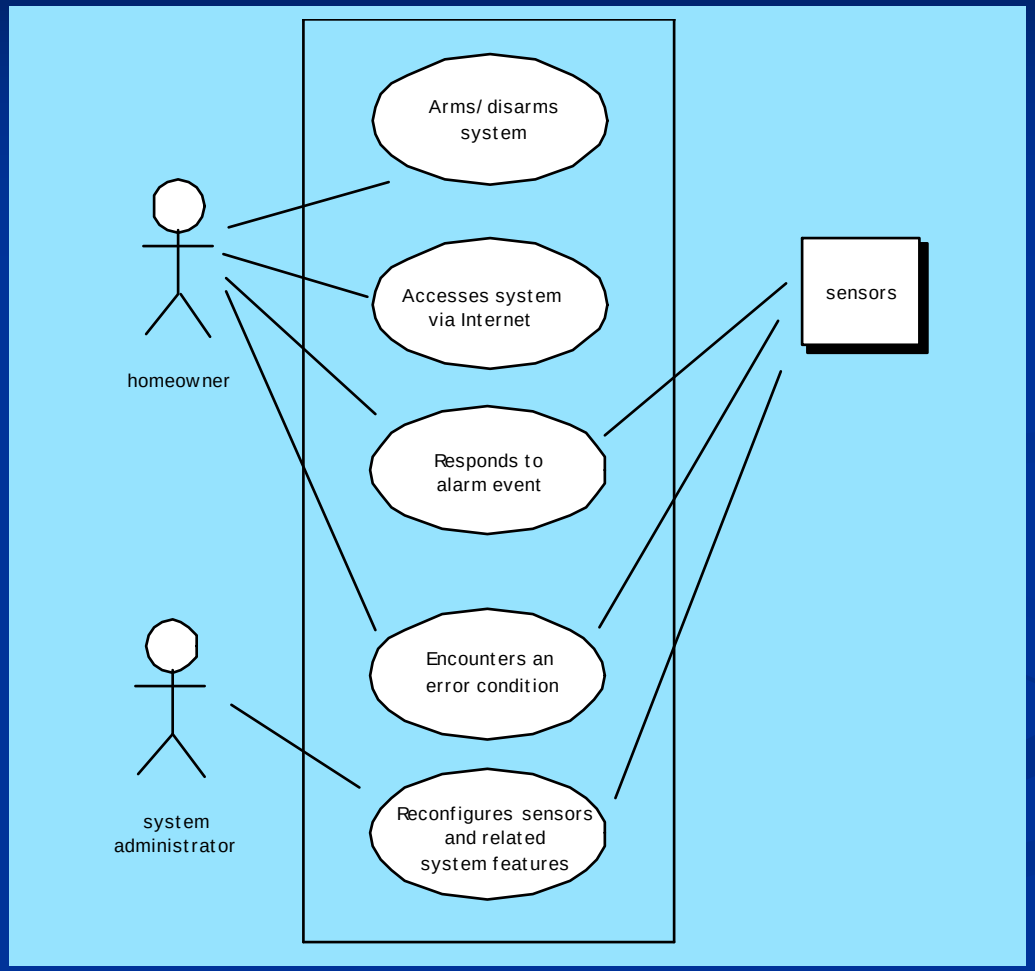
Elicitation Work Products

- The work product resulted from elicitation may include:
  - a statement of need and feasibility.
  - a bounded statement of scope for the system or product.
  - a list of customers, users, and other stakeholders who participated in requirements elicitation
  - a description of the system's technical environment.
  - a list of requirements (preferably organized by function) and the domain constraints that apply to each.
  - a set of usage scenarios that provide insight into the use of the system or product under different operating conditions.
  - any prototypes developed to better define requirements**.**

Use-Cases

- A collection of user scenarios that describe the thread of usage of a system
- Each scenario is described from the point-of-view of an “actor”—a person or device that interacts with the software in some way
- Each scenario answers the following questions:
  - Who is the primary actor, the secondary actor (s)?
  - What are the actor’s goals?
  - What preconditions should exist before the story begins?
  - What main tasks or functions are performed by the actor?
  - What extensions might be considered as the story is described?
  - What variations in the actor’s interaction are possible?
  - What system information will the actor acquire, produce, or change?
  - Will the actor have to inform the system about changes in the external environment?
  - What information does the actor desire from the system?
  - Does the actor wish to be informed about unexpected changes?

Use-Case Diagram for SafeHome Security System

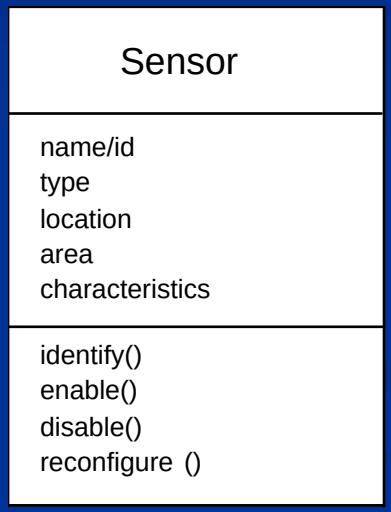


Building the Analysis Model

- Elements of the analysis model
  - Scenario-based elements
    - Functional—processing narratives for software functions
    - Use-case—descriptions of the interaction between an "actor" and the system
  - Class-based elements
    - Implied by scenarios
  - Behavioral elements
    - State diagram
  - Flow-oriented elements
    - Data flow diagram

Class Diagram

From the *SafeHome* system ...



Negotiating Requirements

- Identify the key stakeholders
  - These are the people who will be involved in the negotiation
- Determine each of the stakeholders “win conditions”
  - Win conditions are not always obvious
- Negotiate
  - Work toward a set of requirements that lead to “win-win”

Validating Requirements-I

- Is each requirement consistent with the overall objective for the system/product?
- Have all requirements been specified at the proper level of abstraction? That is, do some requirements provide a level of technical detail that is inappropriate at this stage?
- Is the requirement really necessary or does it represent an add-on feature that may not be essential to the objective of the system?
- Is each requirement bounded and unambiguous?
- Does each requirement have attribution? That is, is a source (generally, a specific individual) noted for each requirement?
- Do any requirements conflict with other requirements?

Validating Requirements-II

- Is each requirement achievable in the technical environment that will house the system or product?
- Is each requirement testable, once implemented?
- Does the requirements model properly reflect the information, function and behavior of the system to be built.
- Has the requirements model been “partitioned” in a way that exposes progressively more detailed information about the system.
- Have requirements patterns been used to simplify the requirements model. Have all patterns been properly validated? Are all patterns consistent with customer requirements?