Dealing with Complexity

Lecture 2

Dealing with Complexity

Three ways to deal with complexity

1. Abstraction and Modeling
2. Decomposition
3. Hierarchy

Introduction to the UML notation
Software Lifecycle Modeling

Abstraction

- Complex systems are hard to understand
  - The 7 ± 2 phenomena
    - Our short term memory cannot store more than 7 ± 2 pieces at the same time -> limitation of the brain
    - Phone Number: 042175052

Abstraction

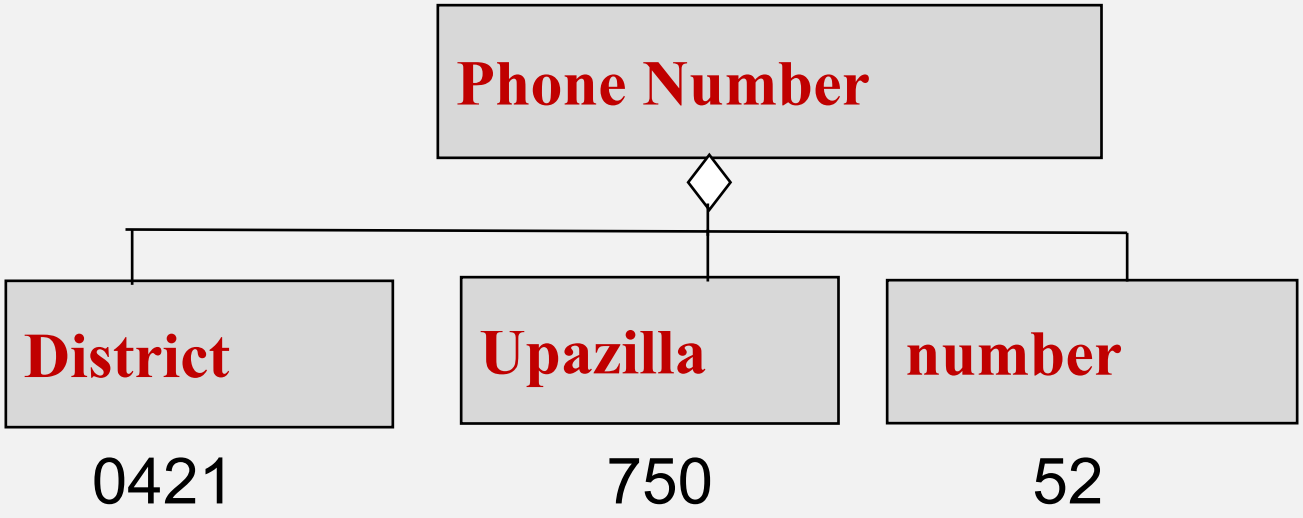
- Complex systems are hard to understand
  - The 7 ± 2 phenomena
    - Our short term memory cannot store more than 7 ± 2 pieces at the same time -> limitation of the brain
    - Phone Number: 0421750 52

- Chunking:
  - Group collection of objects to reduce complexity
  - 3 chunks:
    - District, Upazilla, number

Abstraction

- Complex systems are hard to understand
  - The 7 ± 2 phenomena
    - Our short term memory cannot store more than 7 ± 2 pieces at the same time -> limitation of the brain
    - Phone Number: 042175052
- Chunking:
    - Group collection of objects to reduce complexity

District, Upazilla, number



Abstraction

- Abstraction allows us to ignore unessential details
- Two definitions for abstraction:
  - Abstraction is a *thought process* where ideas are distanced from objects
    - Abstraction as activity

- Ideas can be expressed by models

Model

A model is an abstraction of a system

- A system that no longer exists
- An existing system
- A future system to be built.

We use Models to describe Software Systems

Functional model: What are the functions of the system?

Object model: What is the structure of the system?

Dynamic model: How does the system react to external events?

System Model::
functional model
+
Object model
+
dynamic model

Other models used to describe Software System Development

Task Model:

- PERT Chart: What are the dependencies between tasks?
- Schedule: How can this be done within the time limit?
- Organization Chart: What are the roles in the project?

Issues Model:

- What are the open and closed issues?
- What constraints were imposed by the client?
  - These lead to action items

2. Technique to deal with Complexity: Decomposition

A technique used to master complexity (“divide and conquer”)

Two major types of decomposition

1. Functional decomposition
2. Object-oriented decomposition

Decomposition (cont'd)

Functional decomposition

- The system is decomposed into modules
- Each module is a major function in the application domain
- Modules can be decomposed into smaller modules.

Object-oriented decomposition

- The system is decomposed into classes ("objects")
- Each class is a major entity in the application domain
- Classes can be decomposed into smaller classes

Functional Decomposition

The functionality is spread all over the system

Maintainer must understand the whole system to make a single change to the system

Consequence:

- Source code is hard to understand
- Source code is complex and impossible to maintain
- User interface is often awkward and non-intuitive.

Functional Decomposition

The functionality is spread all over the system

Maintainer must understand the whole system to make a single change to the system

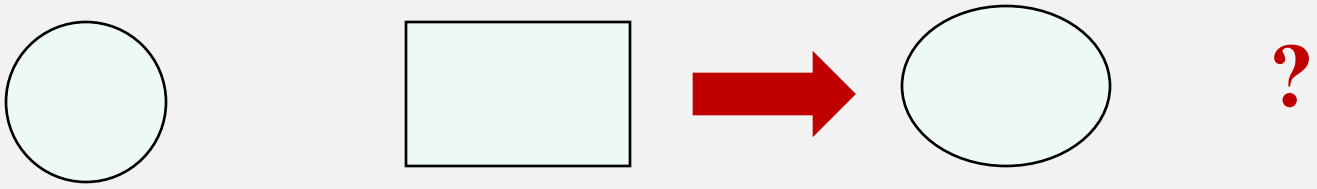
Consequence:

Source code is hard to understand

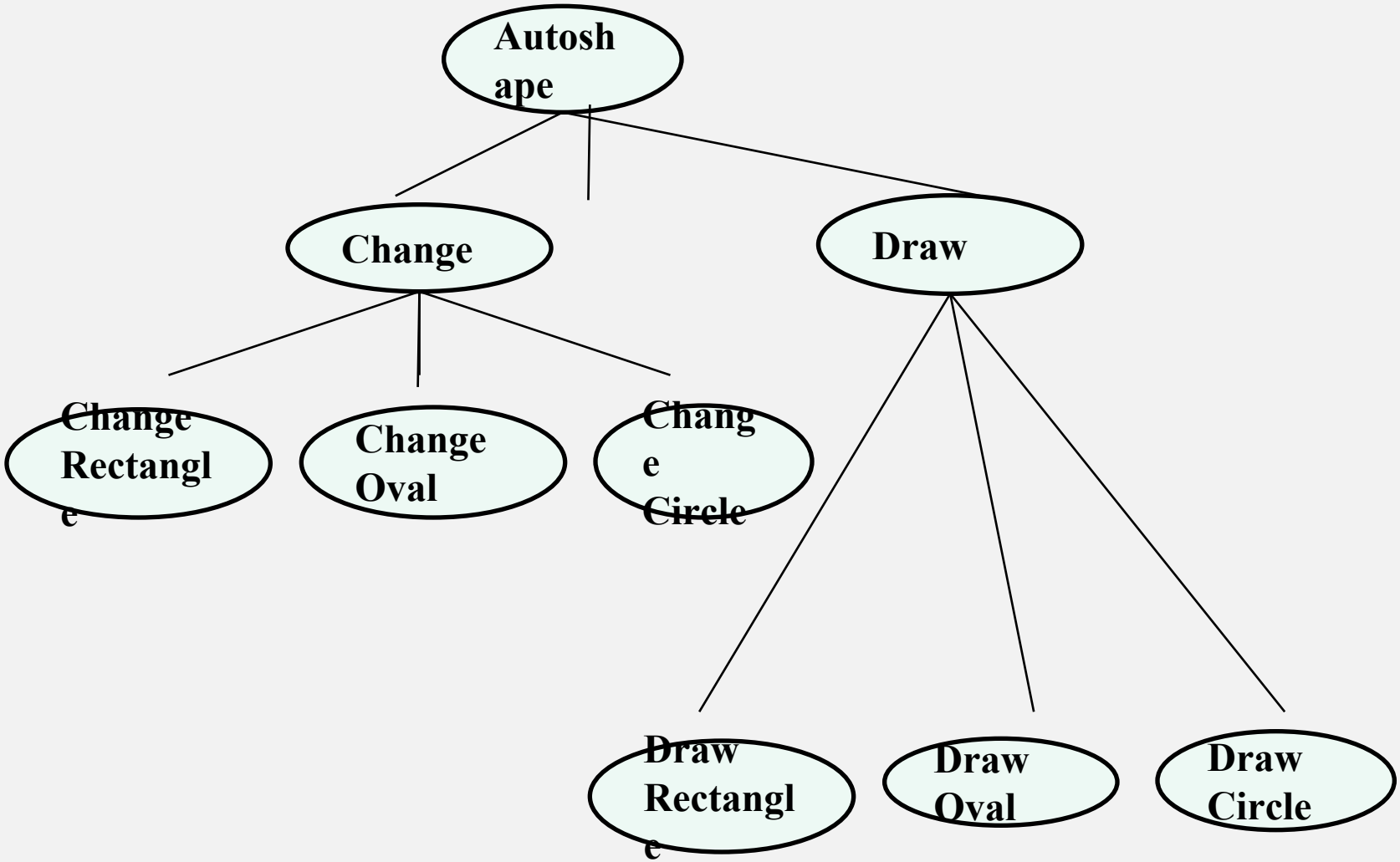
Source code is complex and impossible to maintain

User interface is often awkward and non-intuitive.

- Example: Microsoft Powerpoint's Autoshapes
  - How do I change a square into a circle?



Functional Decomposition: Autoshape



Object Oriented Decomposition

Class Identification

Basic assumptions:

- We can find the *classes for a new software system:* Greenfield Engineering
- We can identify the *classes in an existing system*: Reengineering
- We can create a *class-based interface to an existing system:* Interface Engineering

Class Identification (cont'd)

Why can we do this?

Philosophy, science, experimental evidence

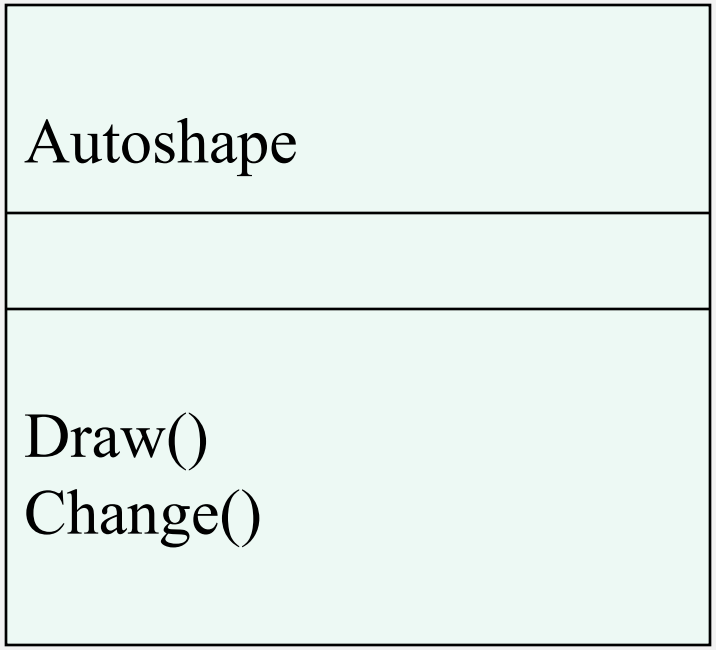
What are the limitations?

Depending on the purpose of the system, different objects might be found

Crucial

Identify the purpose of a system

Object-Oriented View



3. Hierarchy

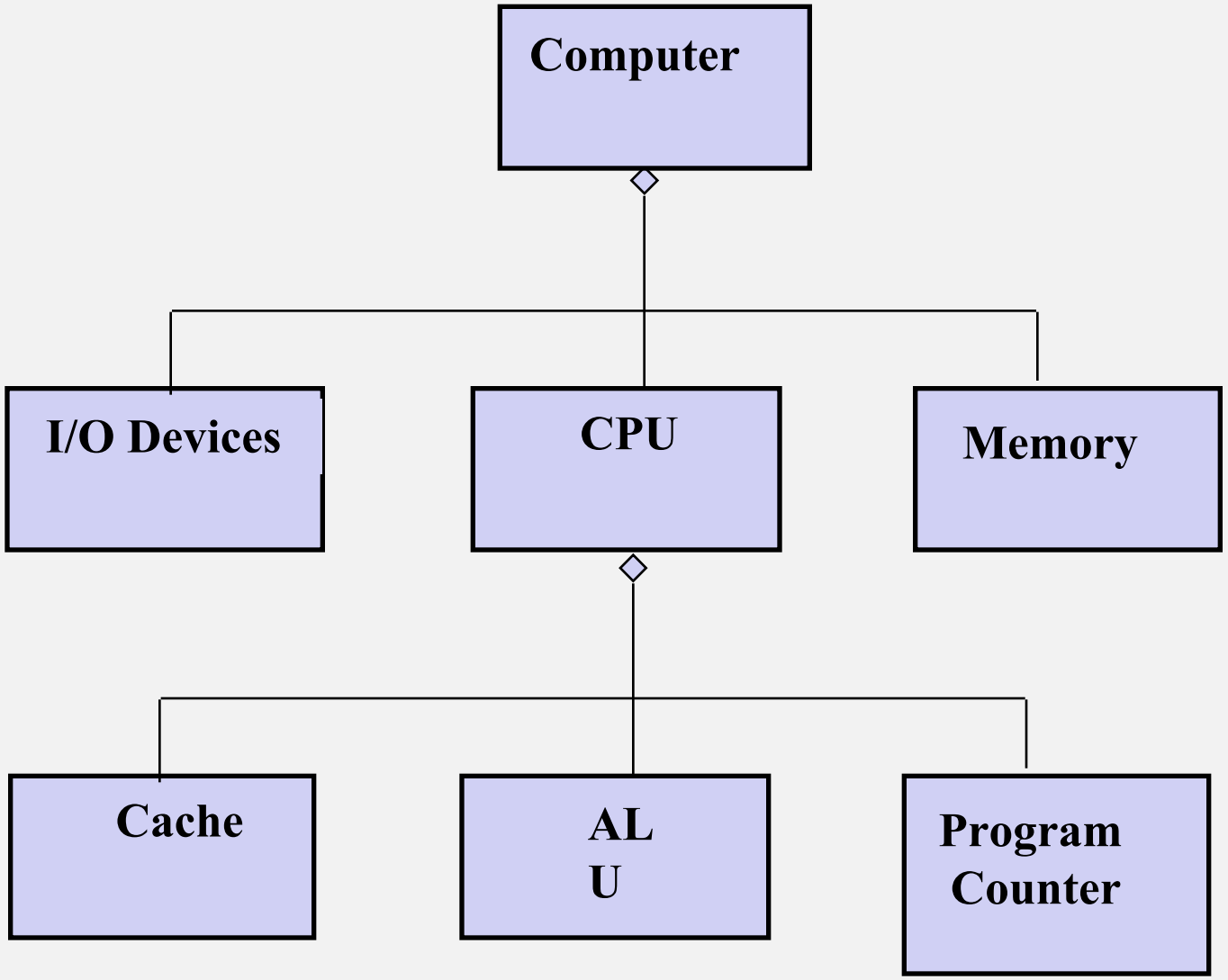
So far we got abstractions

This leads us to classes and objects

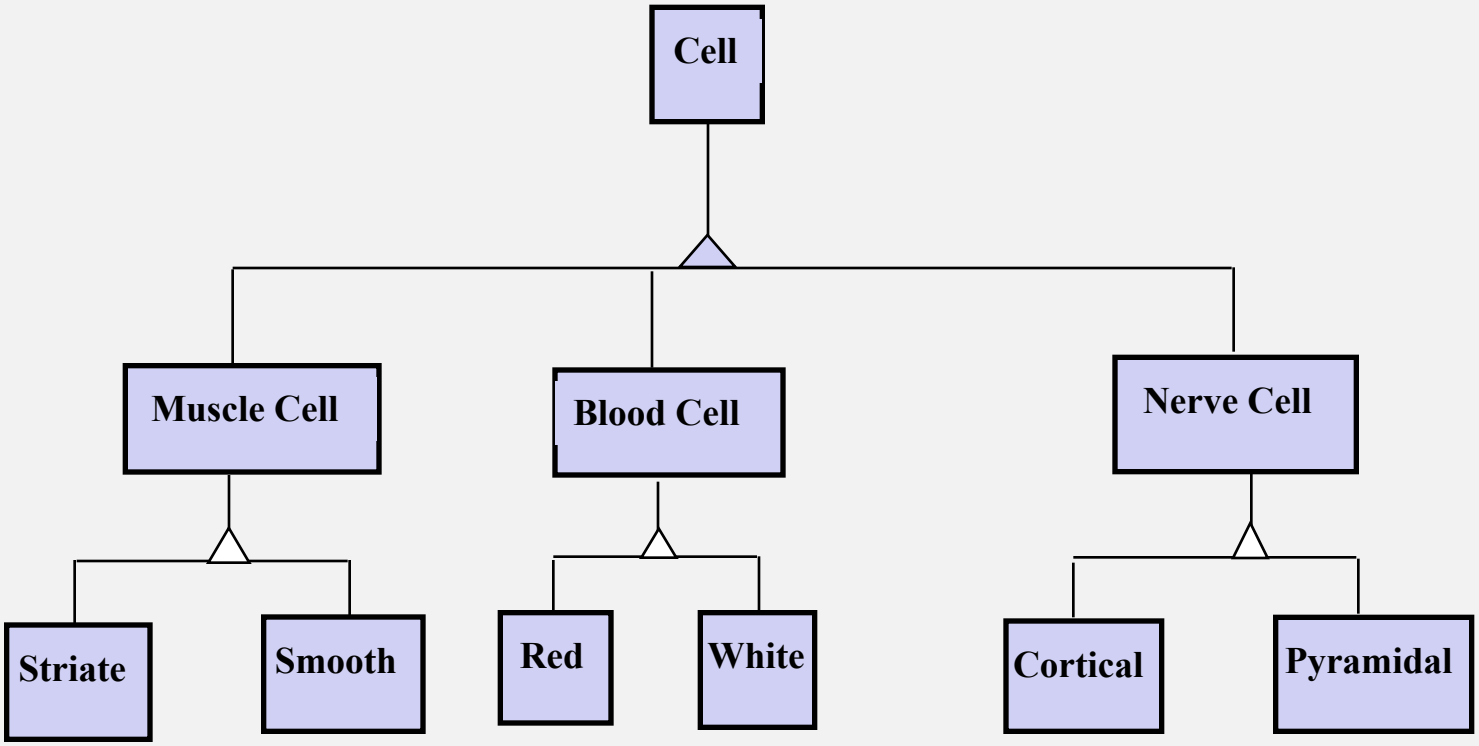
“Chunks”

- Another way to deal with complexity is to provide relationships between these chunks
- One of the most important relationships is hierarchy
- 2 special hierarchies
  - "Part-of" hierarchy
  - "Is-kind-of" hierarchy

Part-of Hierarchy (Aggregation)



Is-Kind-of Hierarchy (Taxonomy)



Where are we now?

Three ways to deal with complexity:

- Abstraction, Decomposition, Hierarchy

Object-oriented decomposition is good

- Unfortunately, depending on the purpose of the system, different objects can be found

How can we do it right?

- Start with a description of the functionality of a system
- Then proceed to a description of its structure

Ordering of development activities

- **Software lifecycle**

Typical Software Life Cycle Questions

Which activities should we select for the software project?

What are the *dependencies between activities*?

How should we *schedule the activities*?

To find these activities and dependencies we can use the same modeling techniques we use for software development:

Functional Modeling of a Software Lifecycle

- Scenarios
- Use case model

Structural modeling of a Software Lifecycle

- Object identification
- Class diagrams

Dynamic Modeling of a Software Lifecycle

- Sequence diagrams, statechart and activity diagrams

Unified Modeling Language

UML (Unified Modeling Language)

- Nonproprietary standard for modeling software systems
- Convergence of notations used in object-oriented methods

Current Version: UML 2.5.1

- Information at the OMG portal http://www.uml.org/

Open Source tools: ArgoUML, StarUML, Umbrello

Unified Modeling Language

You can model 80% of most problems by using about 20 % UML

We teach you those 20%

Unified Modeling Language

Use case diagrams

- Describe the functional behavior of the system as seen by the user

Class diagrams

- Describe the static structure of the system: Objects, attributes, associations

Sequence diagrams

- Describe the dynamic behavior between objects of the system

Statechart diagrams

- Describe the dynamic behavior of an individual object

Activity diagrams

- Describe the dynamic behavior of a system, in particular the workflow.

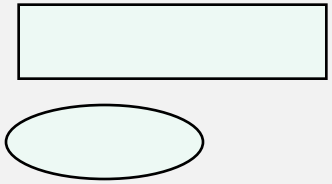
UML Core Conventions

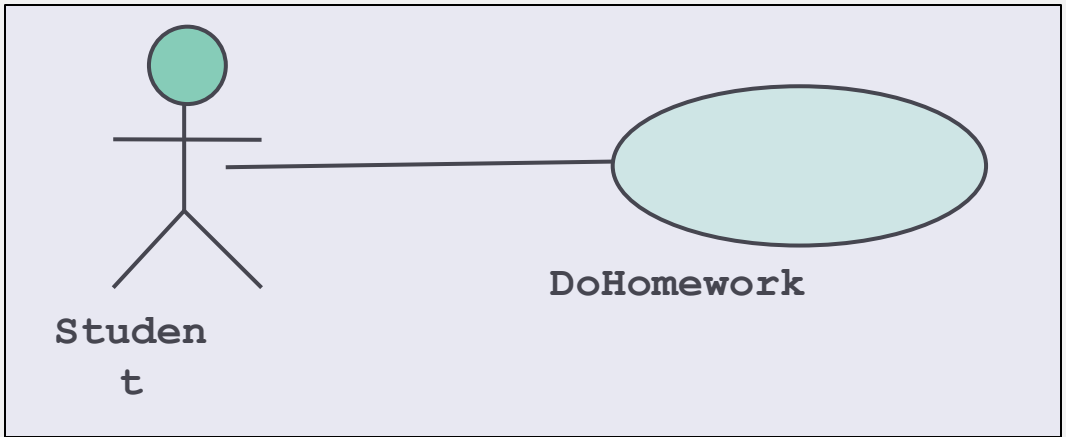
All UML Diagrams denote graphs of nodes and edges

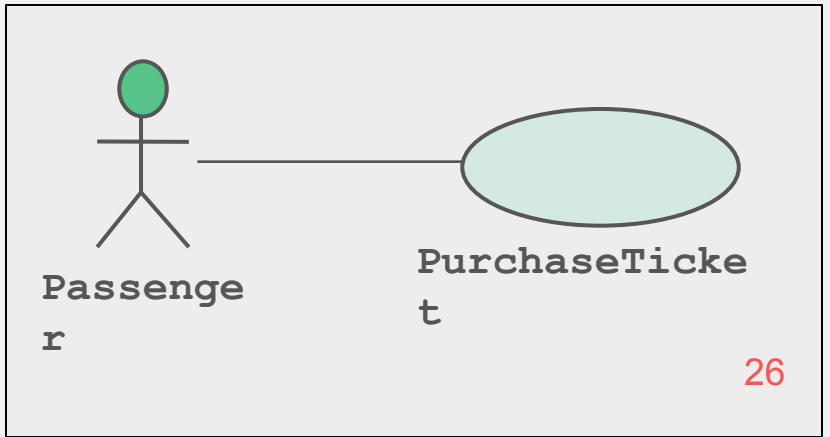
Nodes are entities and drawn as rectangles or ovals

Rectangles denote classes or instances

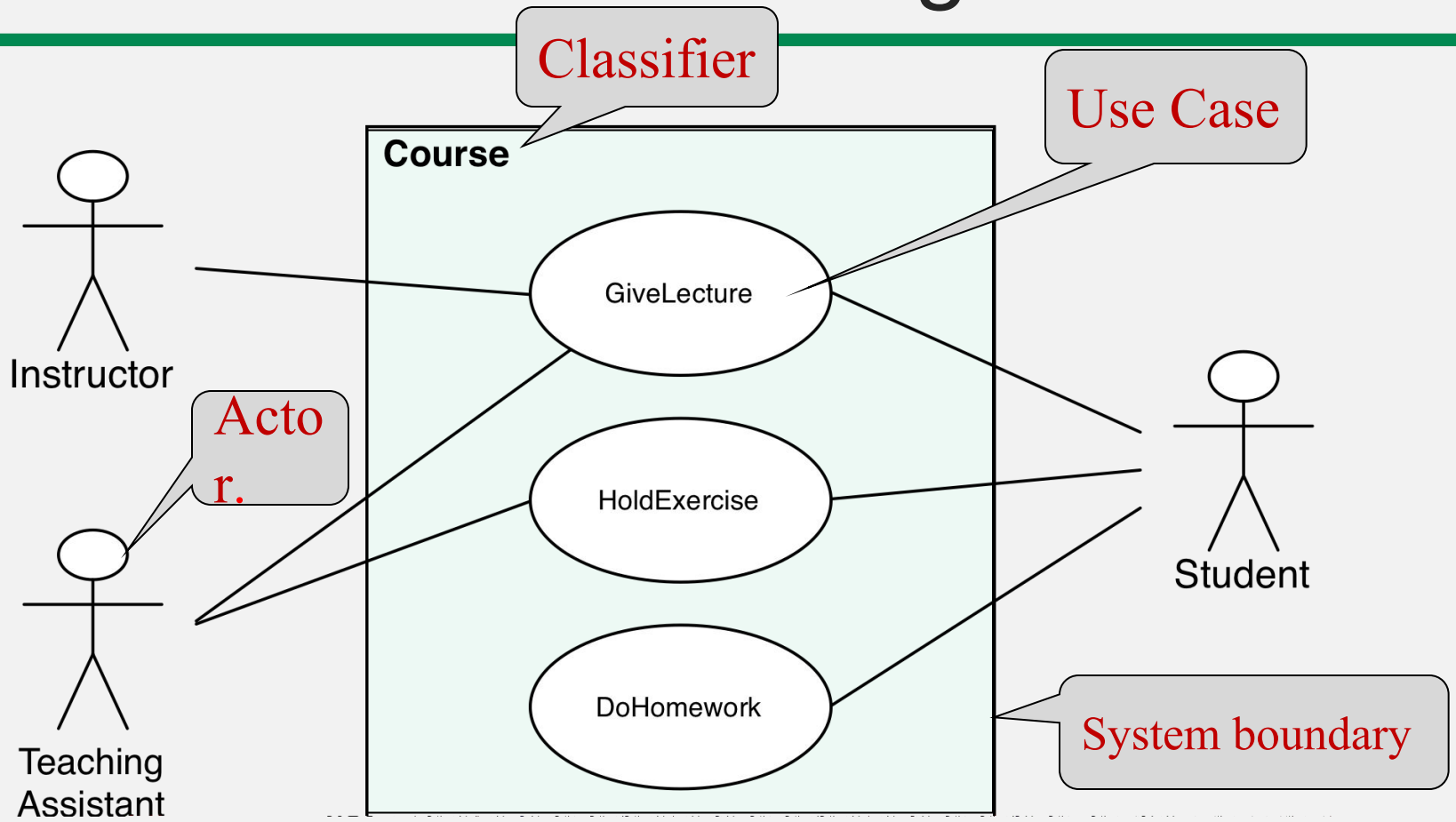
Ovals denote functions





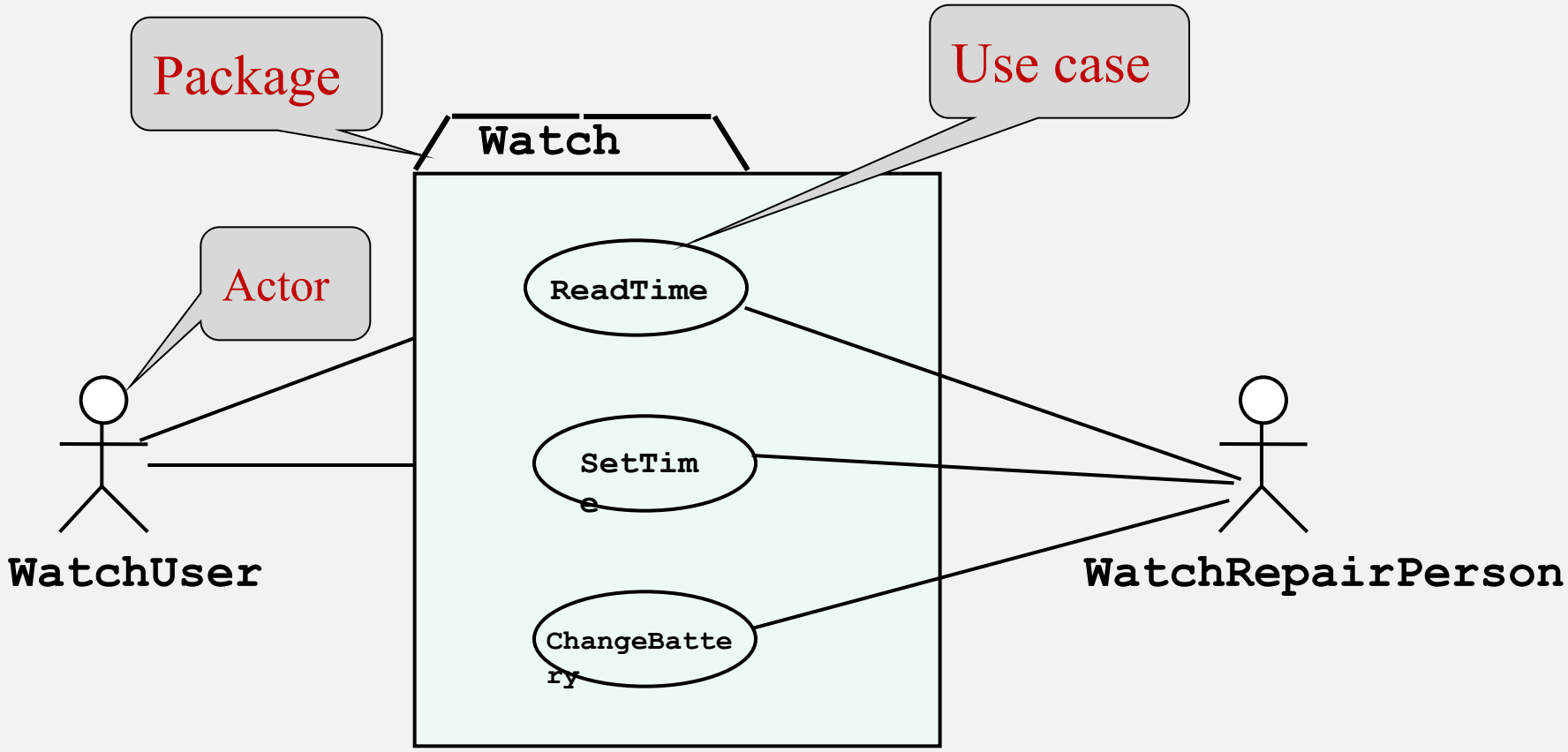


UML: Use case diagrams



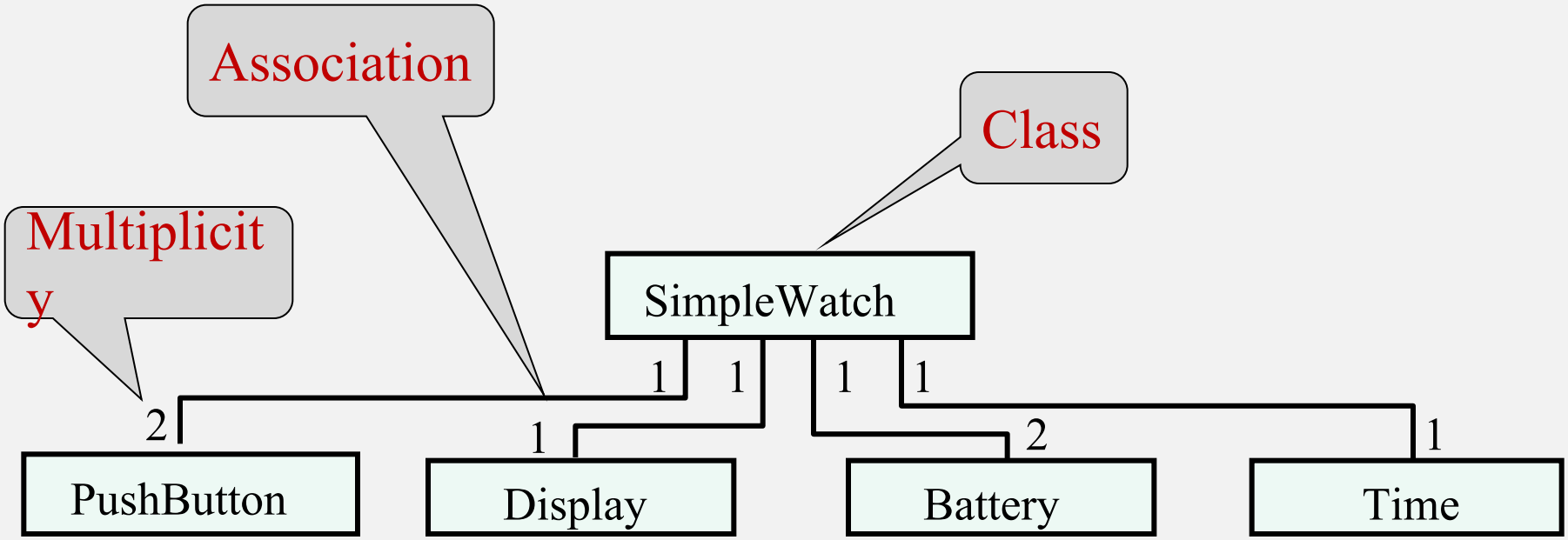
Use case diagrams represent the functionality of the system from user's point of view

Historical Remark: UML 1 used packages



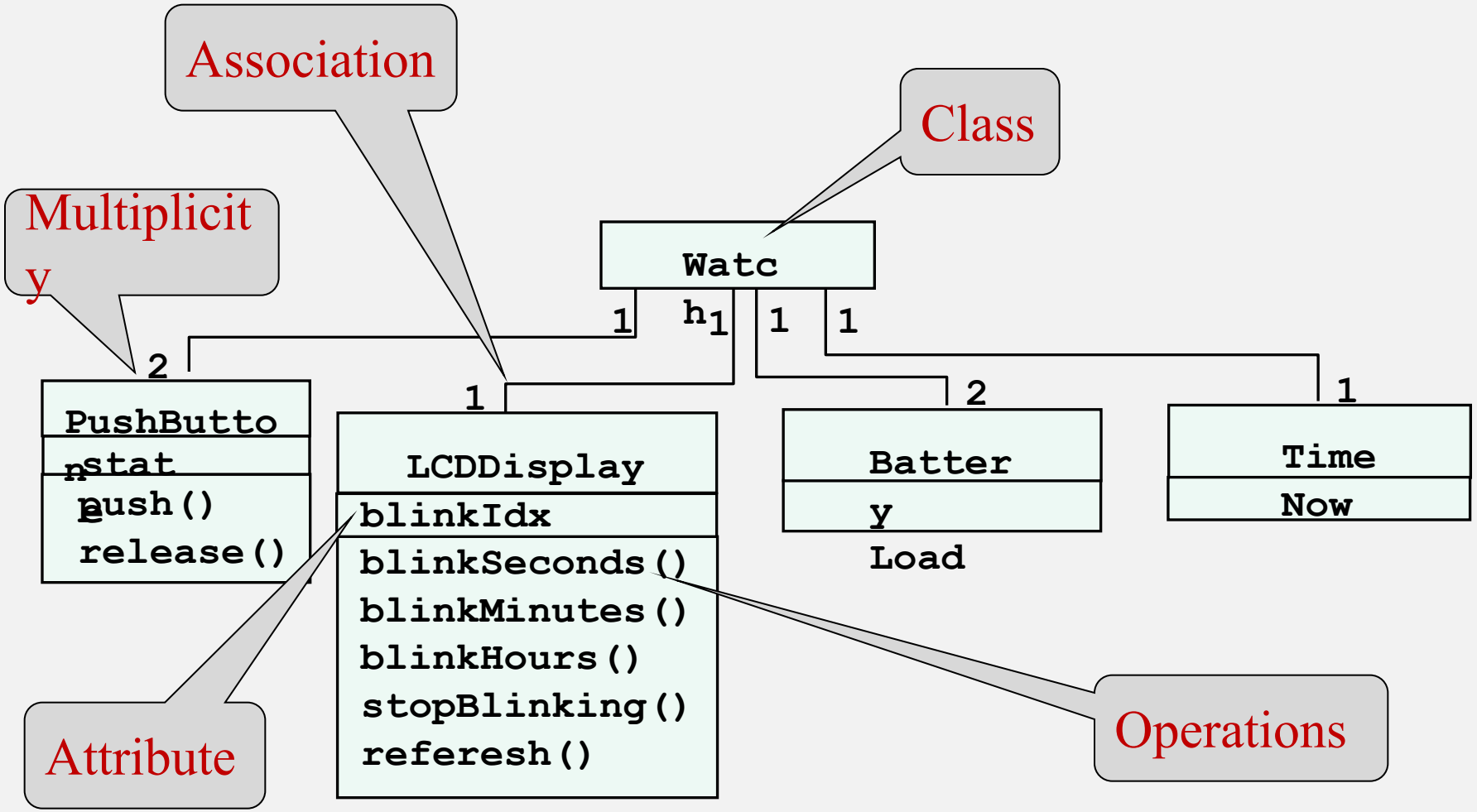
Use case diagrams represent the functionality of the system from user's point of view

UML: Class diagrams



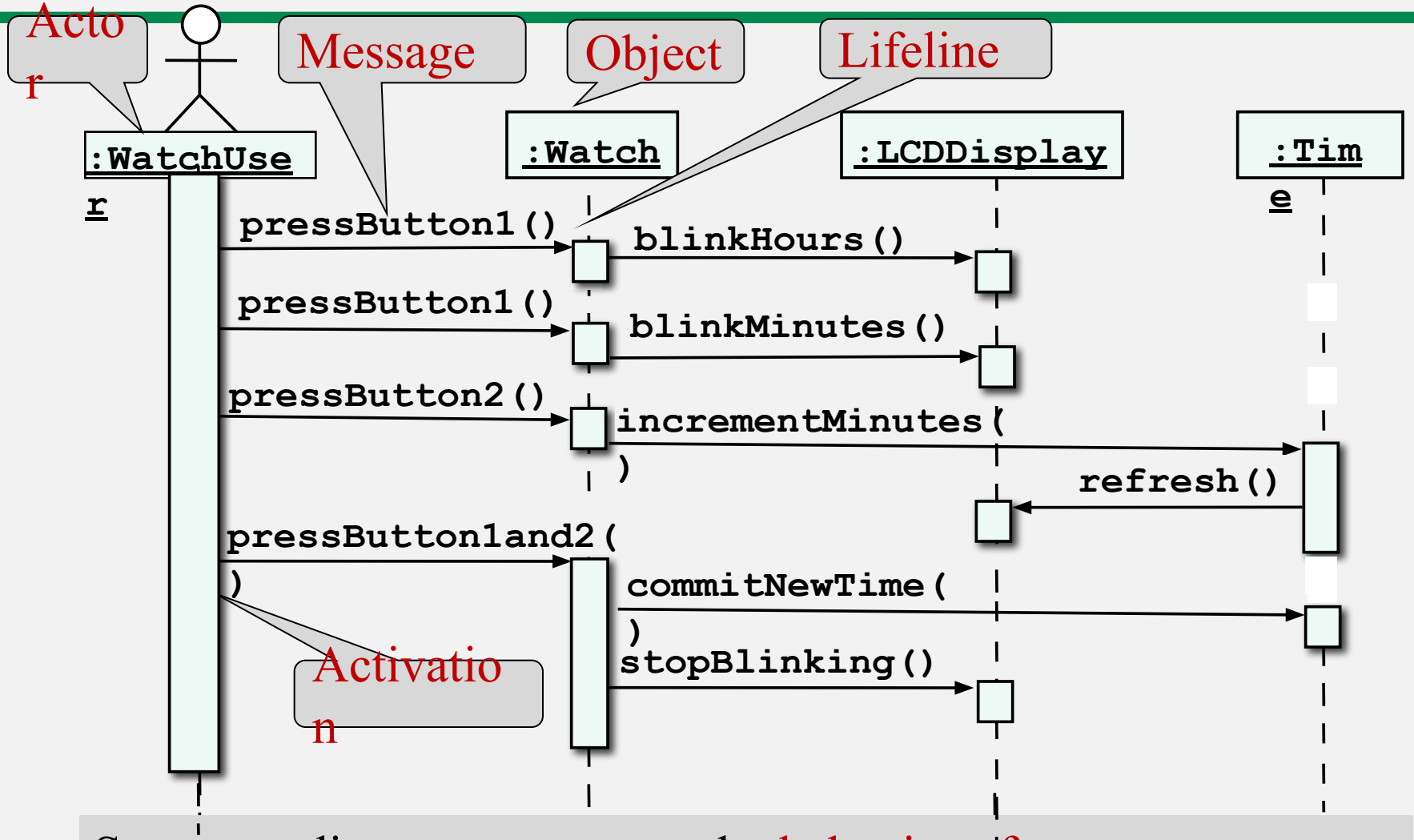
Class diagrams represent the structure of the system

UML first pass: Class diagrams



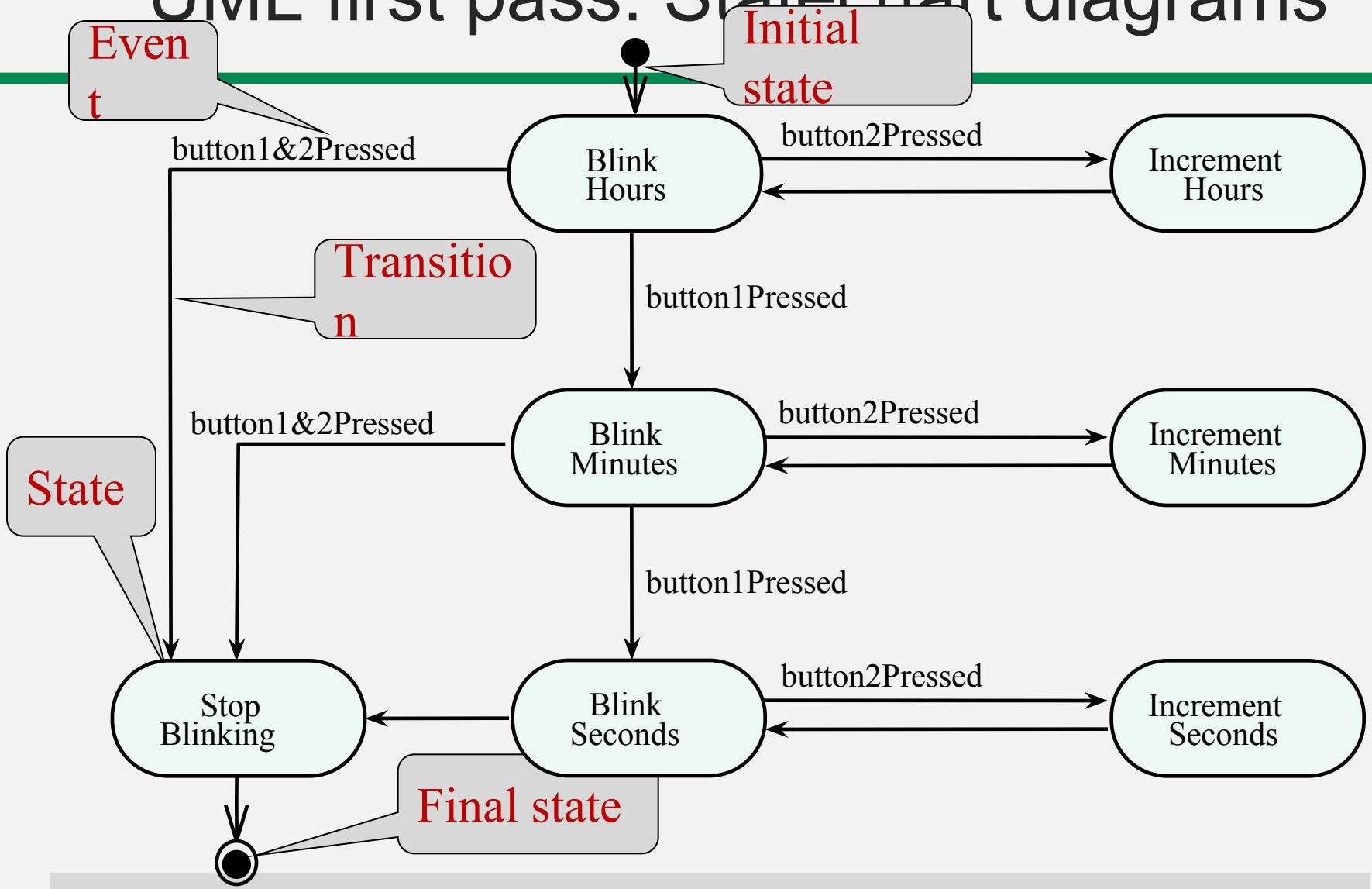
Class diagrams represent the structure of the system

UML: Sequence diagram



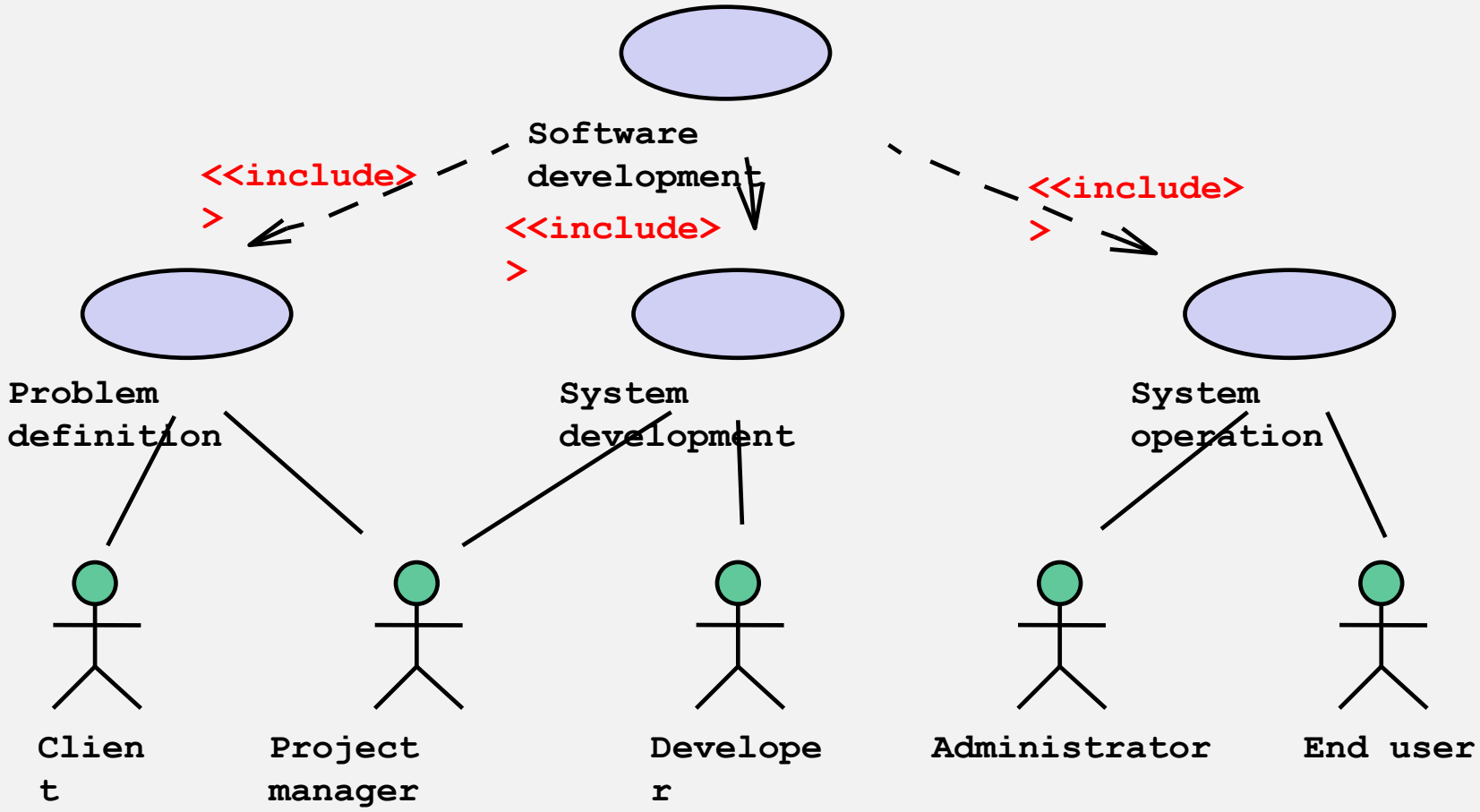
Sequence diagrams represent the behavior of a system as messages ("interactions") between *different objects*

UML first pass: Statechart diagrams



Represent behavior of *a single object* with interesting dynamic behavior.

Functional Model of a simple life cycle model



Activity Diagram for t he same Life Cycle Model

Software development goes through a linear progression of states called software development activities

Two Major Views of the Software Life Cycle

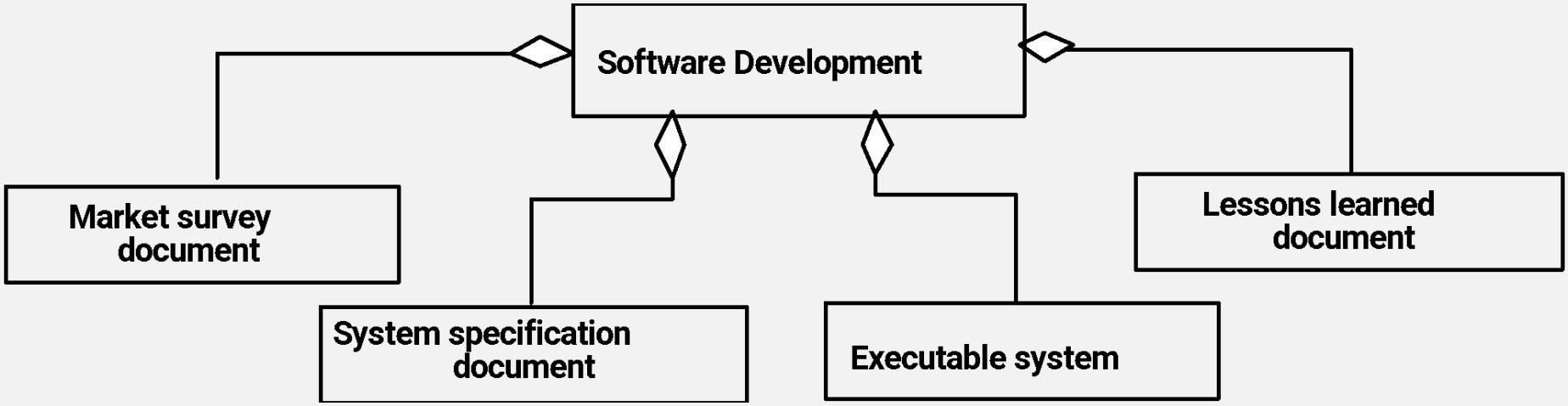
Activity-oriented view of a software life cycle

- Software development consists of a set of development activities
- all the examples so far

Entity-oriented view of a software life cycle

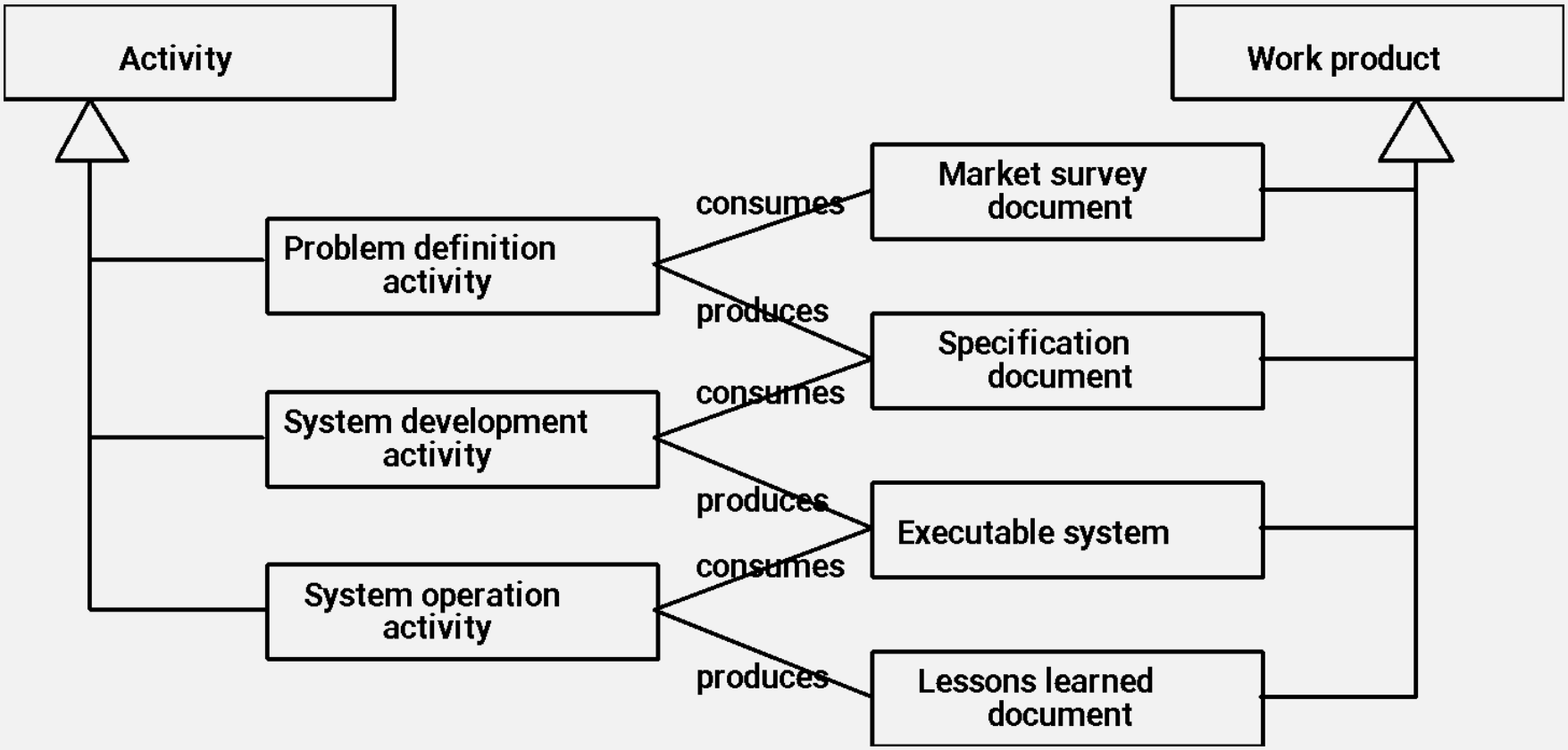
- Software development consists of the creation of a set of deliverables.

Entity-centered view of Software Development

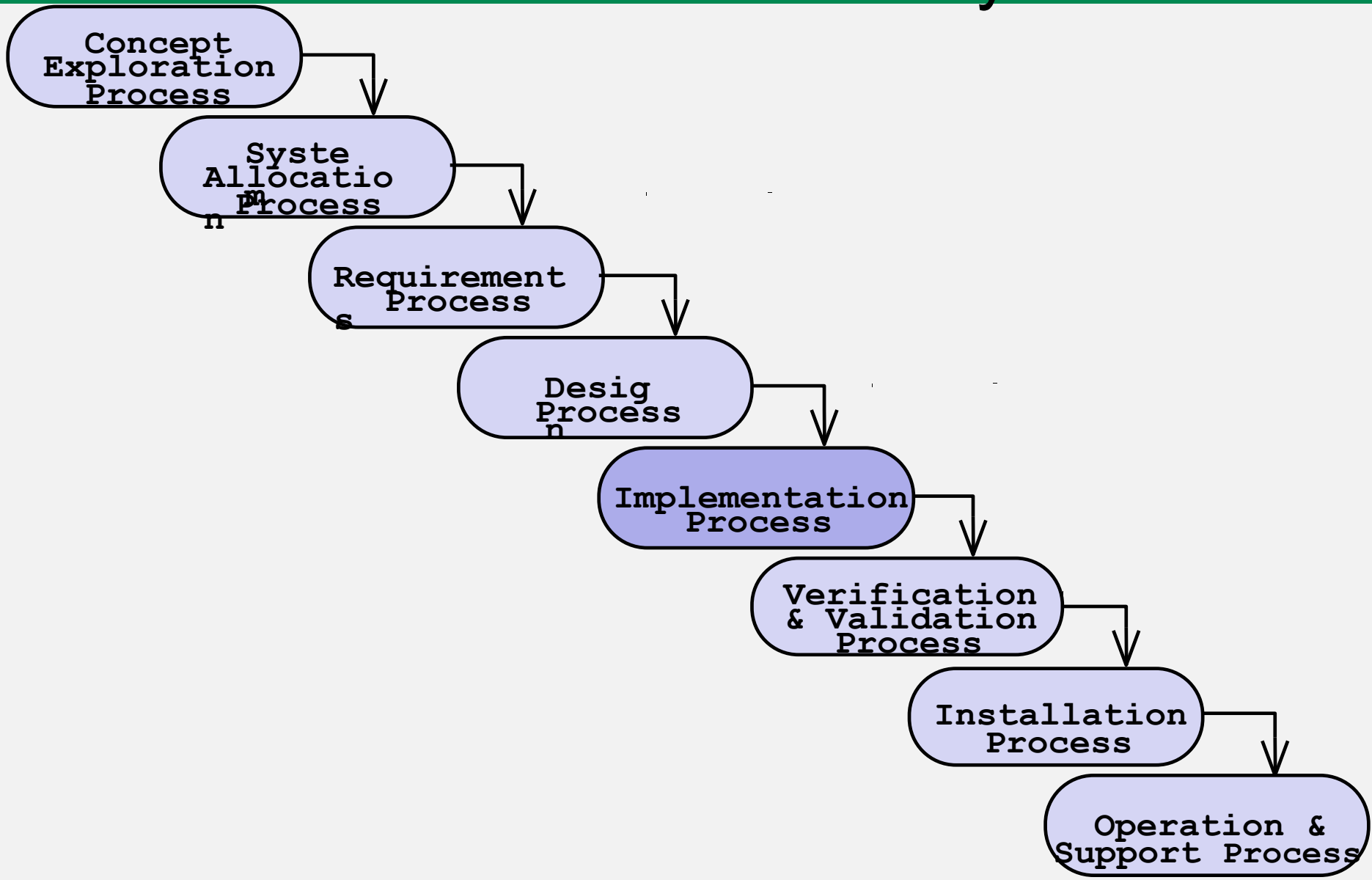


Software development consists of the creation of a set of deliverables

Combining Activities and Entities in One View



The Waterfall Model of the Software Life Cycle

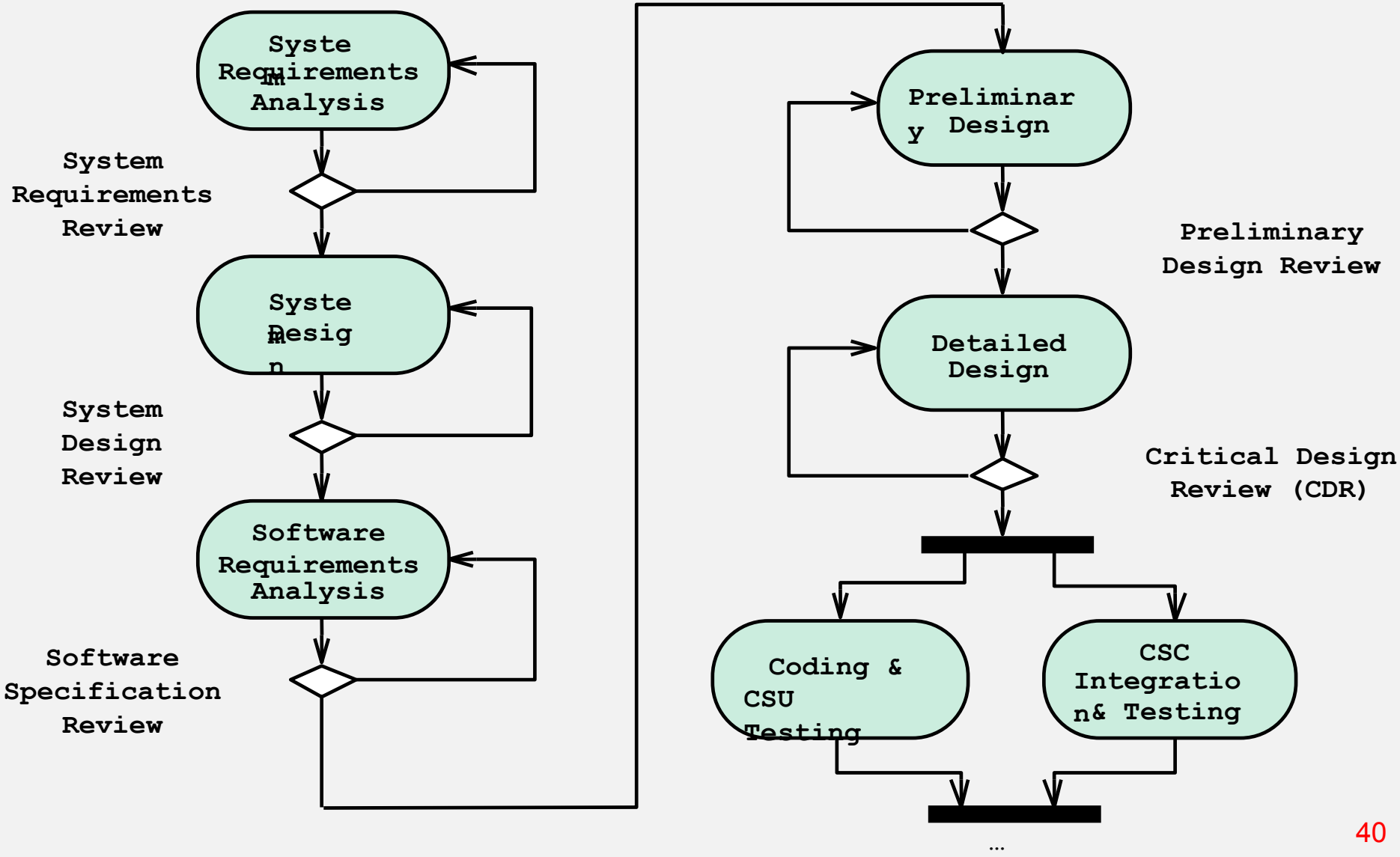


Example of a waterfall model : MIL-DOD-STD-2167A

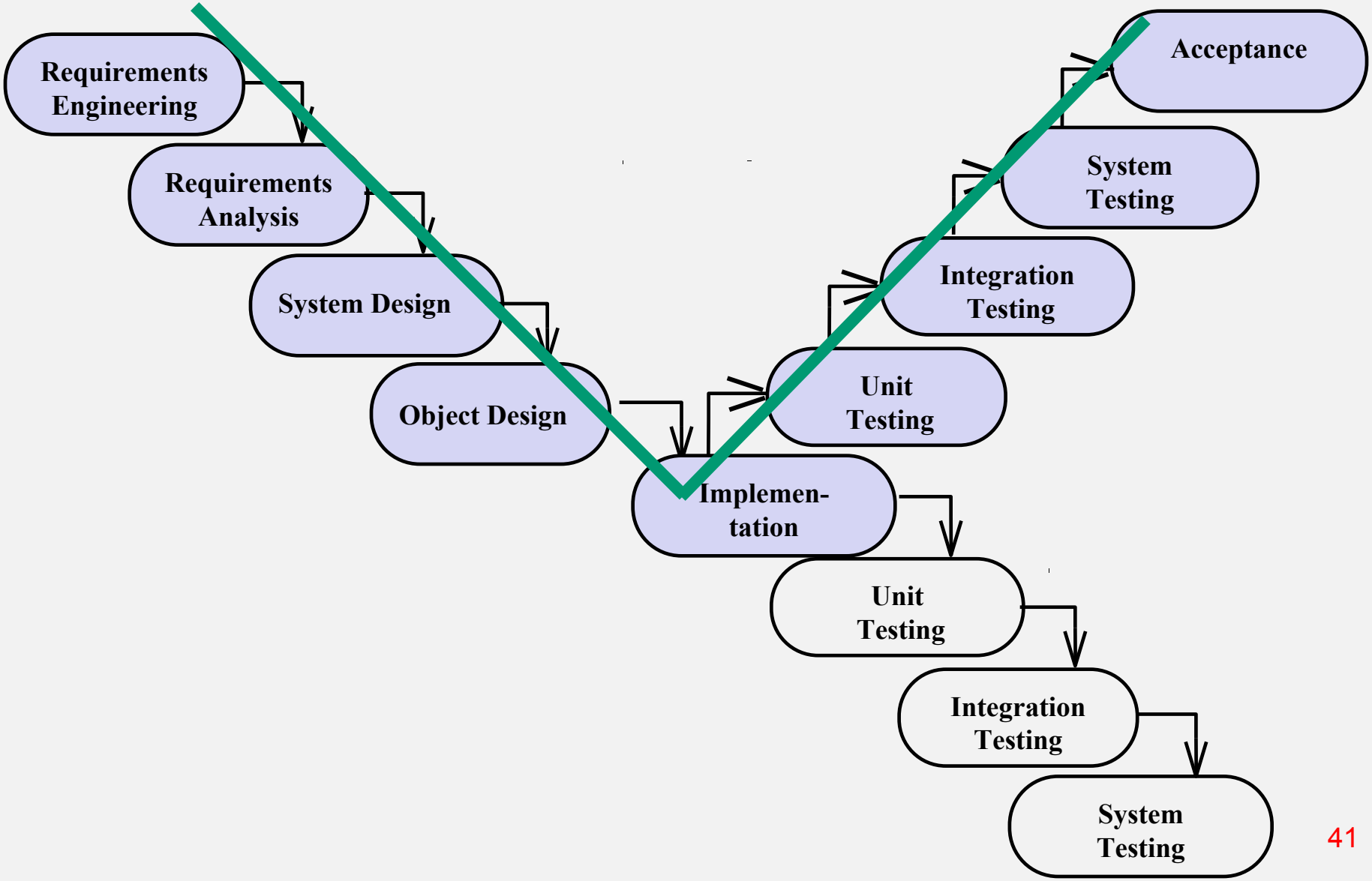
Software development activities:

- System Requirements Analysis/Design
- Software Requirements Analysis
- Preliminary Design and Detailed Design
- Coding and Unit testing
- Integration Testing
- System integration and Testing

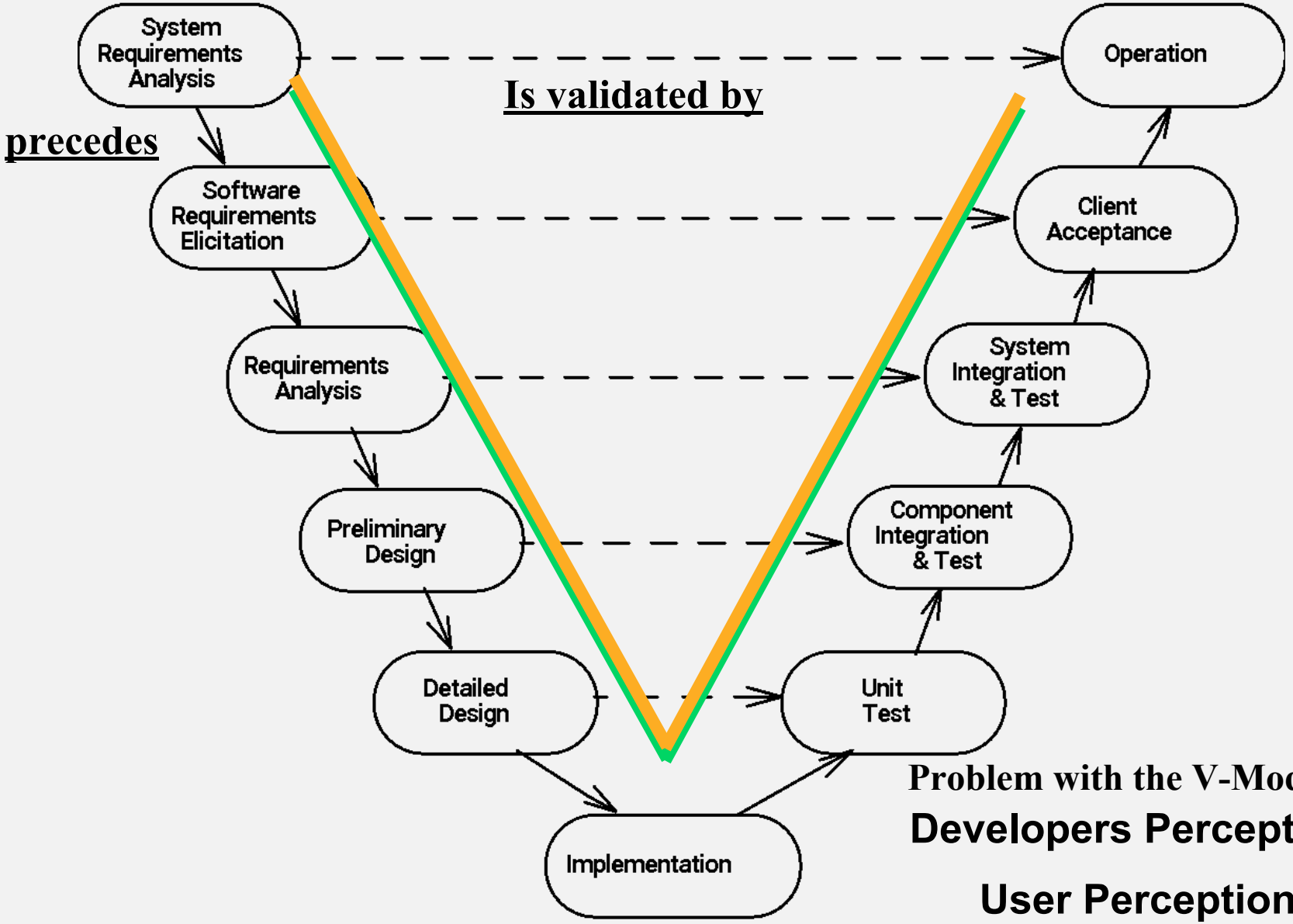
Activity Diagram of MIL DOD-STD-2167A



From the Waterfall Model to the V Model



Activity Diagram of the V Model



Problem with the V-Model:

Developers Perception =

User Perception

Properties of Waterfall-based Models

Managers love waterfall models

- Nice milestones
- No need to look back (linear system)
- Always one activity at a time
- Easy to check progress during development: 90% coded, 20% tested

However, software development is non-linear

- While a design is being developed, problems with requirements are identified
- While a program is being coded, design and requirement problems are found
- While a program is tested, coding errors, design errors and requirement errors are found.

Spiral Model

The spiral model proposed by Boehm has the following set of activities

- Determine objectives and constraints
- Evaluate alternatives
- Identify risks
- Resolve risks by assigning priorities to risks
- Develop a series of prototypes for the identified risks starting with the highest risk
- Use a waterfall model for each prototype development
- If a risk has successfully been resolved, evaluate the results of the round and plan the next round
- If a certain risk cannot be resolved, terminate the project immediately

This set of activities is applied to a couple of so-called rounds.

Rounds in Boehm's Spiral Model

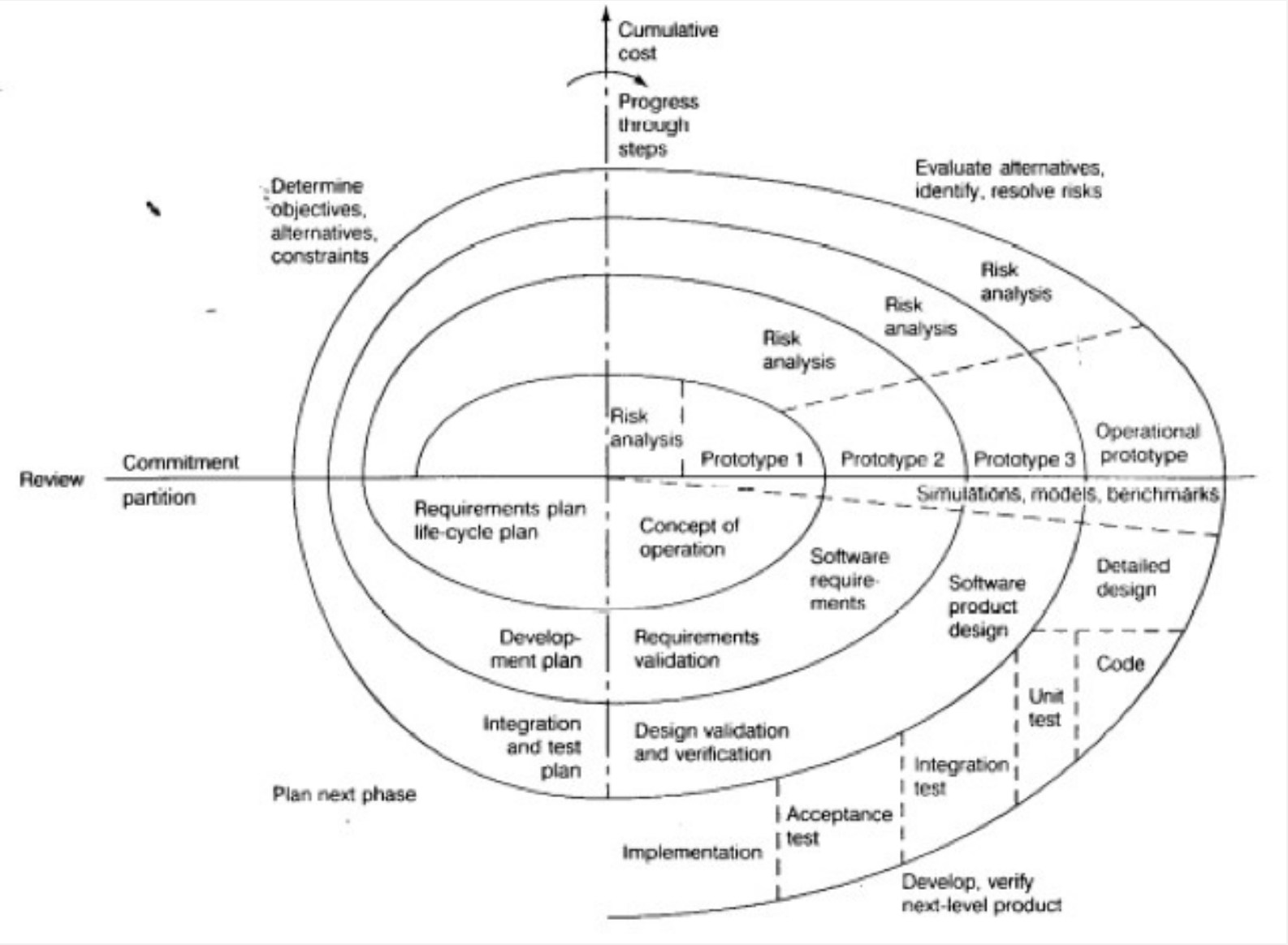
For each round go through these activities:

- Define objectives, alternatives, constraints
- Evaluate alternatives, identify and resolve risks
- Develop and verify a prototype
- Plan the next round.

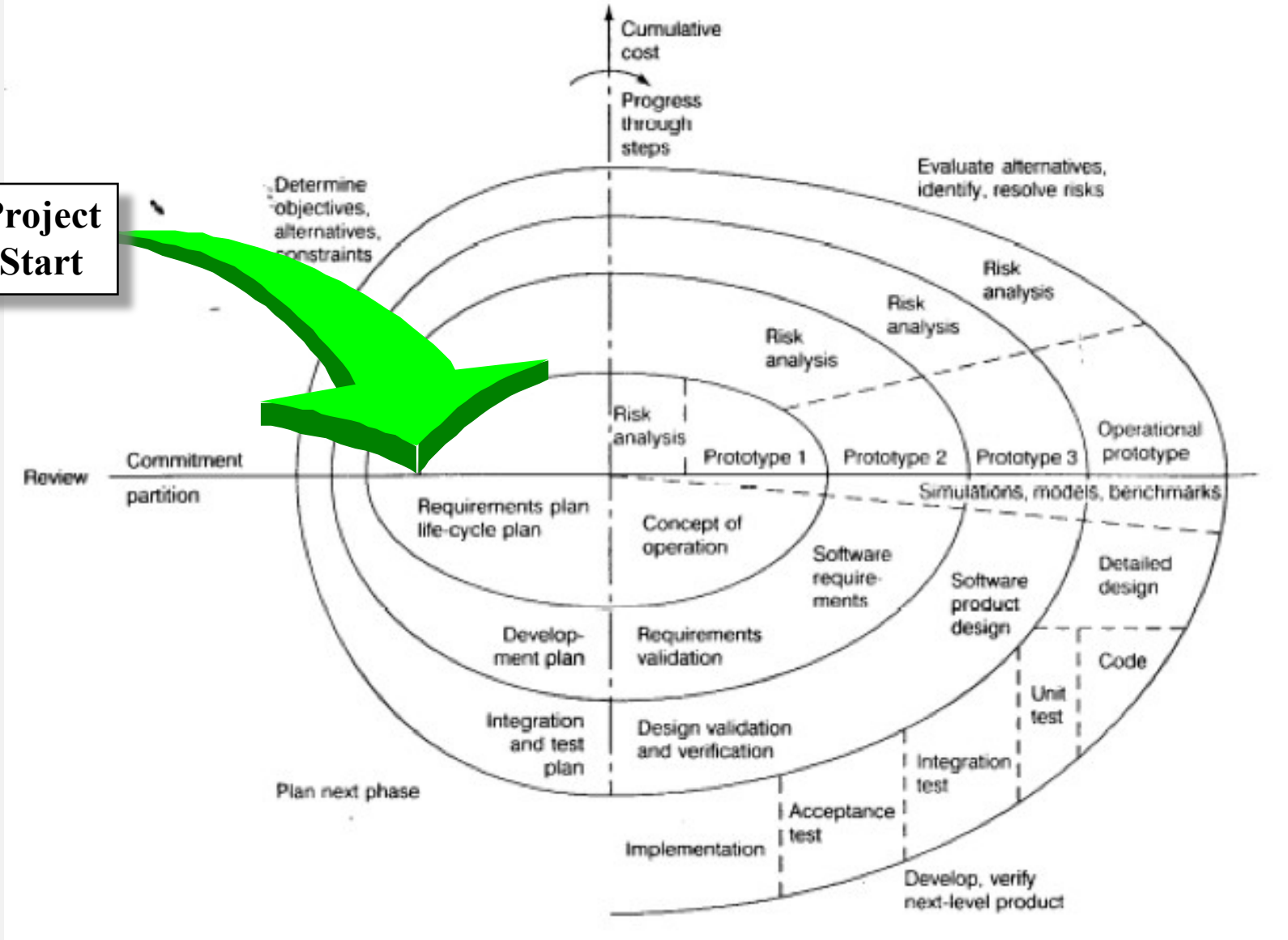
round

Concept of Operations
Software Requirements
Software Product Design
Detailed Design
Code
Unit Test
Integration and Test
Acceptance Test
Implementation

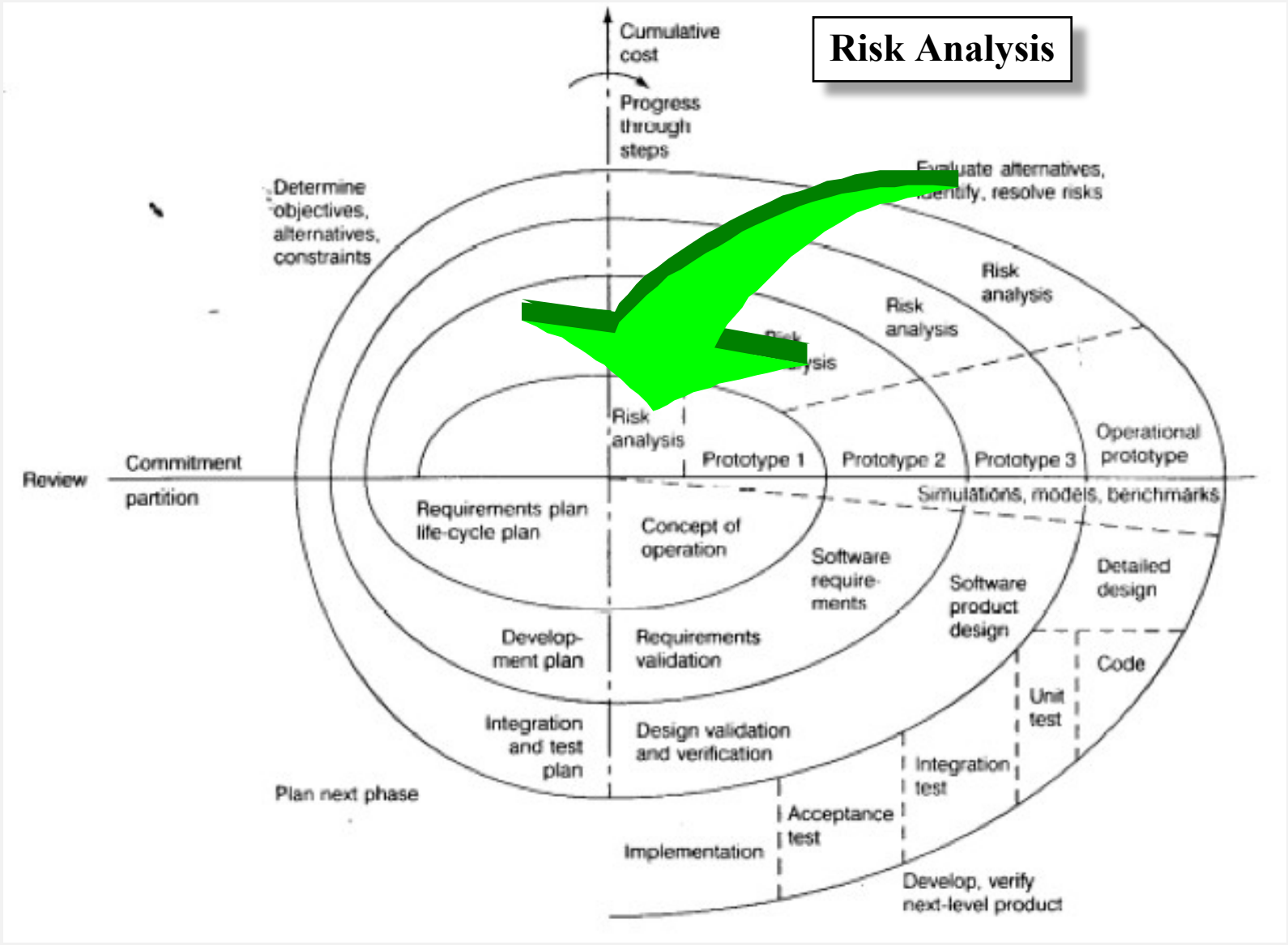
Diagram of Boehm's Spiral Model



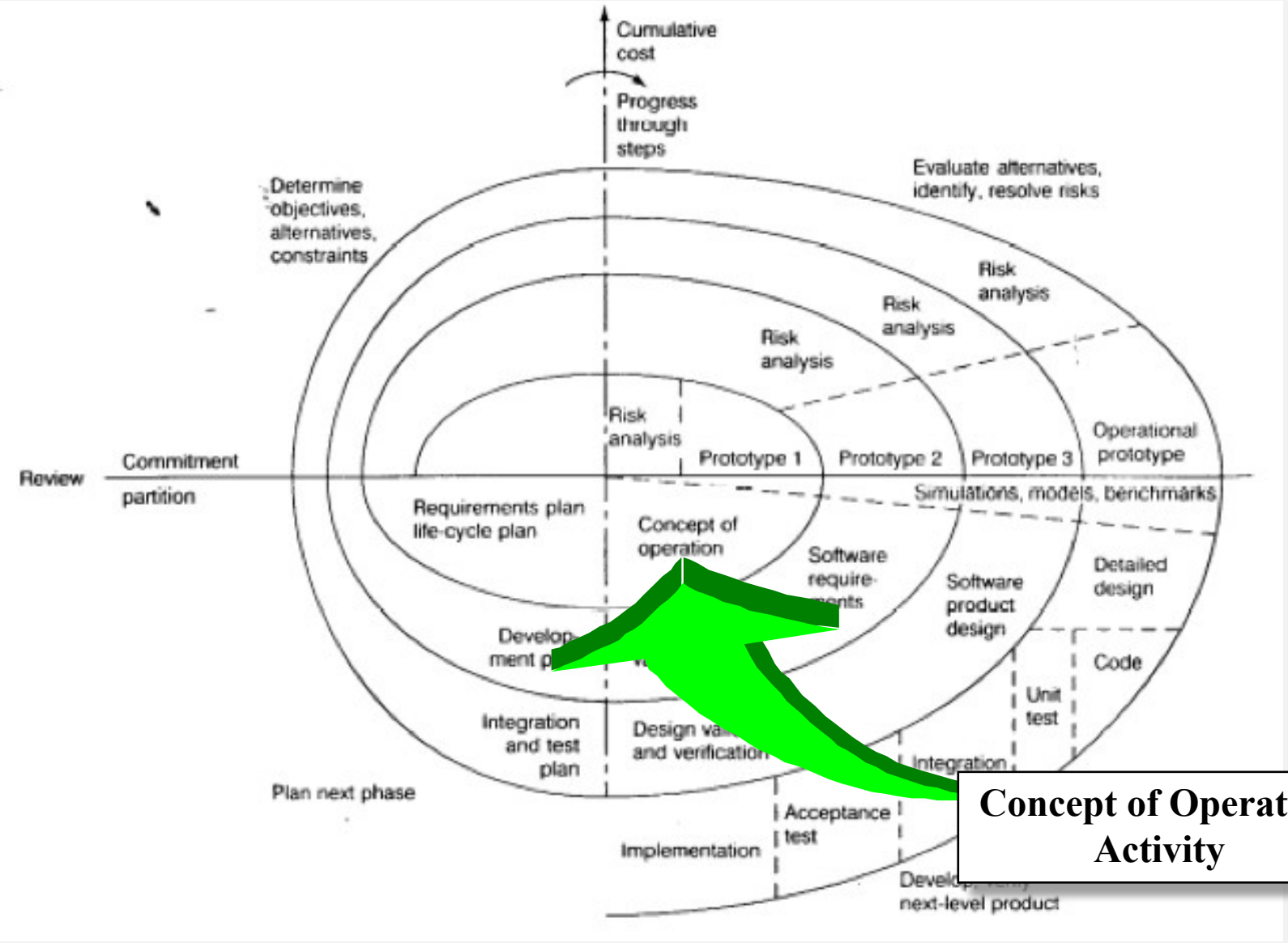
Round 1, Concept of Operations, Quadrant IV: Determine Objectives,Alternatives & Constraints



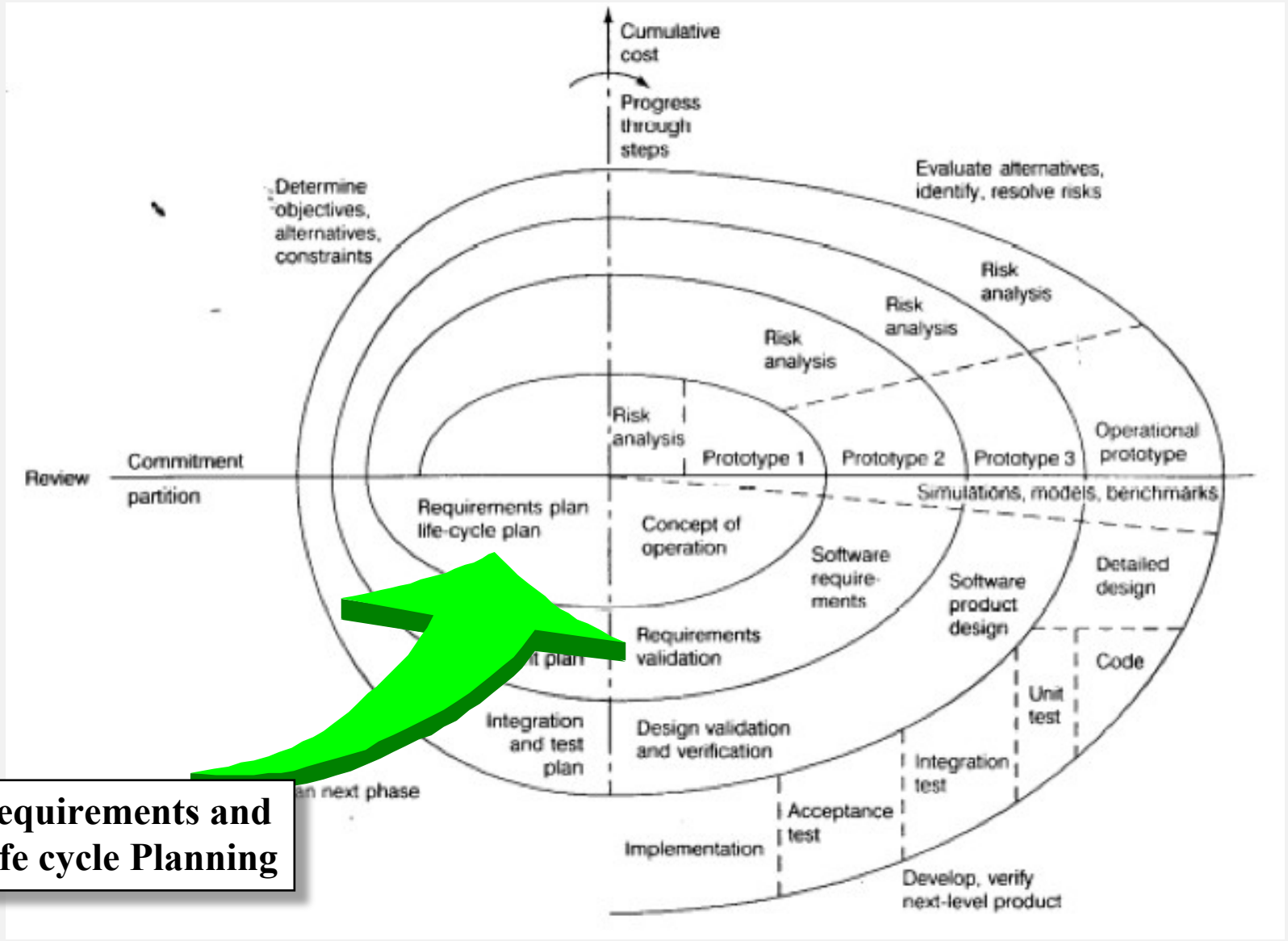
Round 1, Concept of Operations, Quadrant I: Evaluate Alternatives, identify & resolve Risks



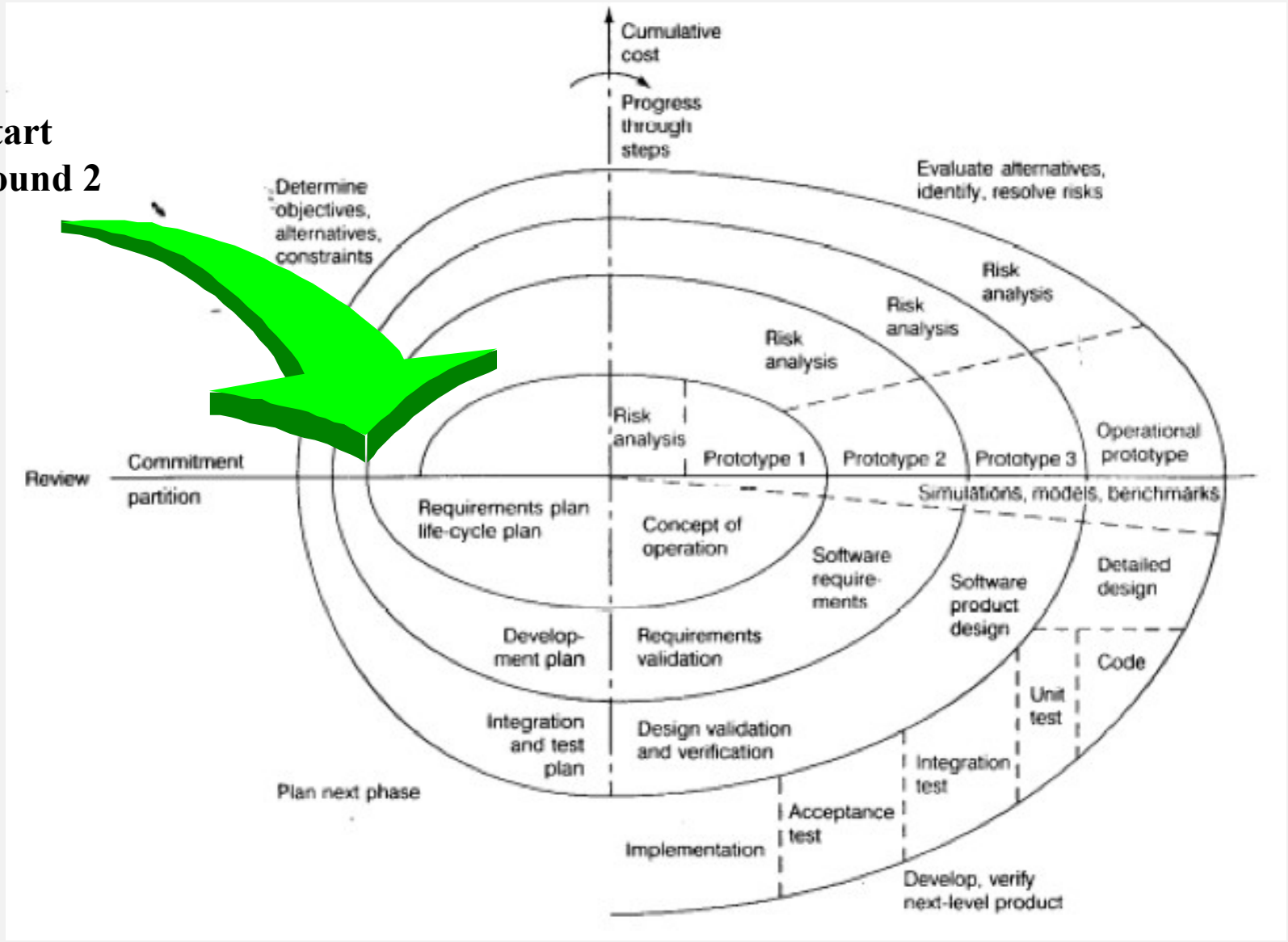
Round 1, Concept of Operations, Quadrant II: Develop and Verify



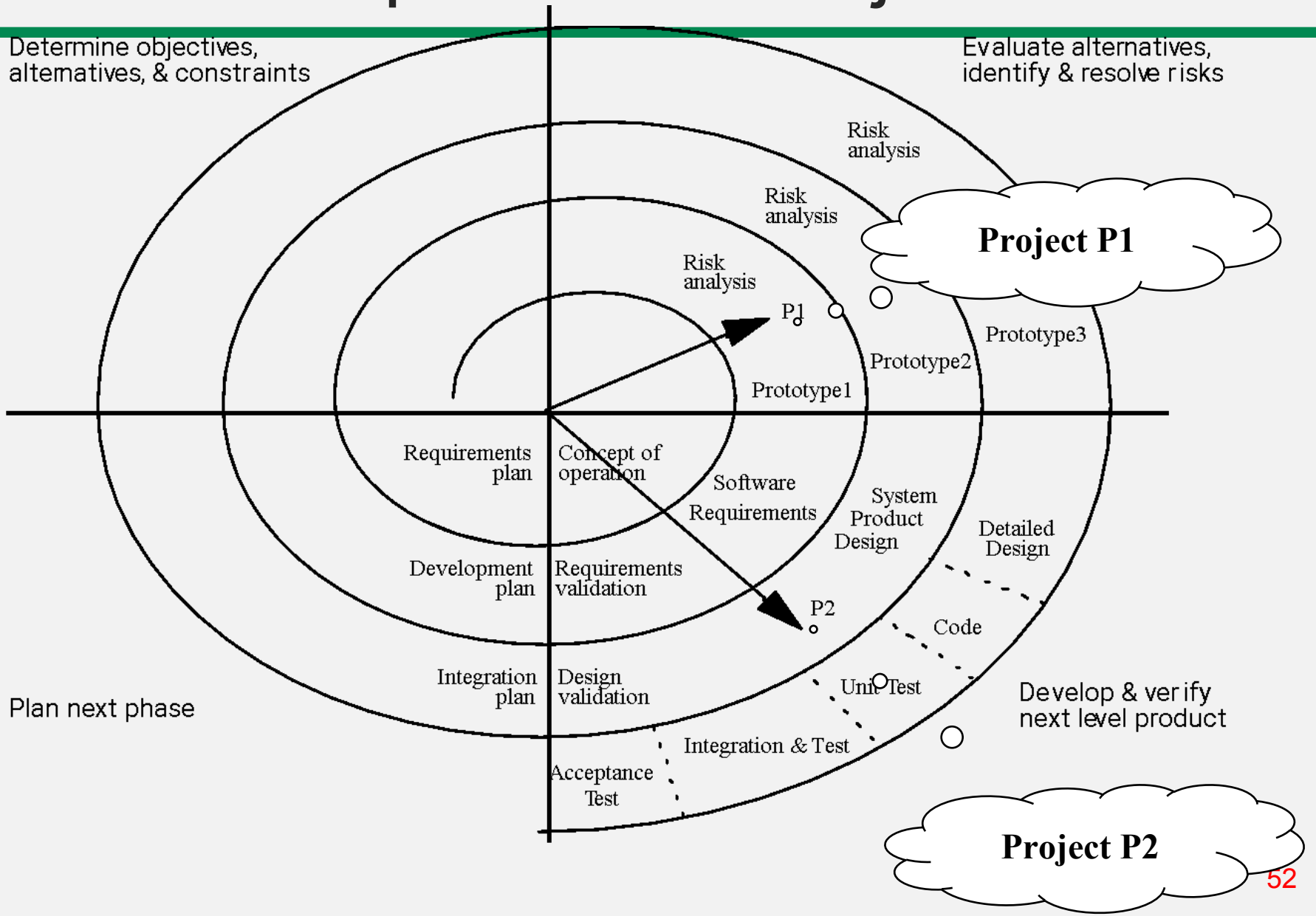
Round 1, Concept of Operations, Quadrant III: Prepare for Next Activity



Round 2, Software Requirements, Quadrant IV: Determine Objectives,Alternatives & Constraints



Comparison of Projects



Limitations of Waterfall and Spiral Models

Neither of these models deal well with **frequent change**

- The Waterfall model assumes that once you are done with a phase, all issues covered in that phase are closed and cannot be reopened
- The Spiral model can deal with change between phases, but does not allow change within a phase

What do you do if change is happening more frequently?

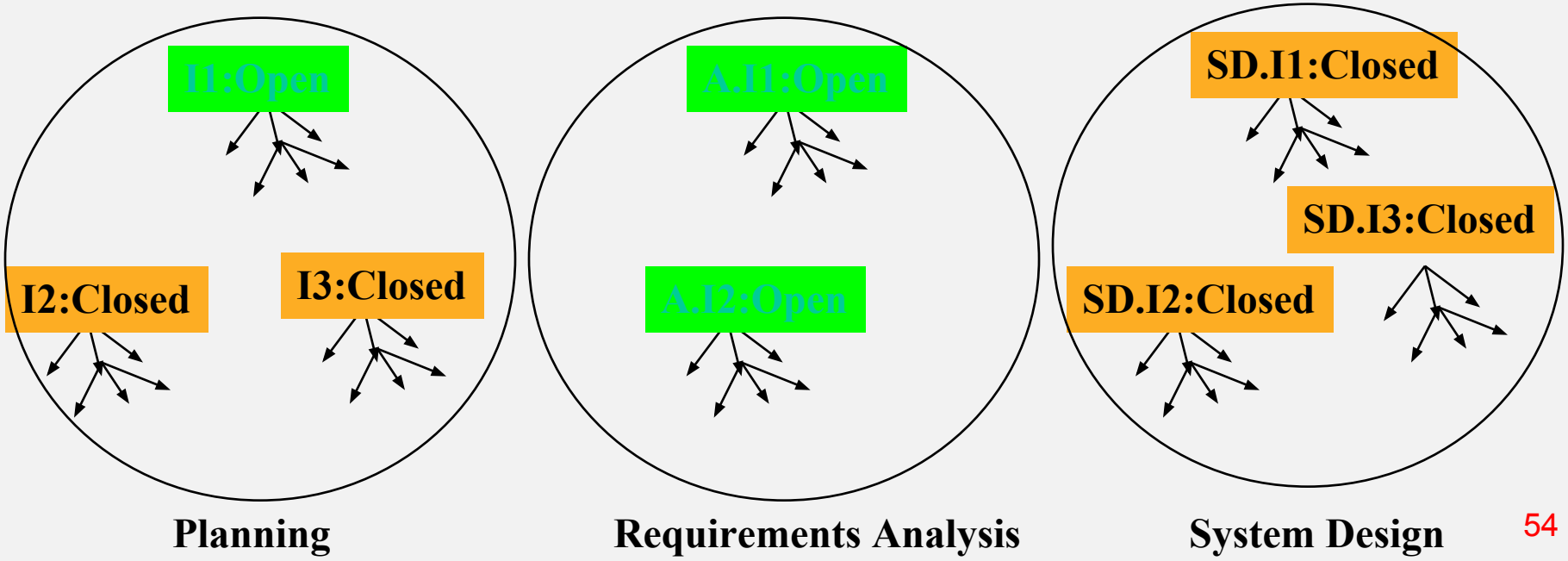
"The only constant is the change"

An Alternative: Issue-Based Development

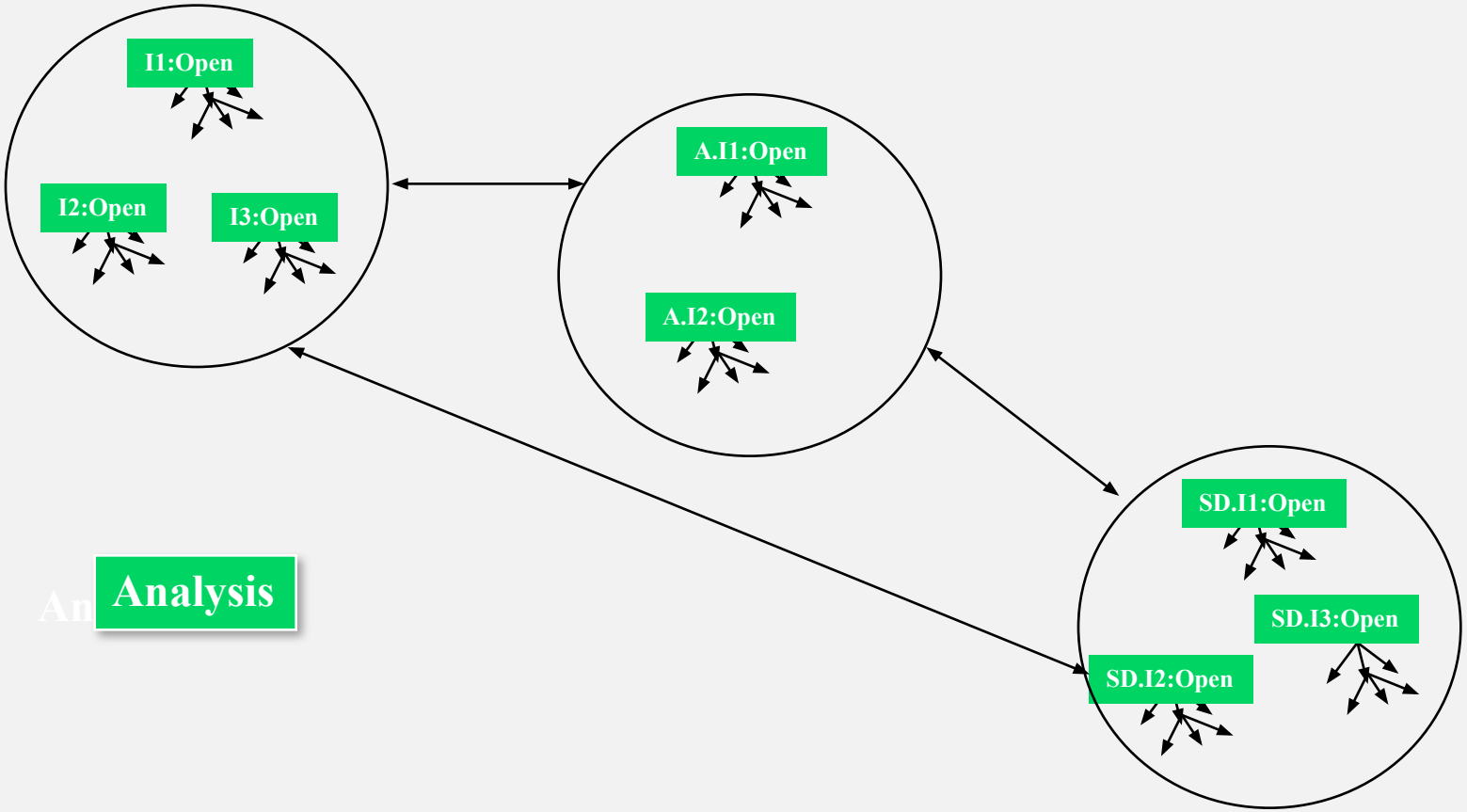
A system is described as a collection of issues

- Issues are either closed or open
- Closed issues have a resolution
- Closed issues can be reopened (Iteration!)

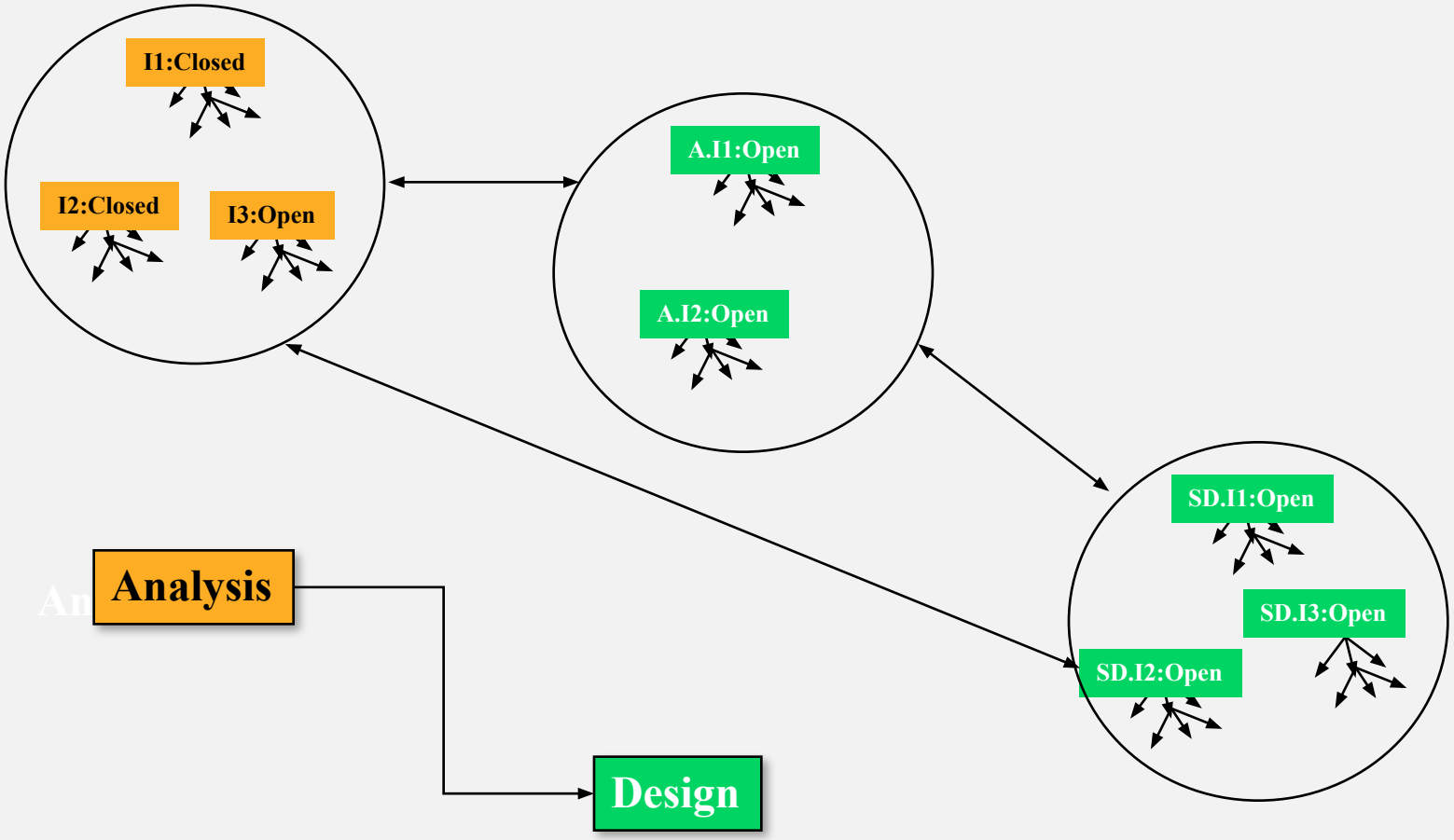
The set of closed issues is the basis of the system model



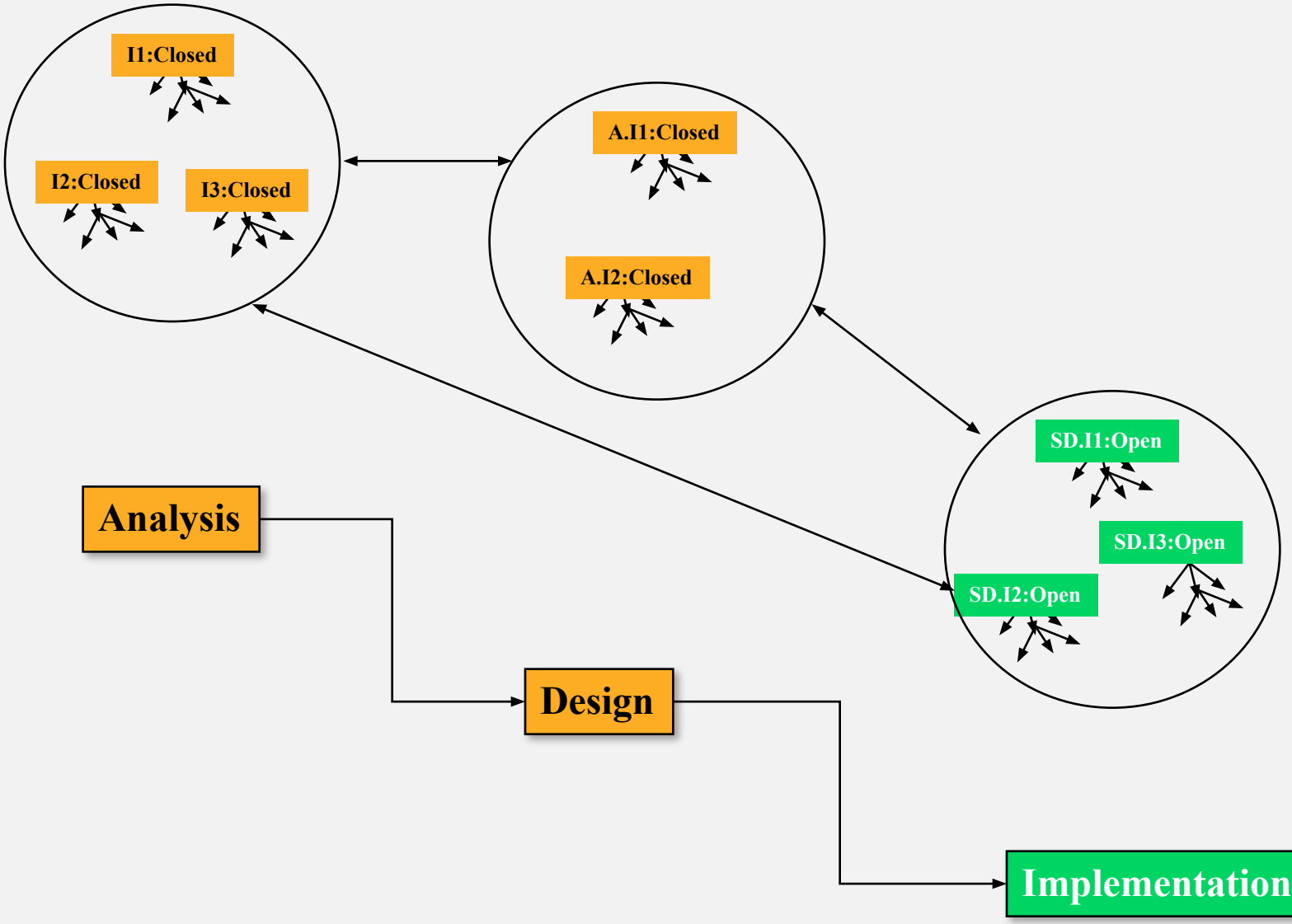
Waterfall Model: Analysis Phase



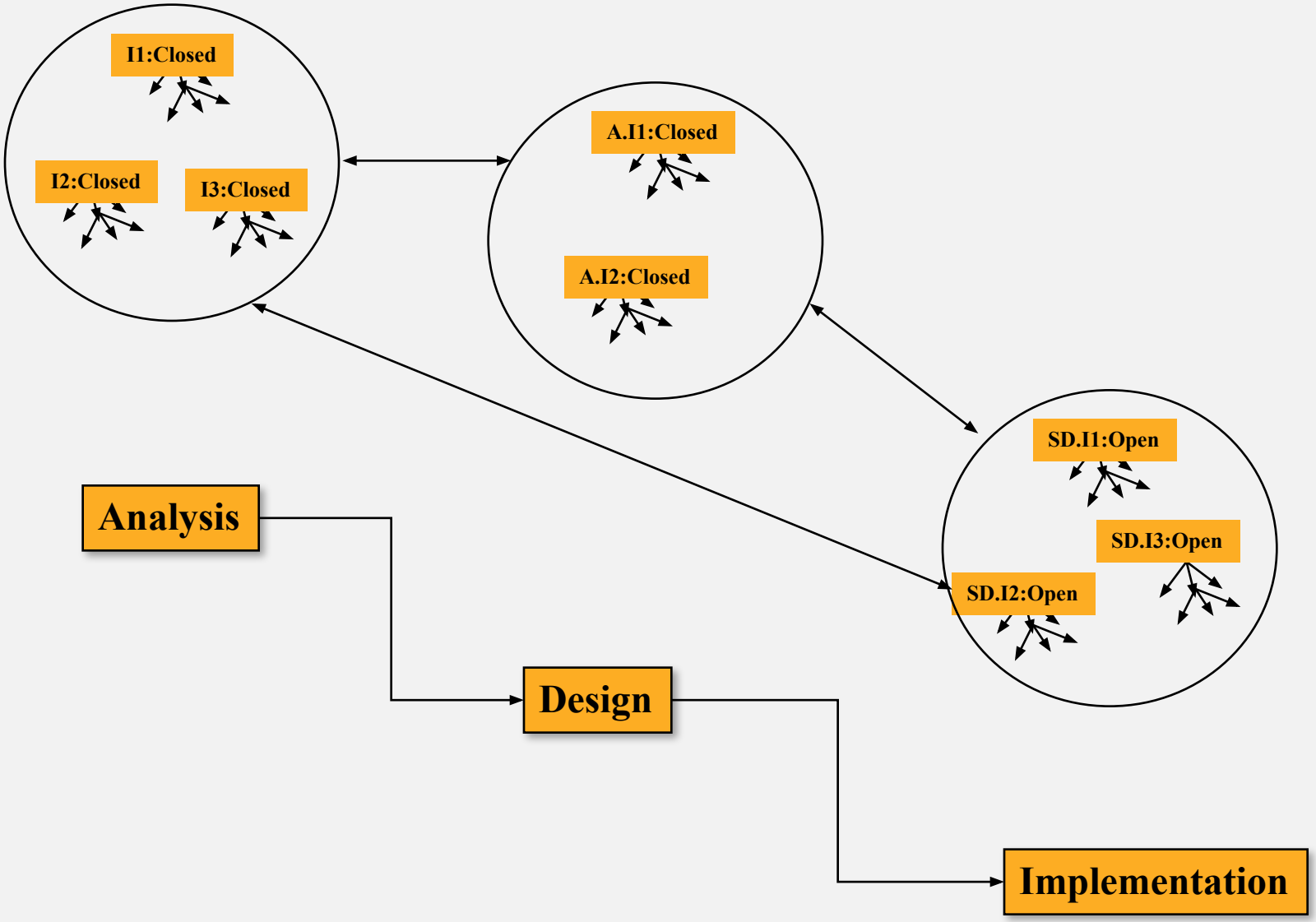
Waterfall Model: Design Phase



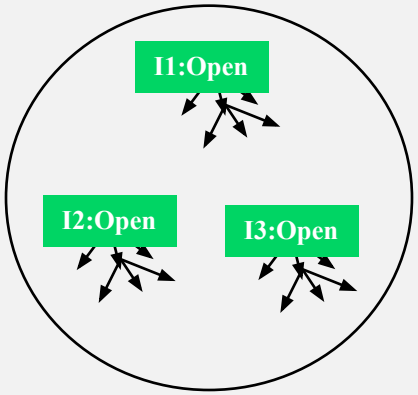
Waterfall Model: Implementation Phase

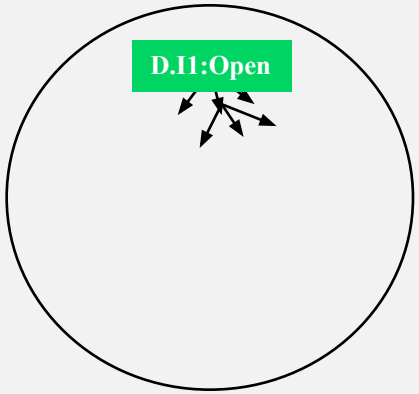


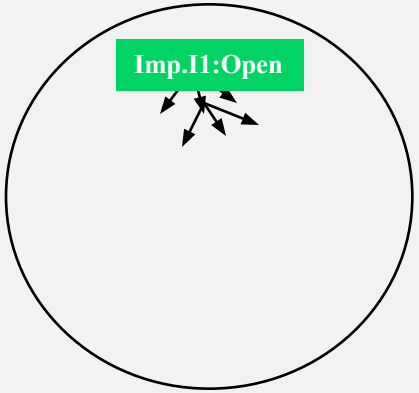
Waterfall Model: Project is Done



Issue-Based Model: Analysis Phase





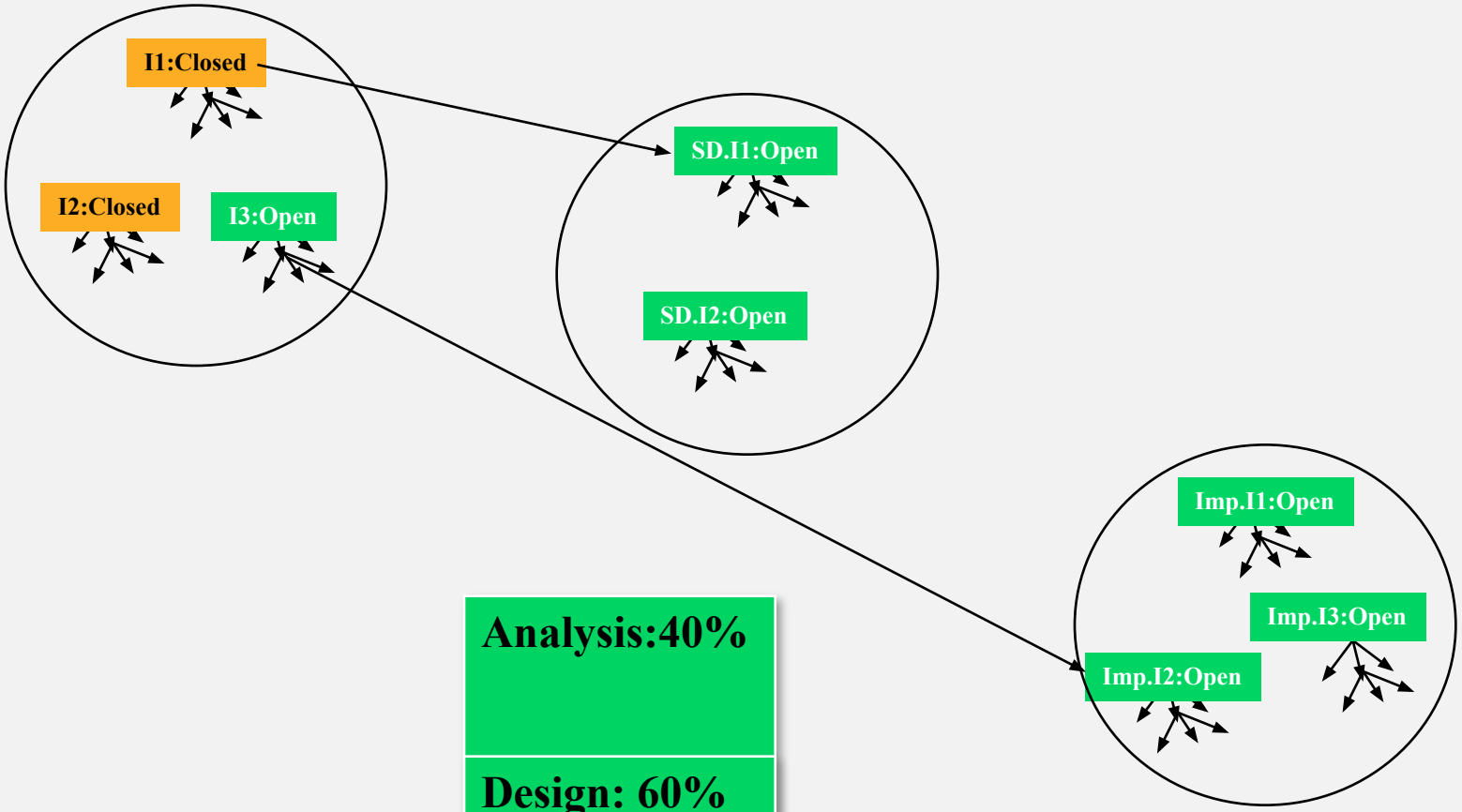


Analysis:80%

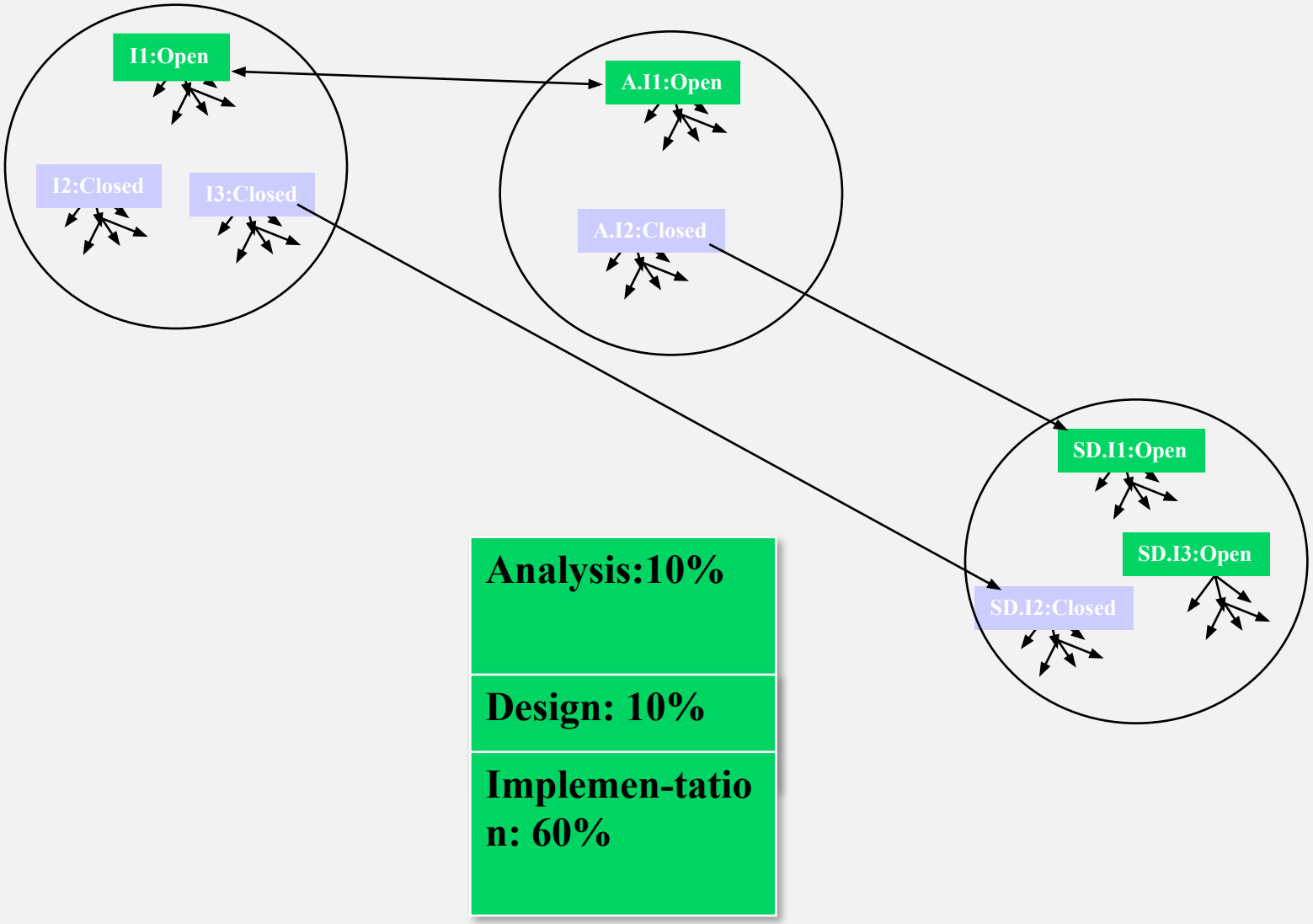
Design: 10%

Implemen-tation: 10%

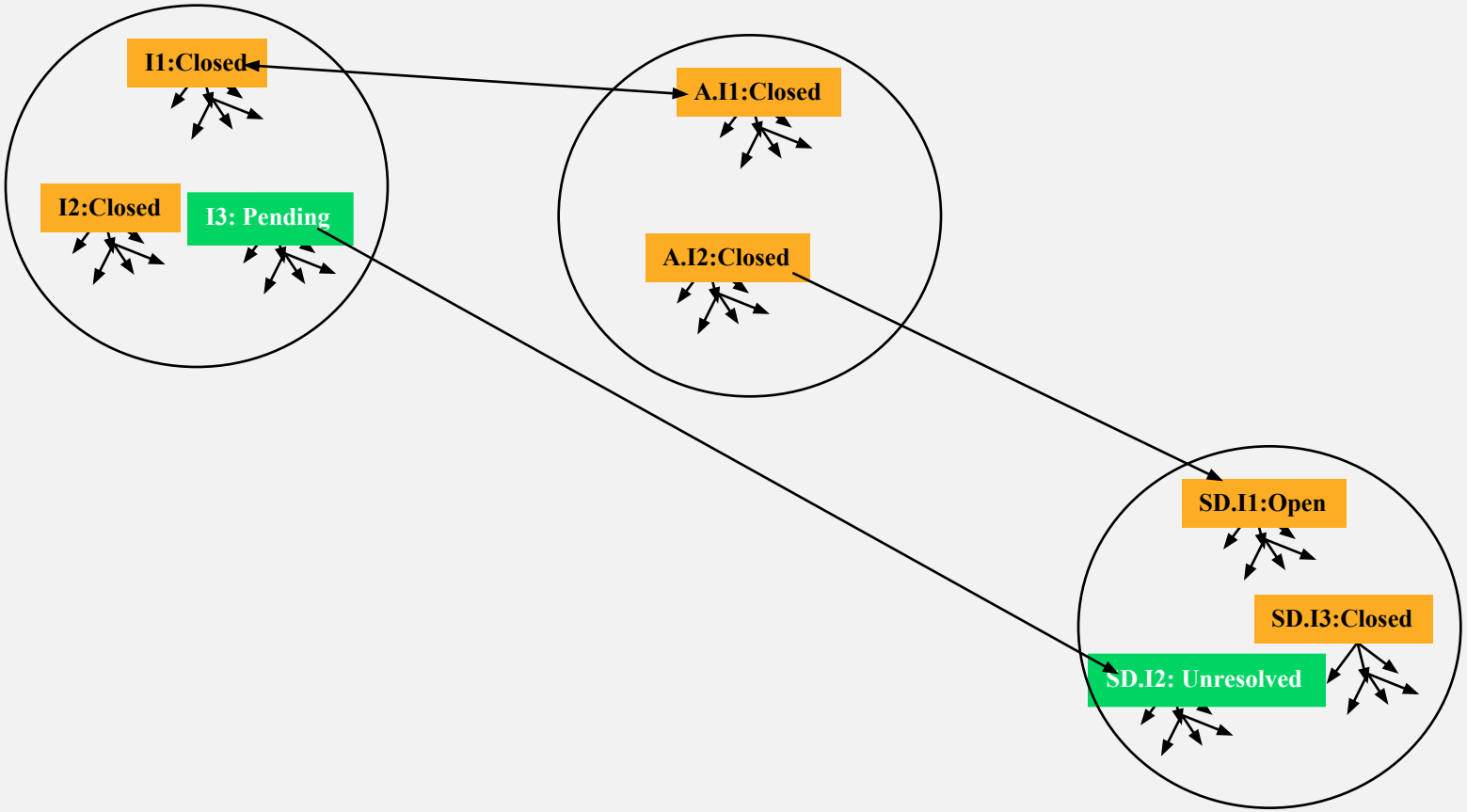
Issue-Based Model: Design Phase



Issue-Based Model: Implementation Phase



Issue-Based Model: Prototype is Done



Frequency of Change and Choice of Software Lifecycle Model

PT = Project Time, MTBC = Mean Time Between Change

Change rarely occurs (MTBC » PT)

- Linear Model (Waterfall, V-Model)
- Open issues are closed before moving to next phase

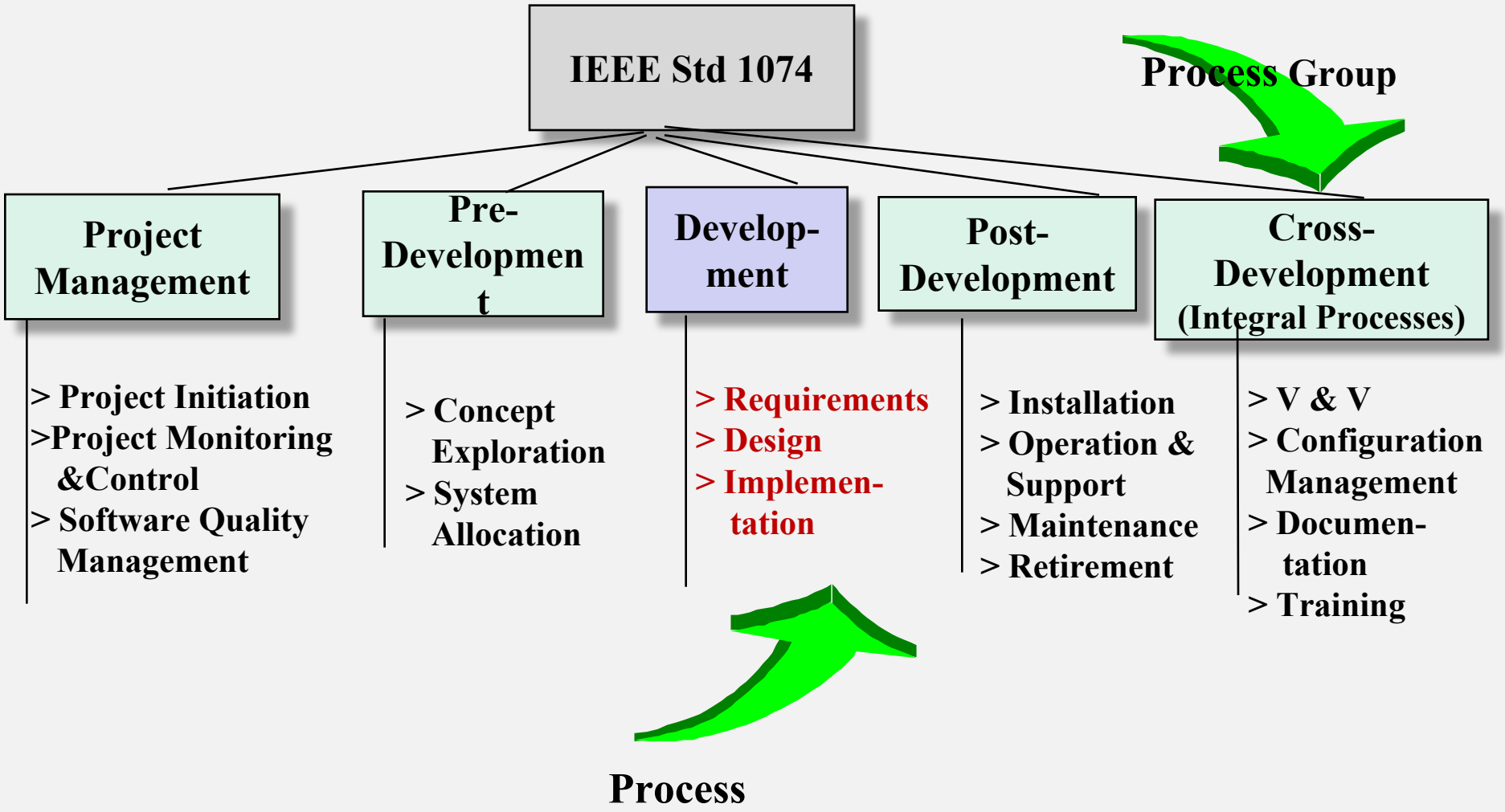
Change occurs sometimes (MTBC ≈ PT)

- Iterative model (Spiral Model, Unified Process)
- Change occurring during phase may lead to iteration of a previous phase or cancellation of the project

Change is frequent (MTBC « PT)

- Issue-based Model (Concurrent Development, Scrum)
- Phases are never finished, they all run in parallel.

IEEE Std 1074: Standard for Software Life Cycle Activities

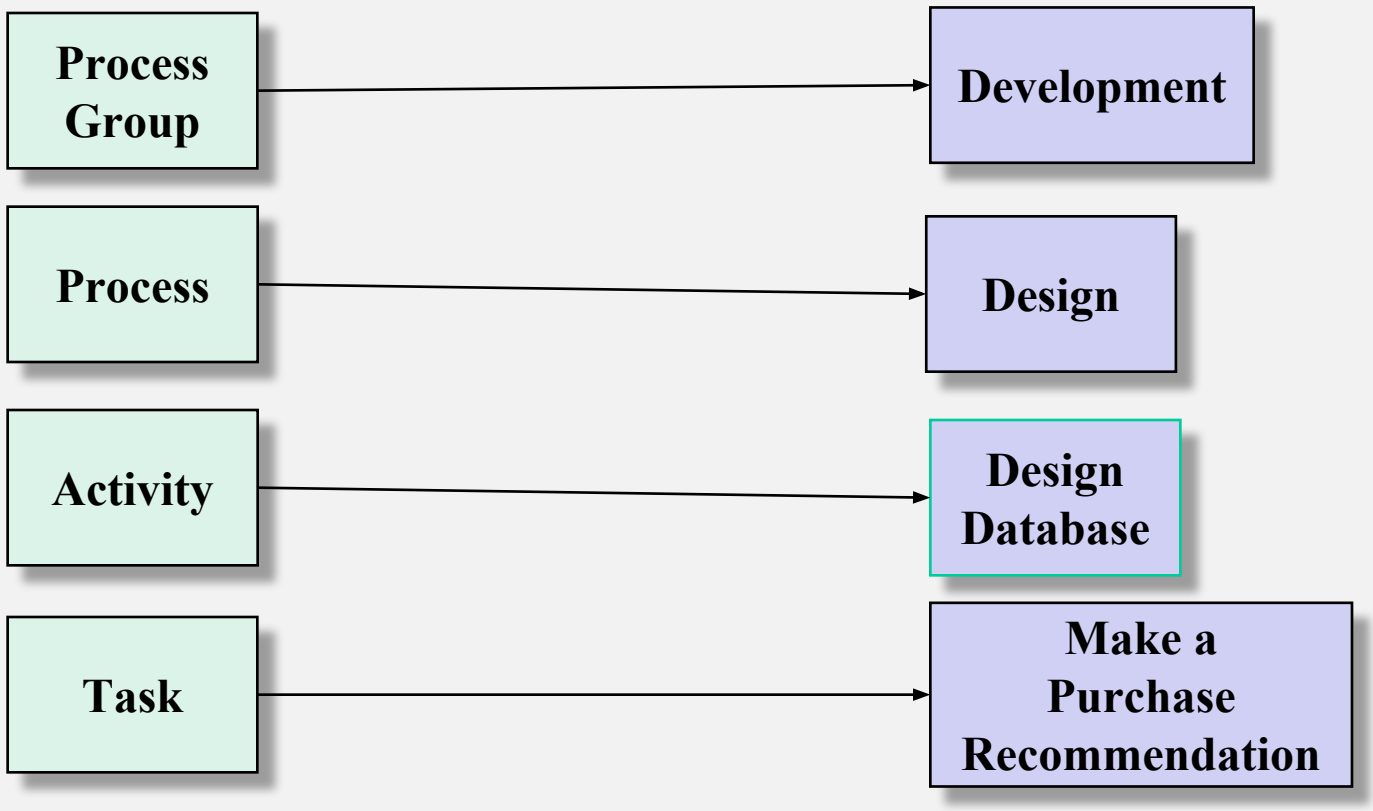


Processes, Activities and Tasks

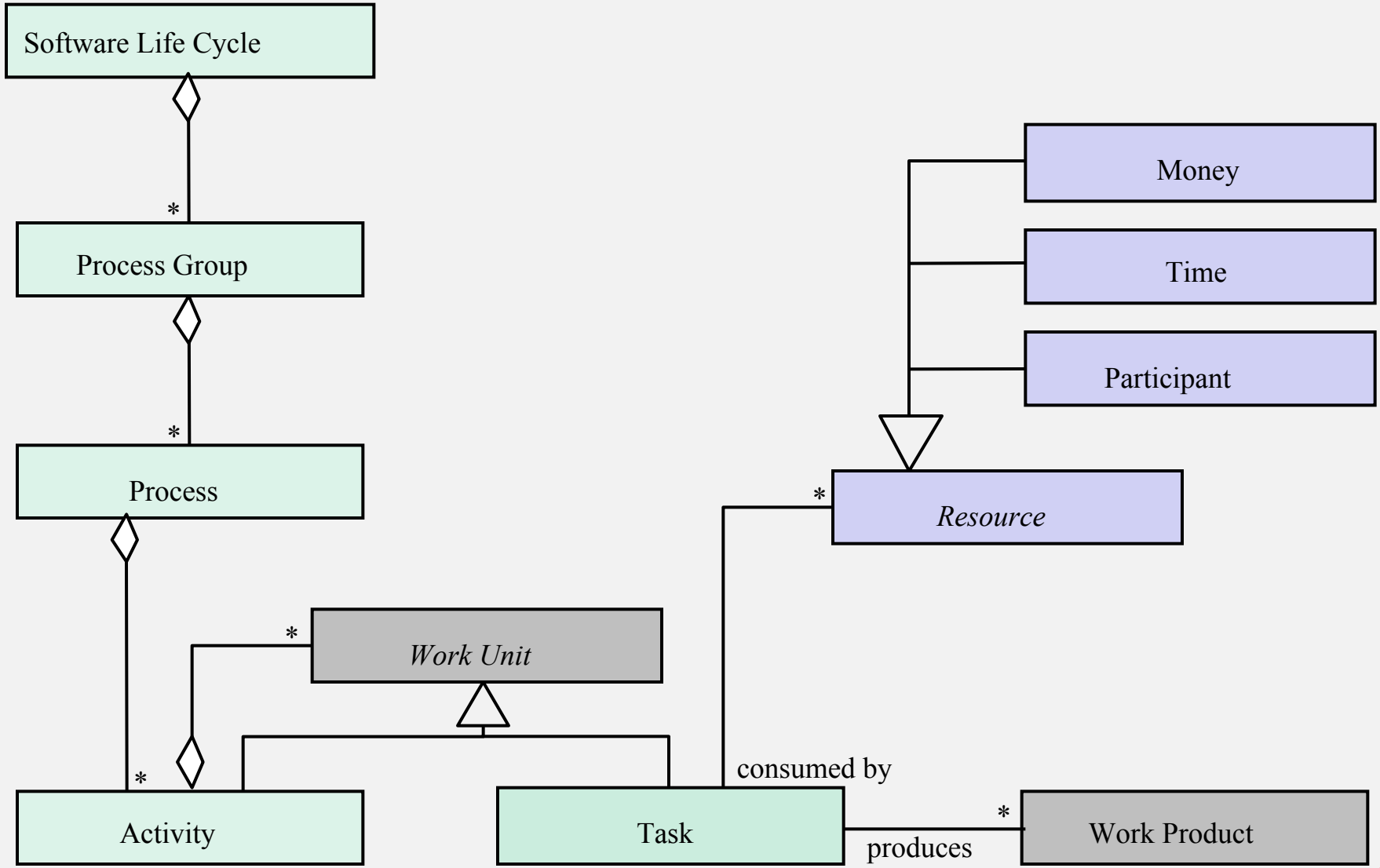
Process Group: Consists of a set of processes

Process: Consists of activities

Activity: Consists of sub activities and tasks



Object Model of the IEEE 1074 Standard



UML Basic Notation Summary

UML provides a wide variety of notations for modeling many aspects of software systems

Today we concentrated on a few notations:

- Functional model: Use case diagram
- Object model: Class diagram
- Dynamic model: Sequence diagrams, statechart

Thank You