Software Engineering, Analysis, and Design Concepts

Teknik Informatika – Universitas Komputer Indonesia

List of Material

- Introduction of Software Engineering
- SDLC and Process Model
- Analysis and its principles
- Design and its principles
- Object Oriented Analysis and Design (OOAD)

Introduction of Software Engineering

What is Software?

- Software is a product
  - *Transforms* information - produces, manages, acquires, modifies, displays, or transmits information
  - Delivers computing potential of hardware and networks
- Software is a vehicle for delivering a product
  - Controls other programs (operating system)
  - Effects communications (networking software)
  - Helps build other software (software tools & environments)

What is Software?

Hardware	Software
▪ Manufactured ▪ Wears out ▪ Built using components ▪ Relatively simple	▪ Developed/ engineered ▪ Deteriorates ▪ Custom built ▪ Complex

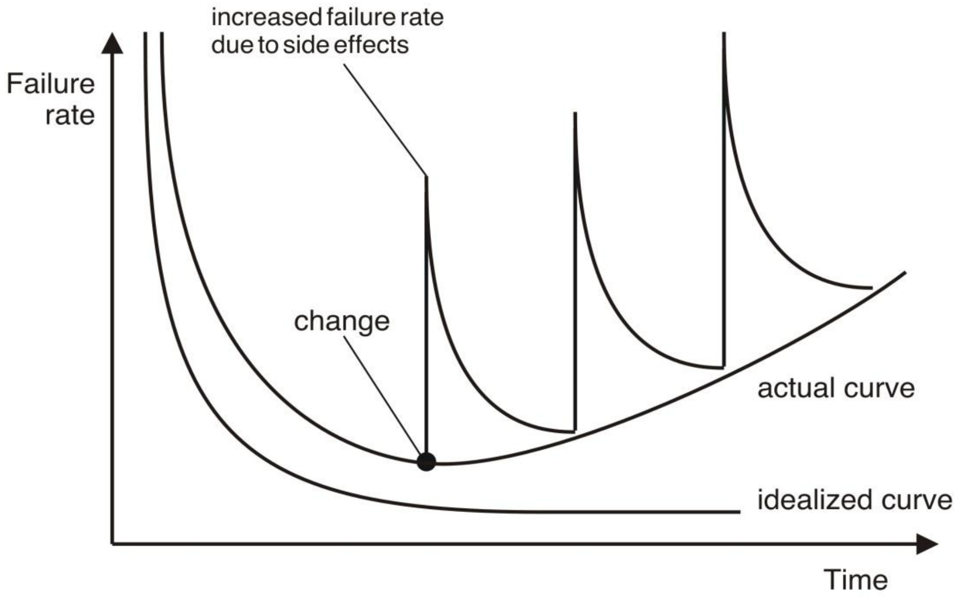
Manufacturing VS Development

- Once a hardware product has been manufactured, it is difficult or impossible to modify. In contrast, software products are routinely modified and upgraded.

- In hardware, hiring more people allows you to accomplish more work, but the same does not necessarily hold true in software engineering.

- Unlike hardware, software costs are concentrated in design rather than production

Wears VS Deteriorates



Criteria of Good Software

1. **Maintainability**

   Software must evolve to meet changing needs

2. **Dependability**

   Software must be trustworthy

3. **Efficiency**

   Software should not make wasteful use of system resources

4. **Usability**

   Software must be usable by the users for which it was designed

Software Myth

Affect managers, customers (and other non-technical stakeholders) and practitioners

Are believable because they often have elements of truth,

but …

Invariably lead to bad decisions,

therefore …

Insist on reality as you navigate your way through software engineering

Software Myth

- If we get behind schedule, we can add more programmers and catch up.
- A general statement about objectives is sufficient to begin building programs.
- Change in project requirements can be easily accommodated because software is flexible.

Software Myth

- If we get behind schedule, we can add more programmers and catch up.
- A general statement about objectives is sufficient to begin building programs.
- Change in project requirements can be easily accommodated because software is flexible.

How to break those Myths?

What is Software Engineering?

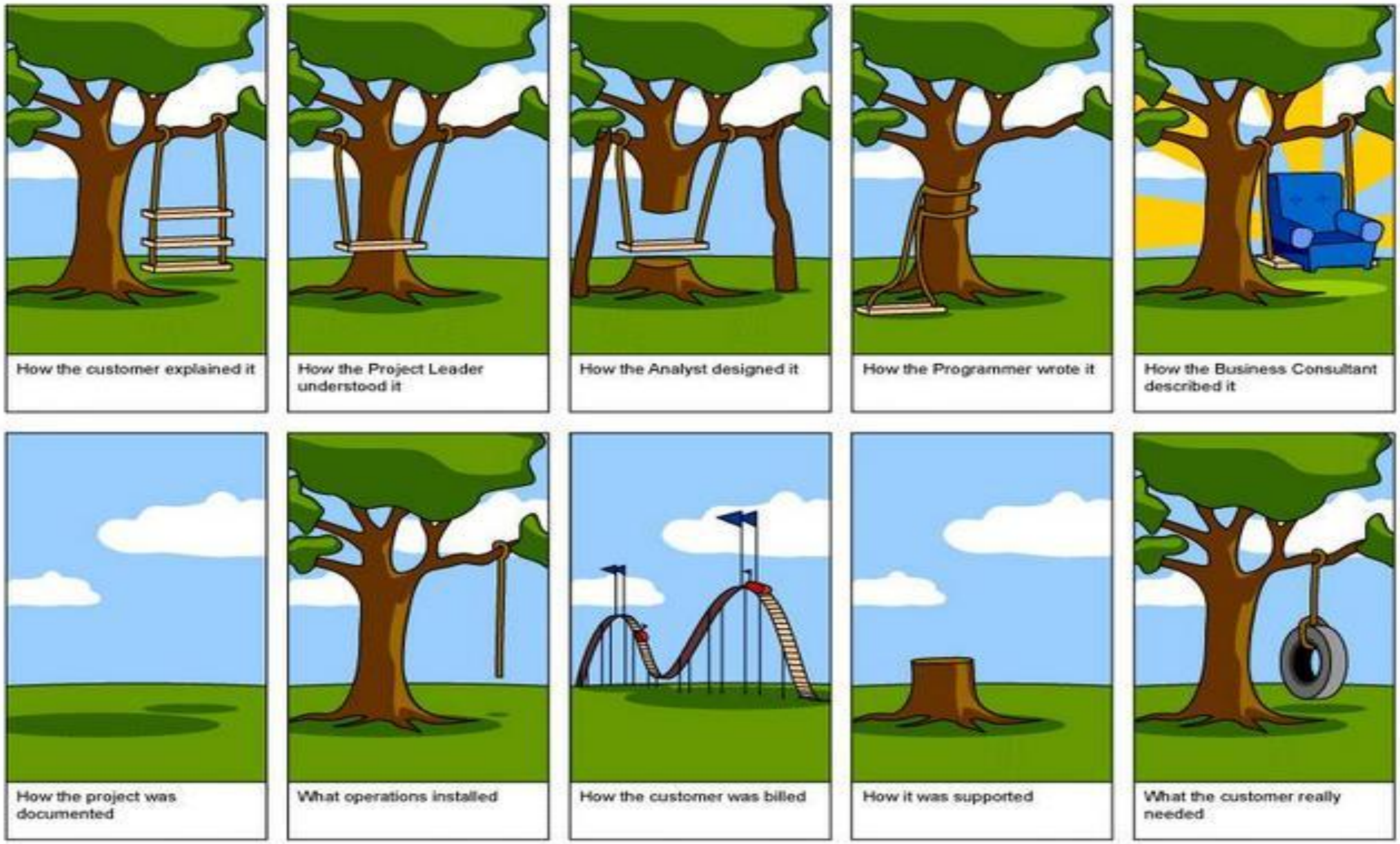
A historical definition:

"The establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines …" [Fritz Bauer, at the 1st NATO Conference on Software Engineering, 1969]

IEEE definition:

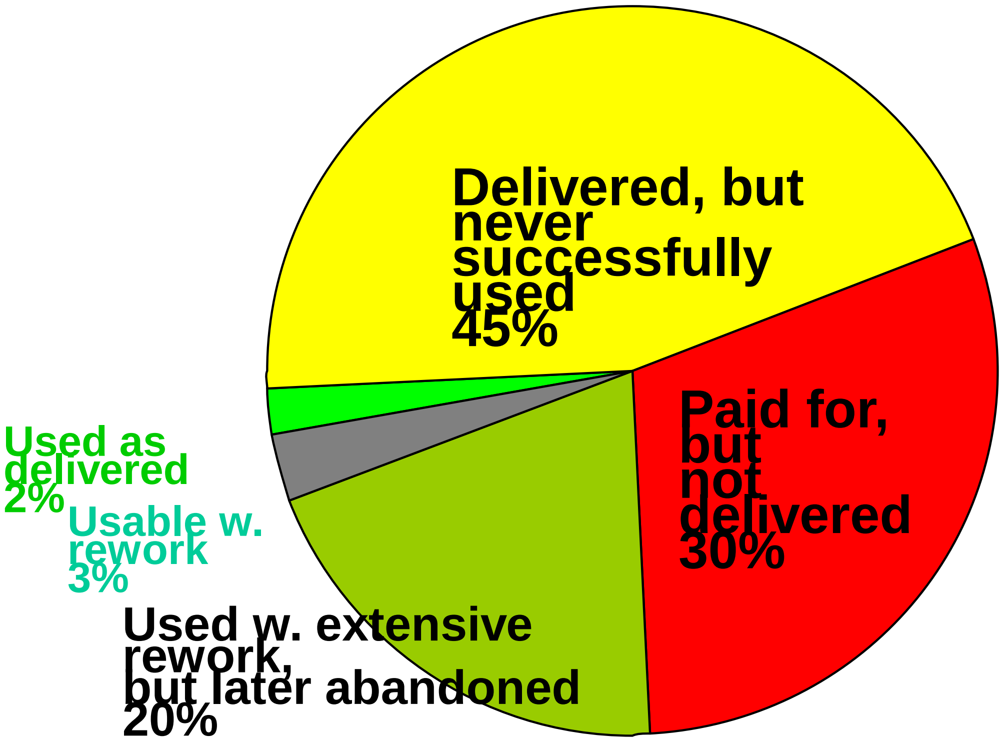
"Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software."

Why We Need Software Engineering?



Why We Need Software Engineering?

9 software projects totaling $96.7 million: Where The Money Went [Report to Congress, Comptroller General, 1979]

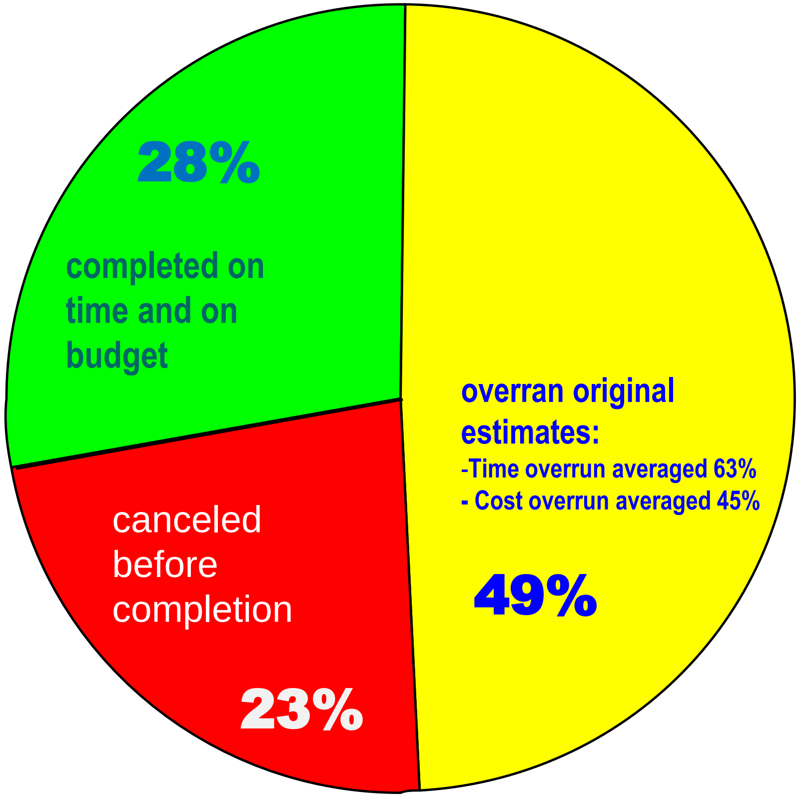


Why?

Software hurts

- **Requirements**
- **Design**

Why We Need Software Engineering?



Project Success Factors

The CHAOS Ten
1. Executive Management Support
2. **User Involvement**
3. Experienced Project Manager
4. **Clear Business Objectives**
5. **Minimized Scope**
6. Standard Software Infrastructure
7. **Firm Basic Requirements**
8. Formal Methodology
9. Reliable Estimates
10. Other

Why We Need Software Engineering?

The CHAOS Ten

Project Challenged Factors

1. Lack of User Input
2. Incomplete Requirements & Specifications
3. Changing Requirements & Specifications
4. Lack of Executive Support
5. Technology Incompetence
6. Lack of Resources
7. Unrealistic Expectations
8. Unclear Objectives
9. Unrealistic Time Frames
10. New Technology

Standish Group, '01 (www.standishgroup.com)

The CHAOS Ten

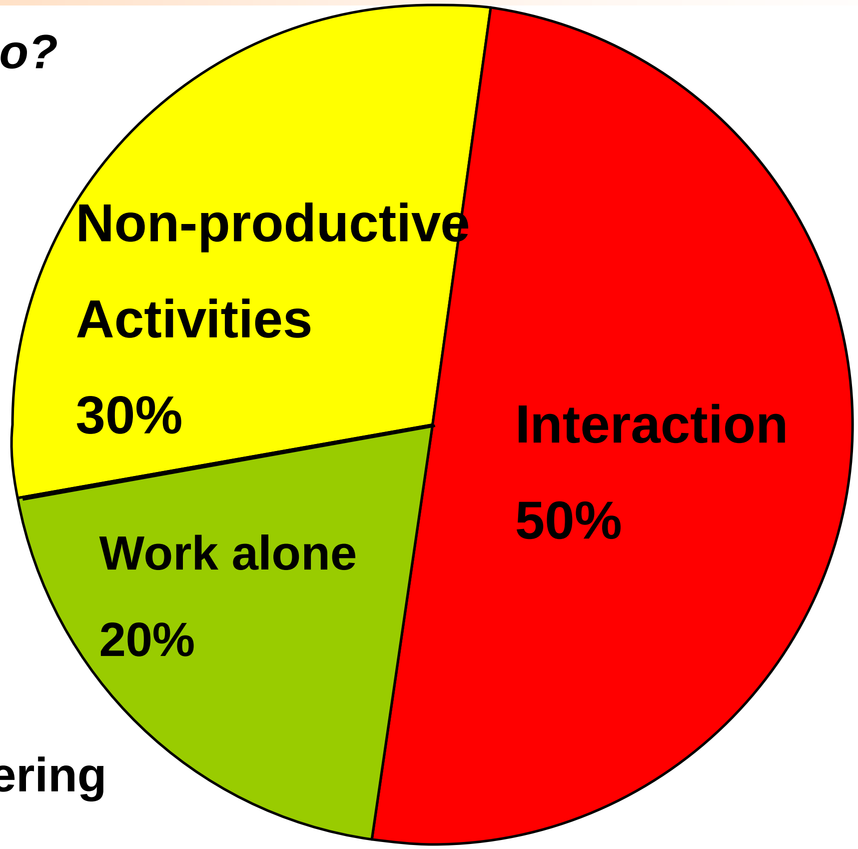
Project Impaired Factors

1. Incomplete Requirements
2. Lack of User Involvement
3. Lack of Resources
4. Unrealistic Expectations
5. Lack of Executive Support
6. Changing Requirements & Spec
7. Lack of Planning
8. Didn't Need It Any Longer
9. Lack of IT Management
10. Technology Illiteracy

"The definition of insanity is doing the same thing over and over again and expecting a different result." [Albert Einstein]

Why We Need Software Engineering?

What do software engineers do?

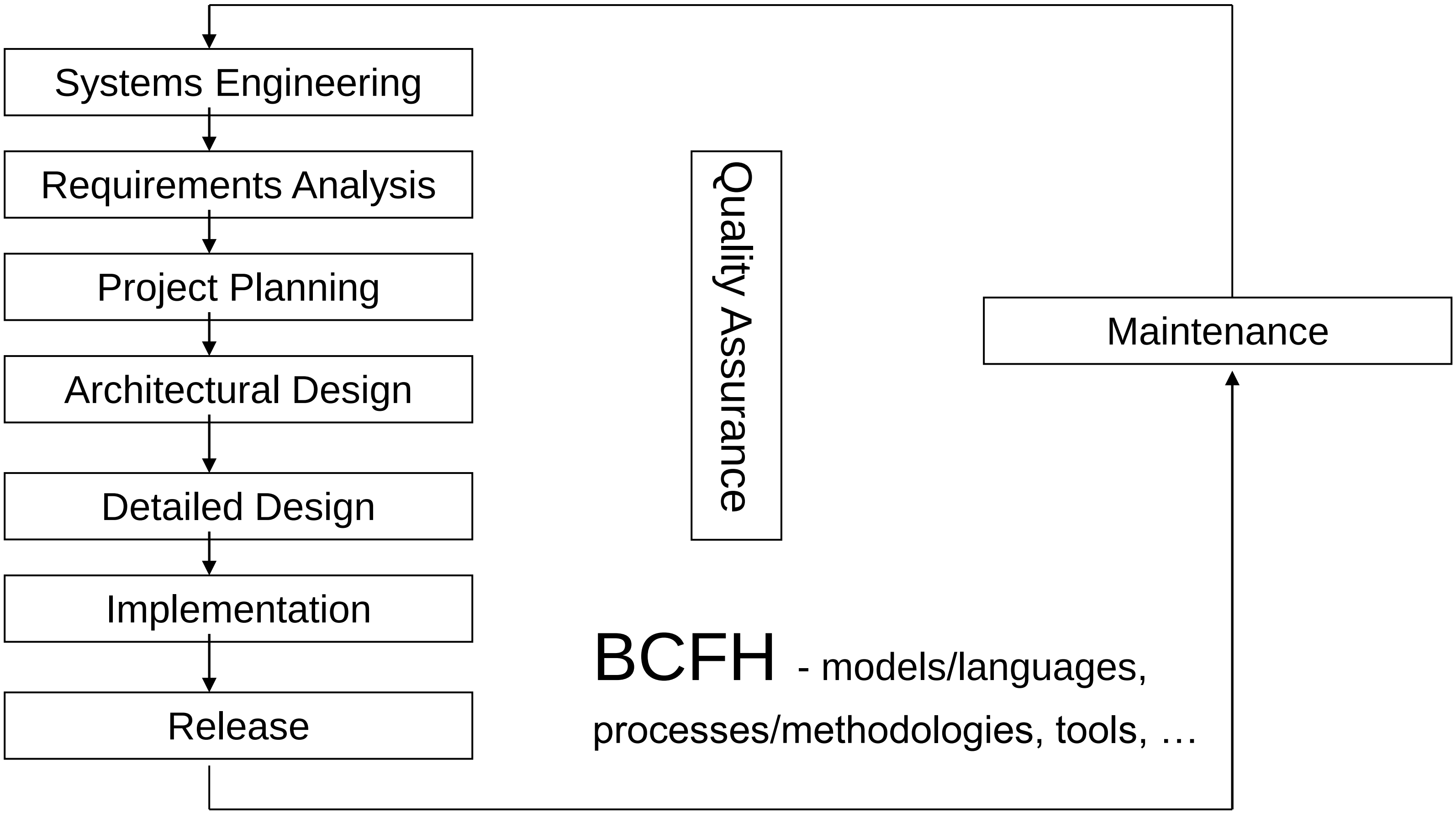


programming ≠ software engineering

- **personal activity** — **team activity**
- **small, clear problem** — **large, nebulous problem**

How to Do Software Engineering?

Software Lifecycle Review

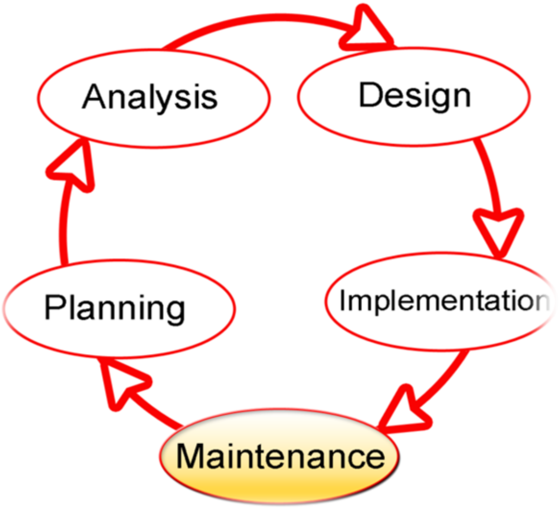


BCFH - models/languages, processes/methodologies, tools, …

SDLC and Process Model

Software Development Life Cycle (SDLC)

Set of activities and their relationships to each other to support the development of a software system.



Software Development Activities

1. Gathering Requirements
2. Team Management
3. Software Design
4. Coding
5. Testing
6. Documentation
7. Software Maintenance

Definition of Generic Process Model

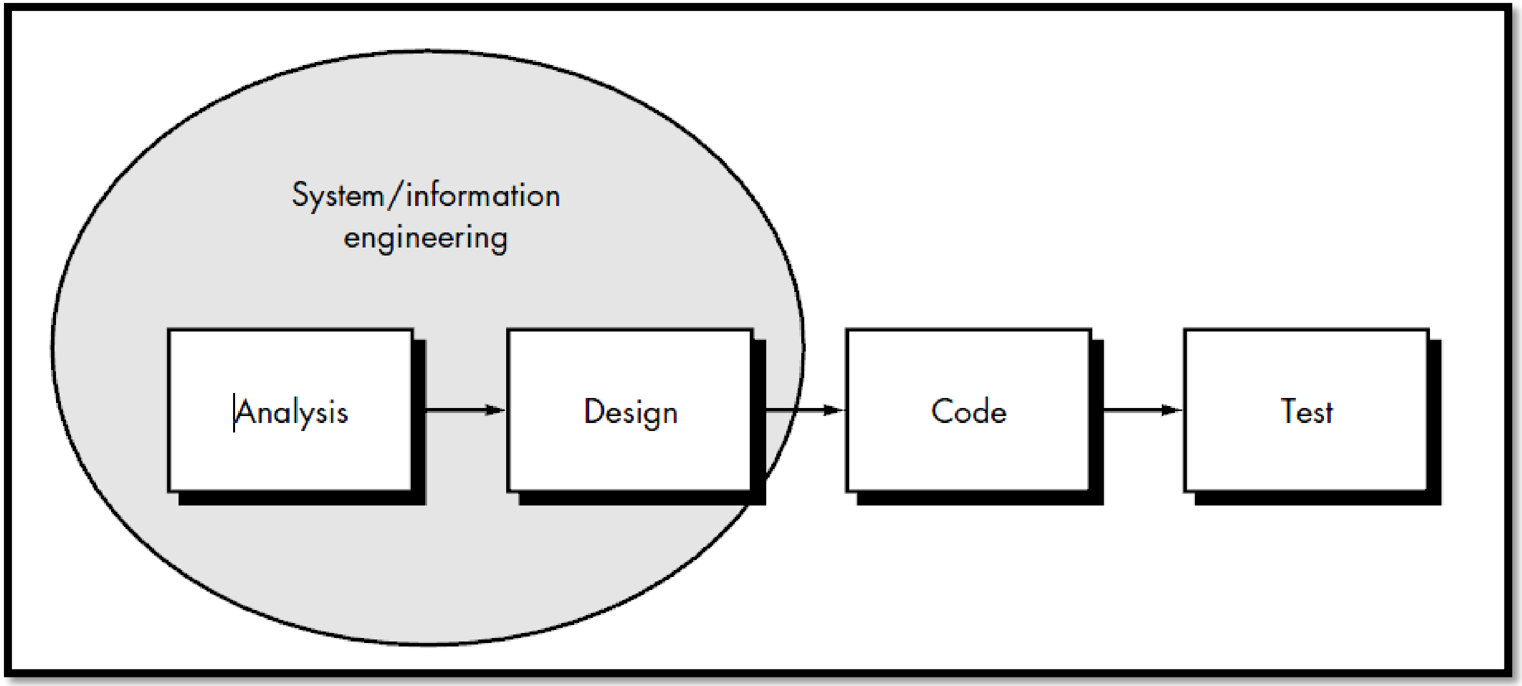
Consist of five general activities in software development, such as:

1. Communication
2. Planning
3. Modeling
4. Construction
5. Deployment.

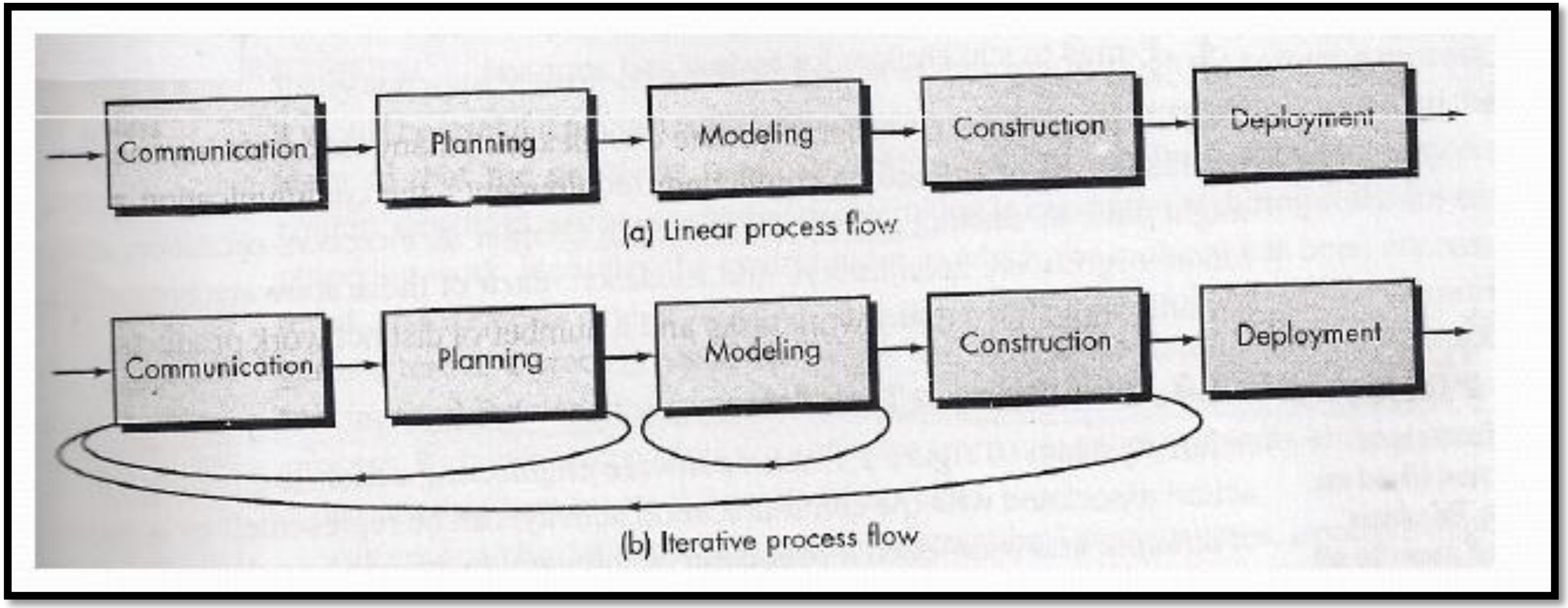
Process Flow

- Linear
- Iterative
- Evolutionary

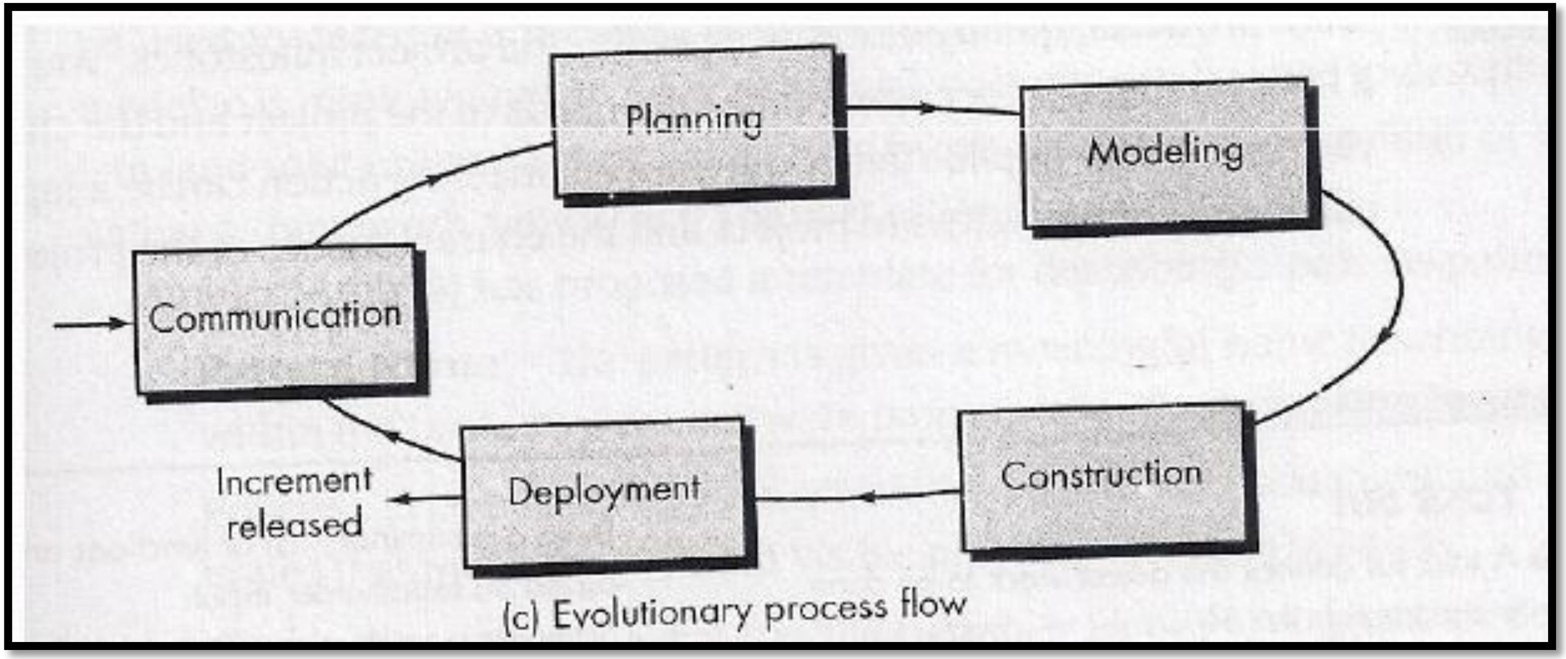
Linear Process Flow



Iterative Process Flow



Evolutionary Process Flow



Prescriptive Process Model

- Waterfall Model
- V Model
- Incremental Process Model
- Evolutionary Process Model
- Specialized Process Model
- Unified Process
- Agile Methods (example: XP)

Waterfall Model

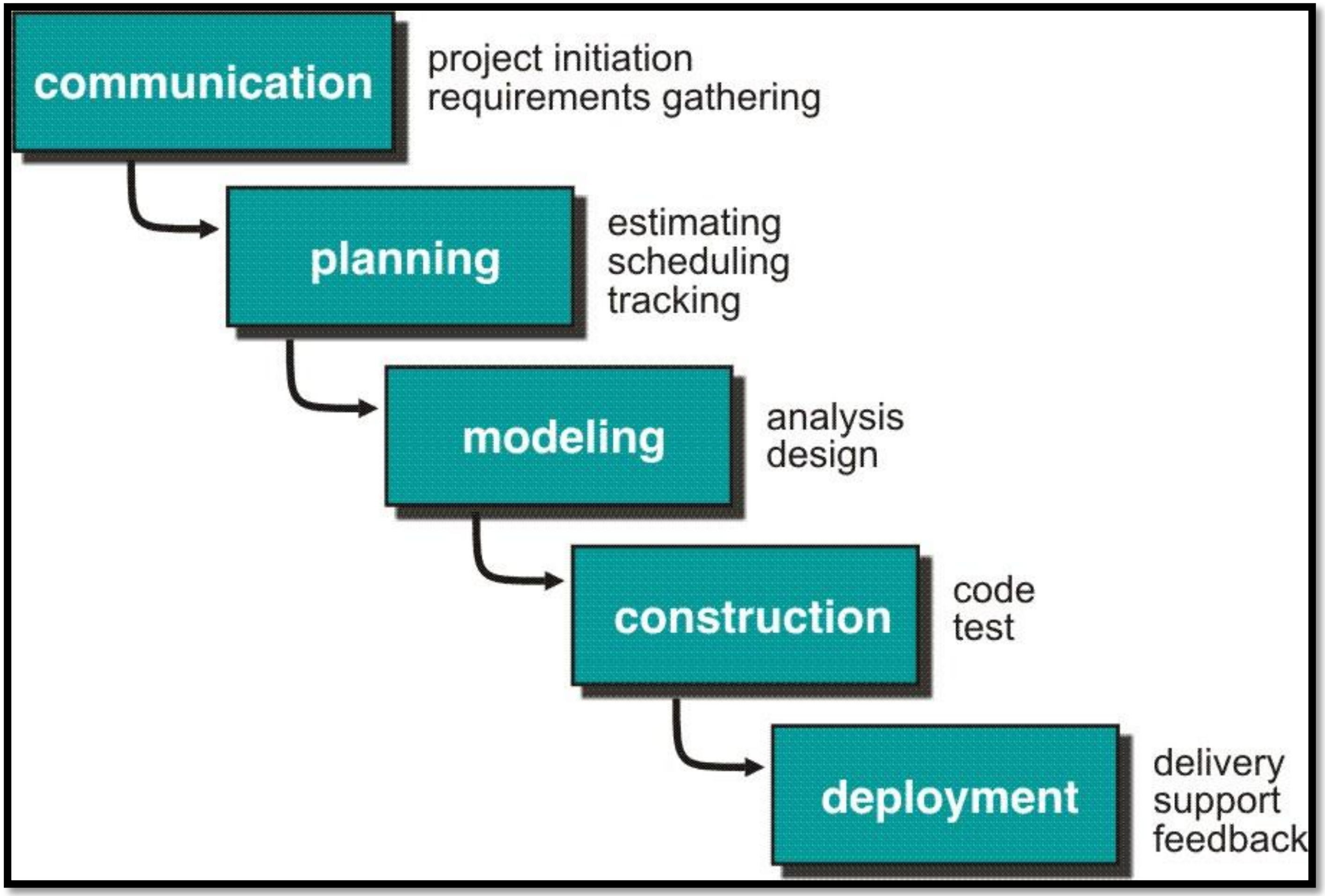
1. The requirements are knowable in advance of implementation.
2. The requirements have no unresolved, high-risk implications

   e.g., risks due to COTS choices, cost, schedule, performance, safety, security, user interfaces, organizational impacts
3. The nature of the requirements will not change very much

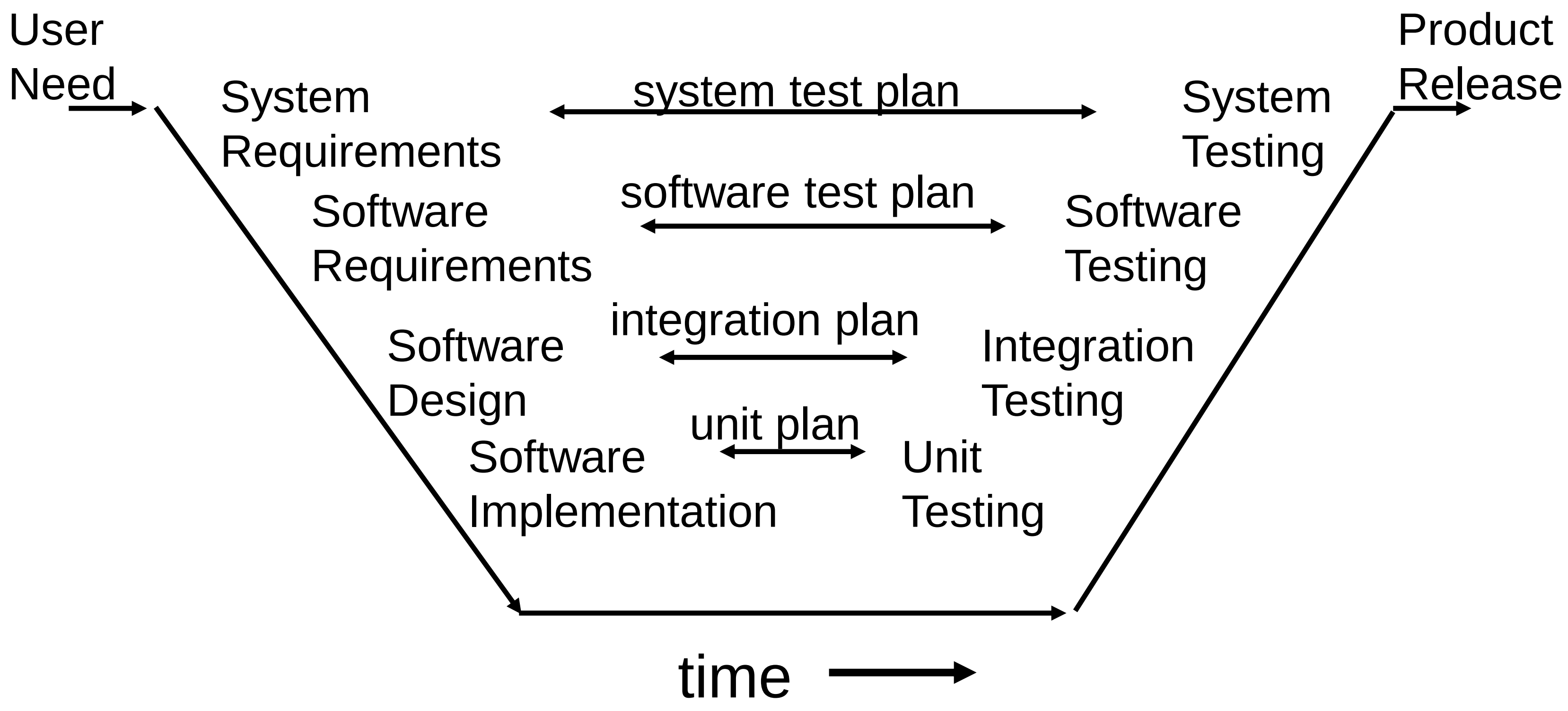
   During development; during evolution
4. The requirements are compatible with all the key system stakeholders' expectations

   e.g., users, customer, developers, maintainers, investors
5. The right architecture for implementing the requirements is well understood.
6. There is enough calendar time to proceed sequentially.

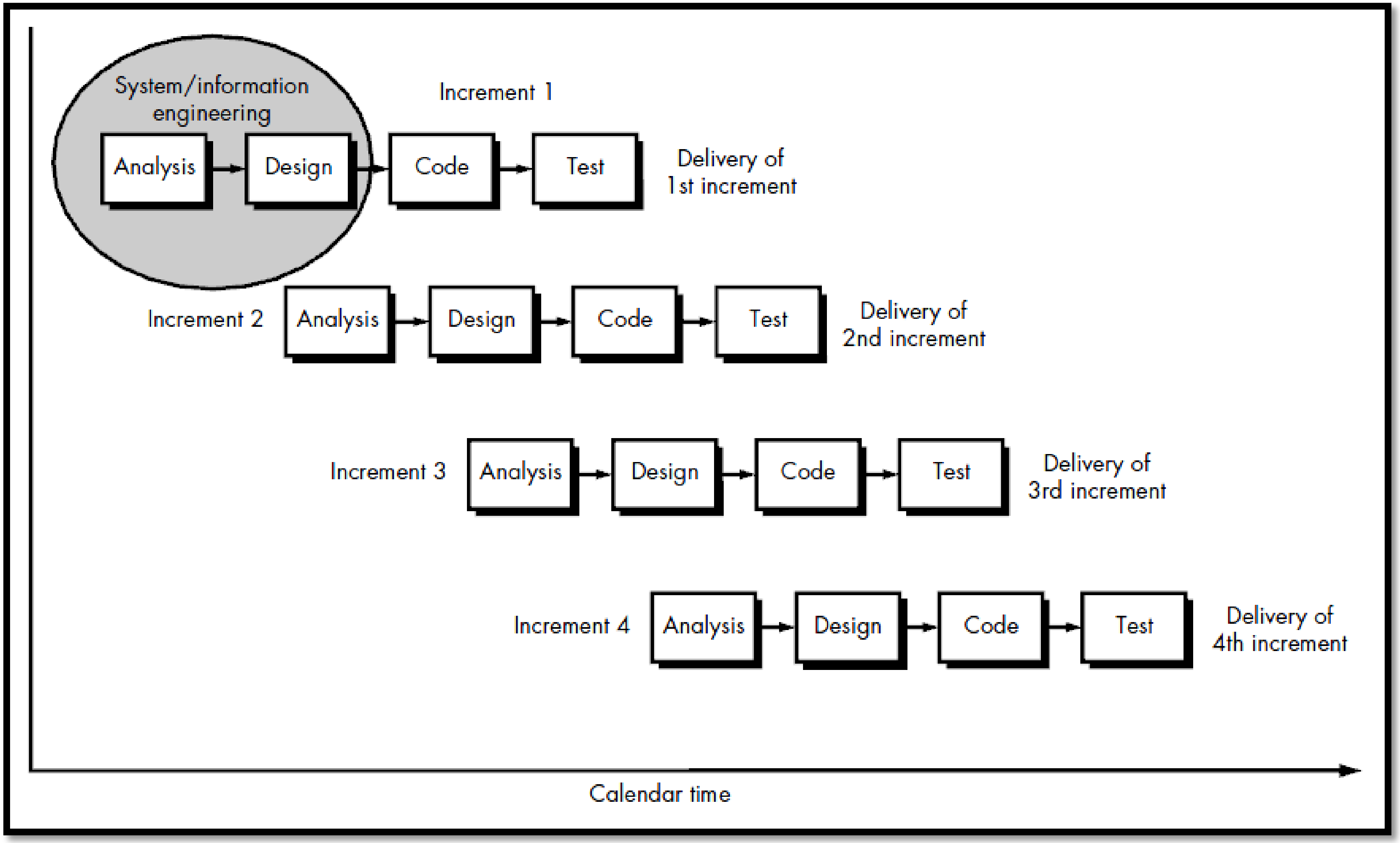
Waterfall Model



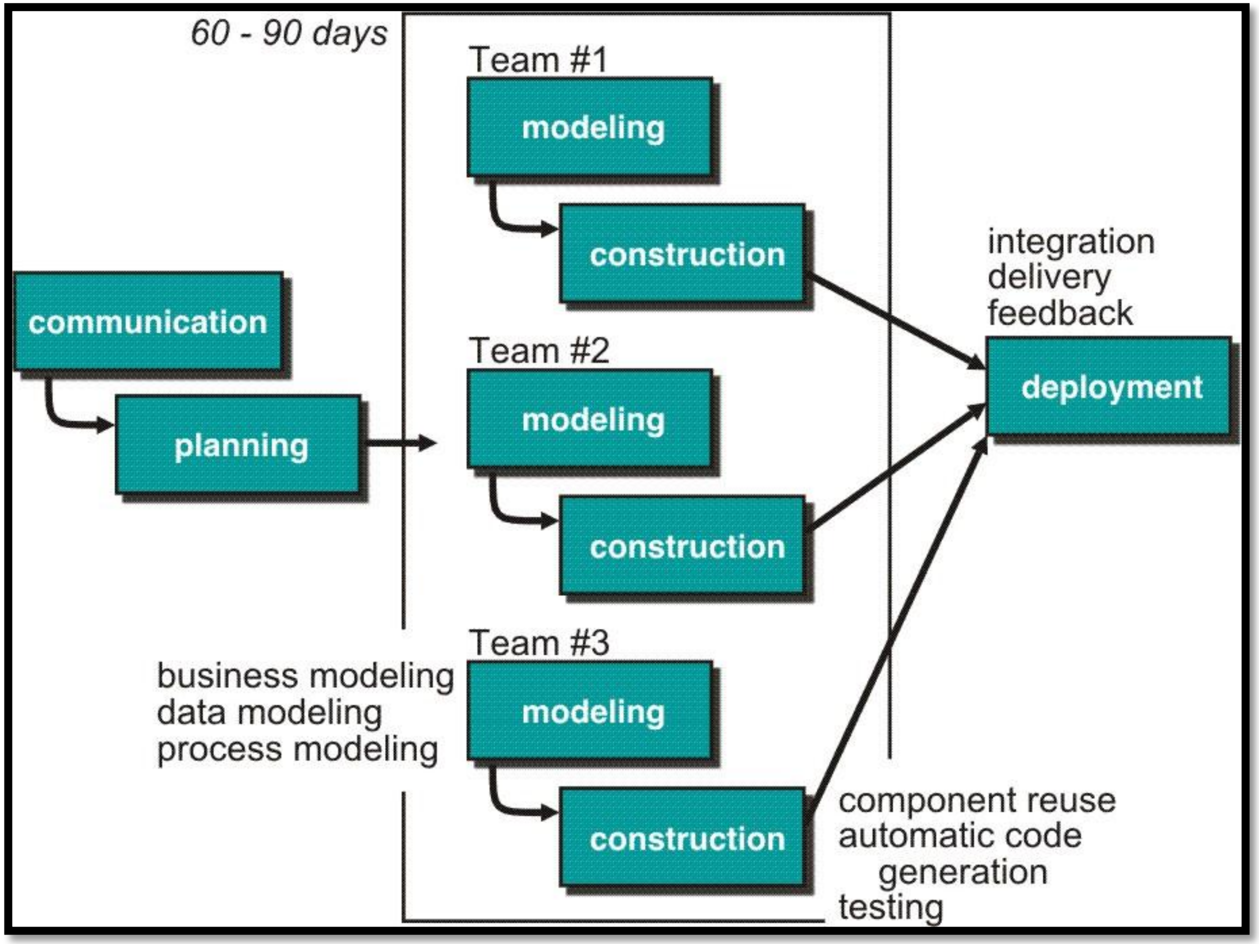
V Model


User Need
System Requirements
system test plan
System Testing
Product Release
Software Requirements
software test plan
Software Testing
Software Design
integration plan
Integration Testing
Software Implementation
unit plan
Unit Testing
time

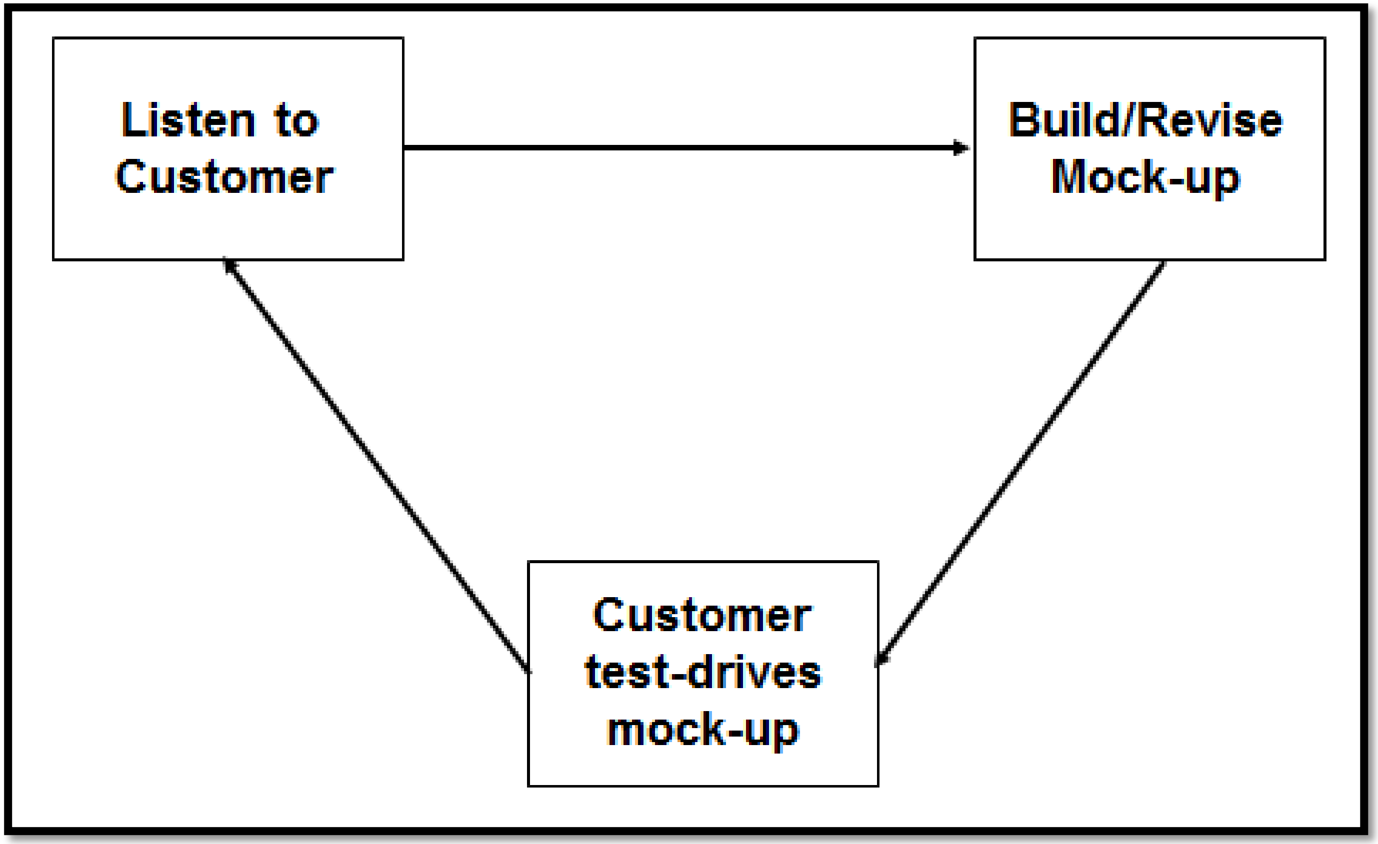
Incremental Process Model: Incremental



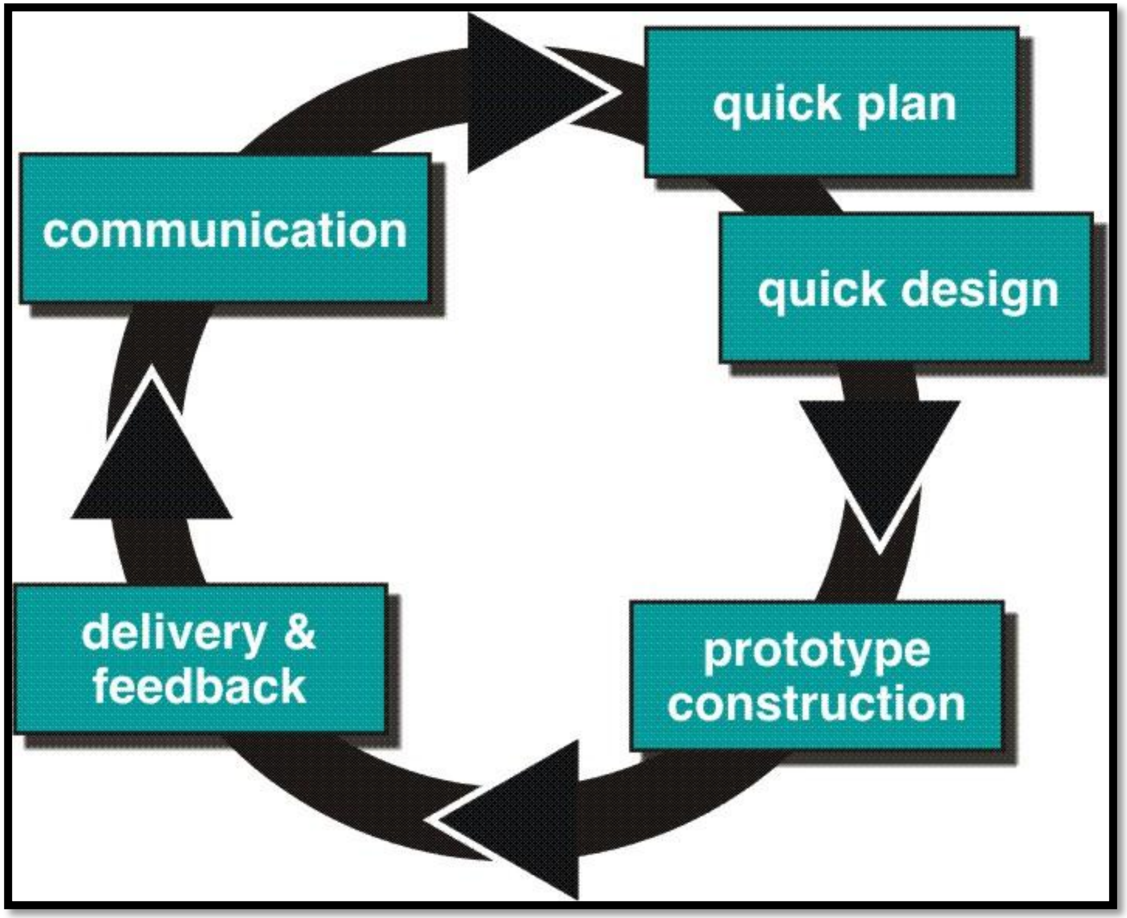
Incremental Process Model: RAD



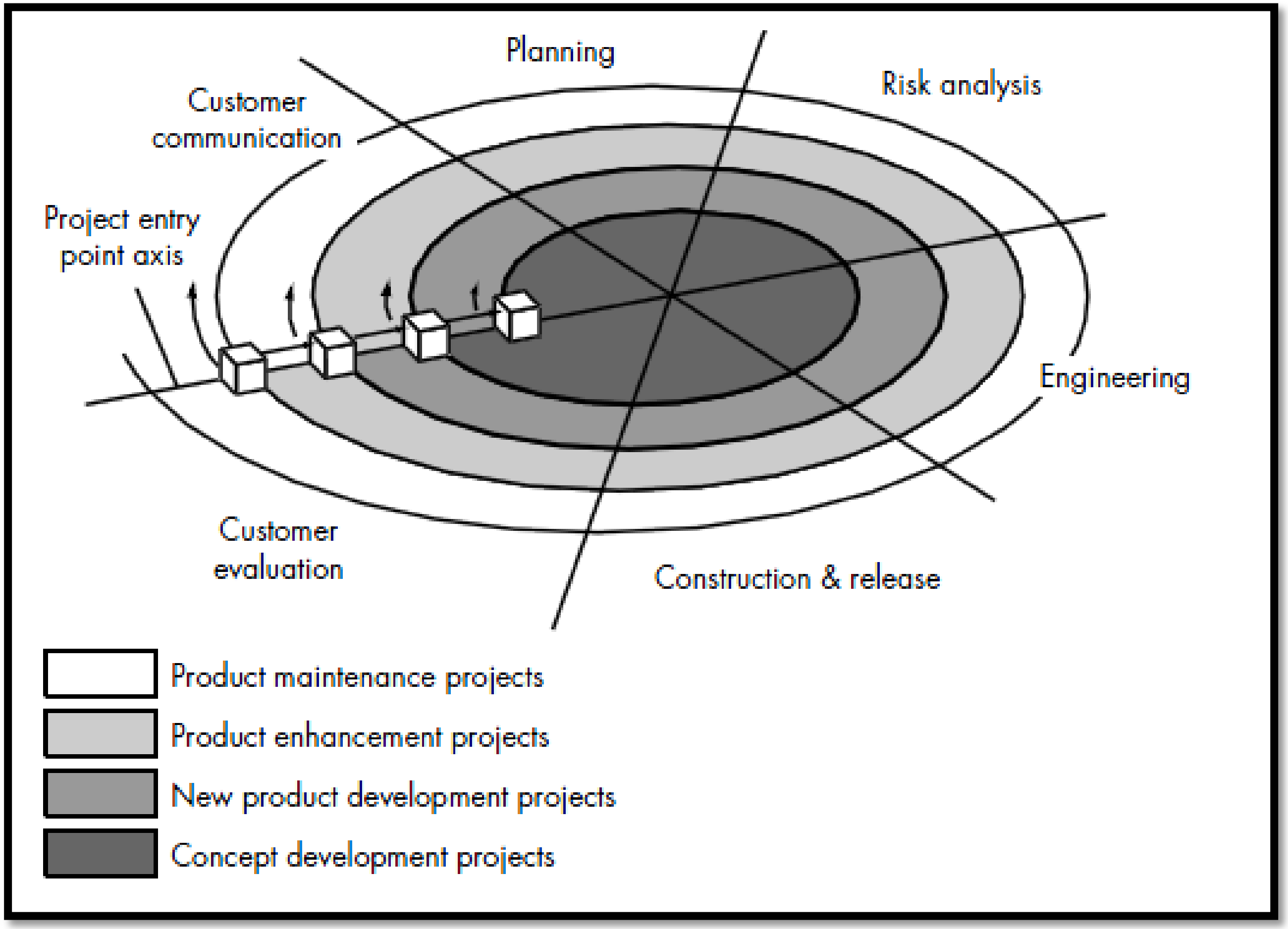
Evolutionary Process Model: Prototyping



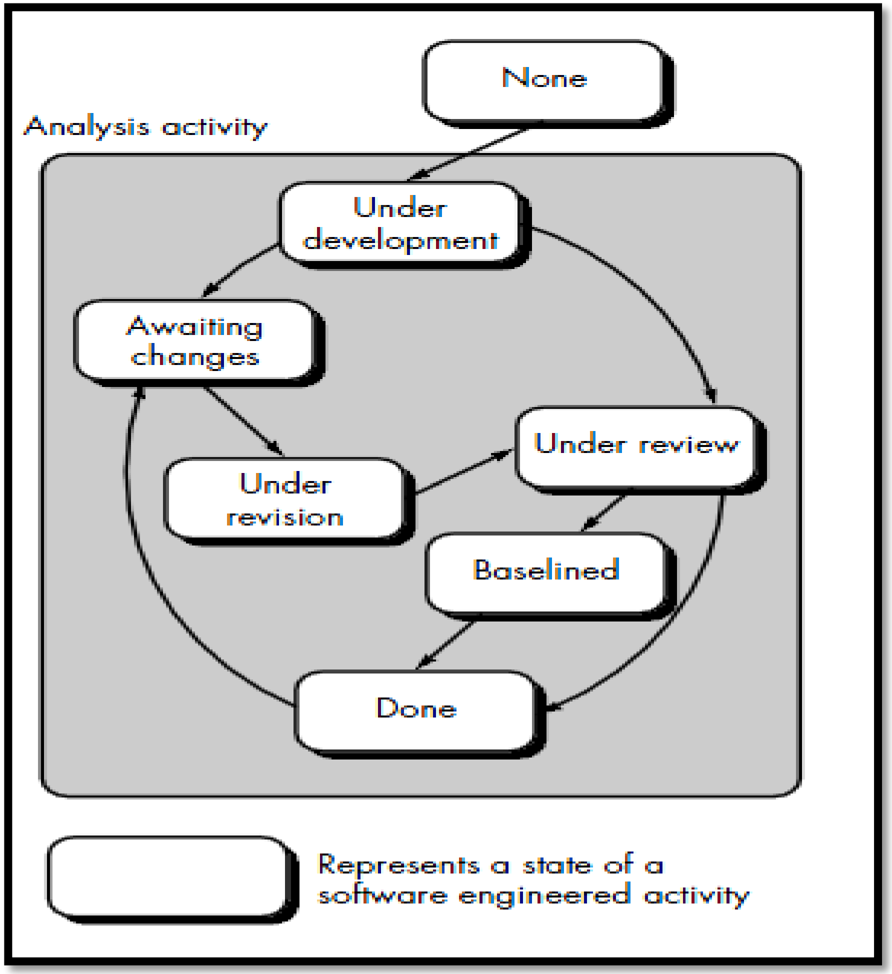
Evolutionary Process Model: Prototyping



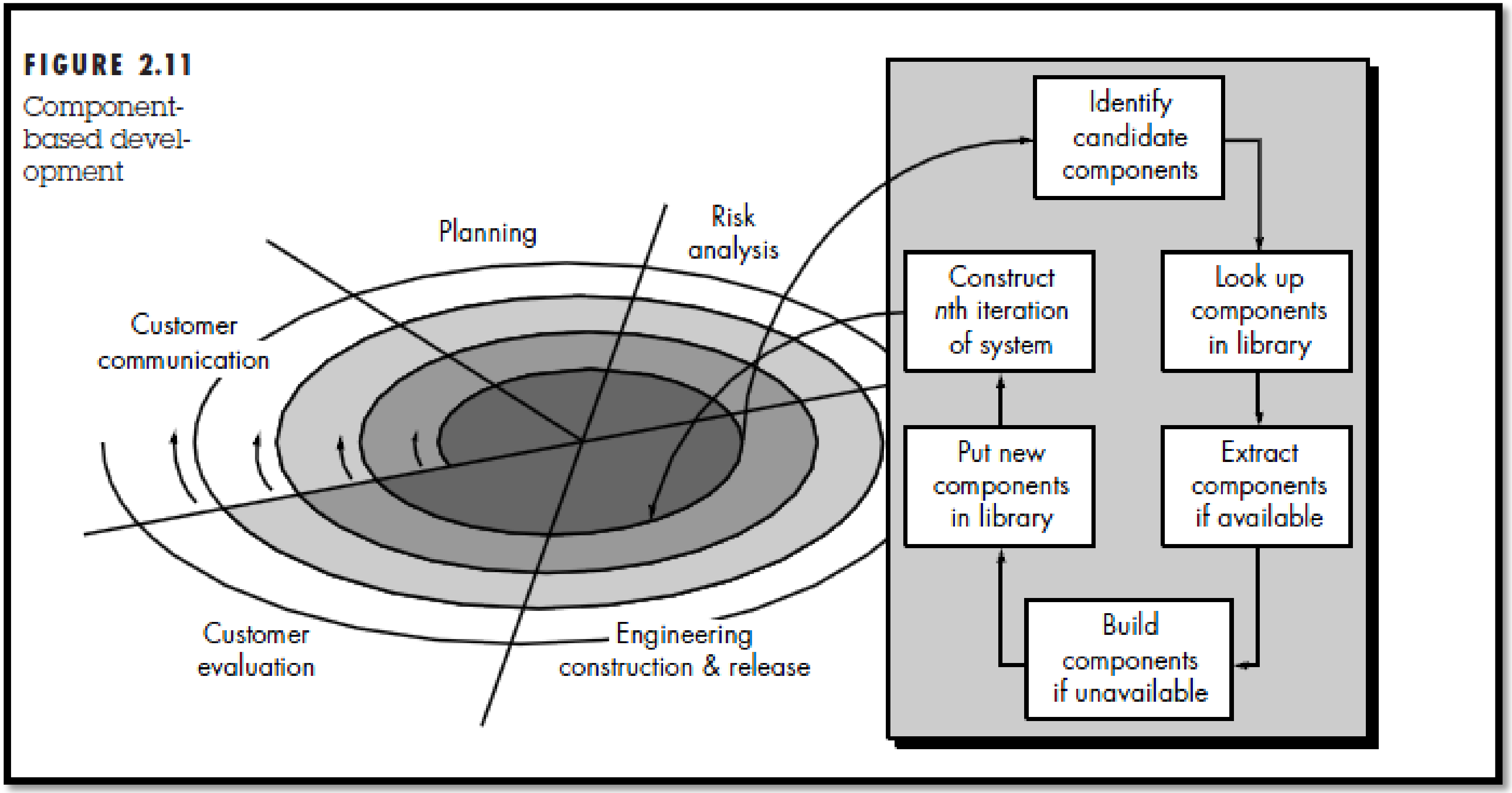
Evolutionary Process Model: Spiral



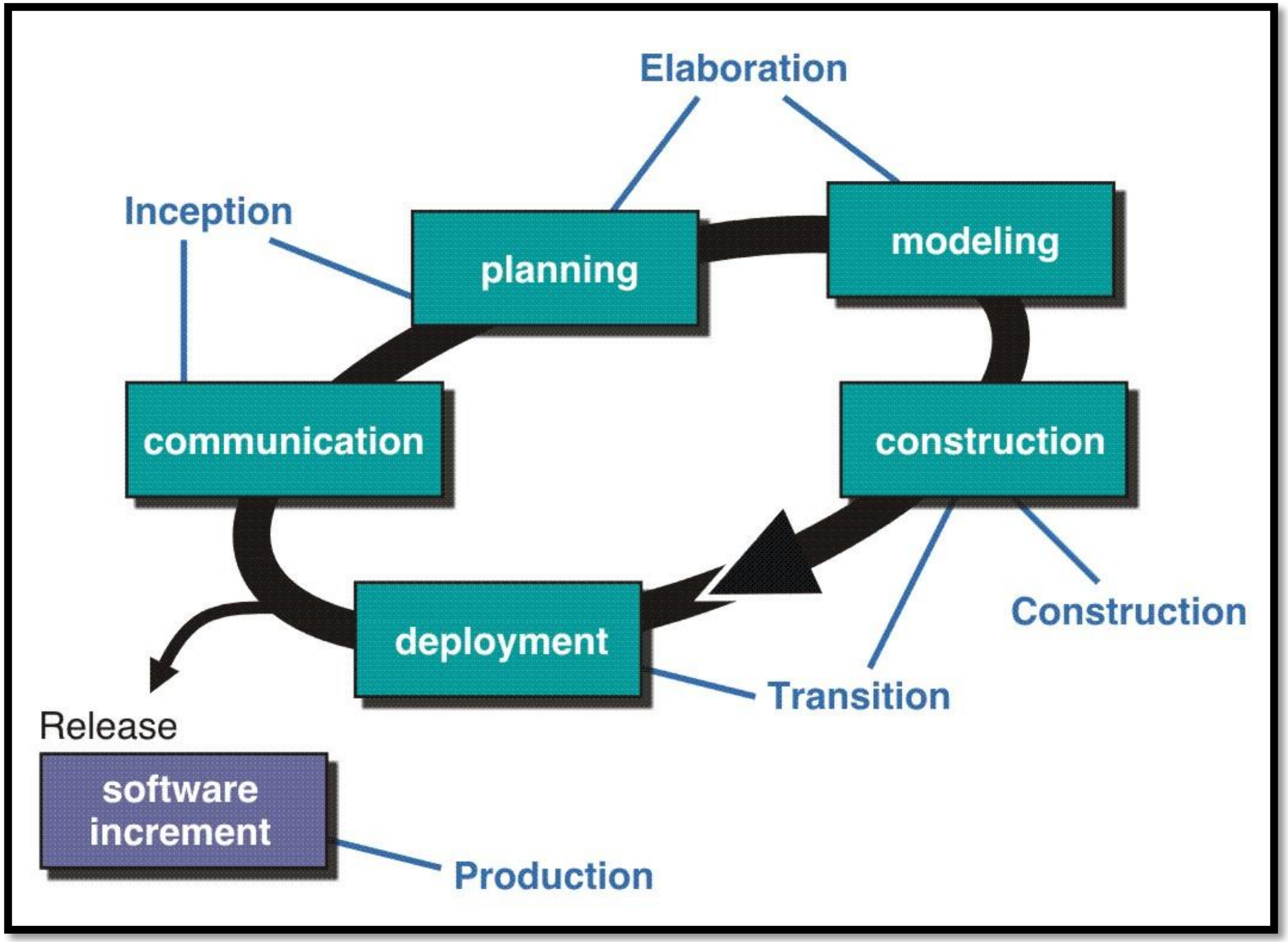
Evolutionary Process Model: Concurent



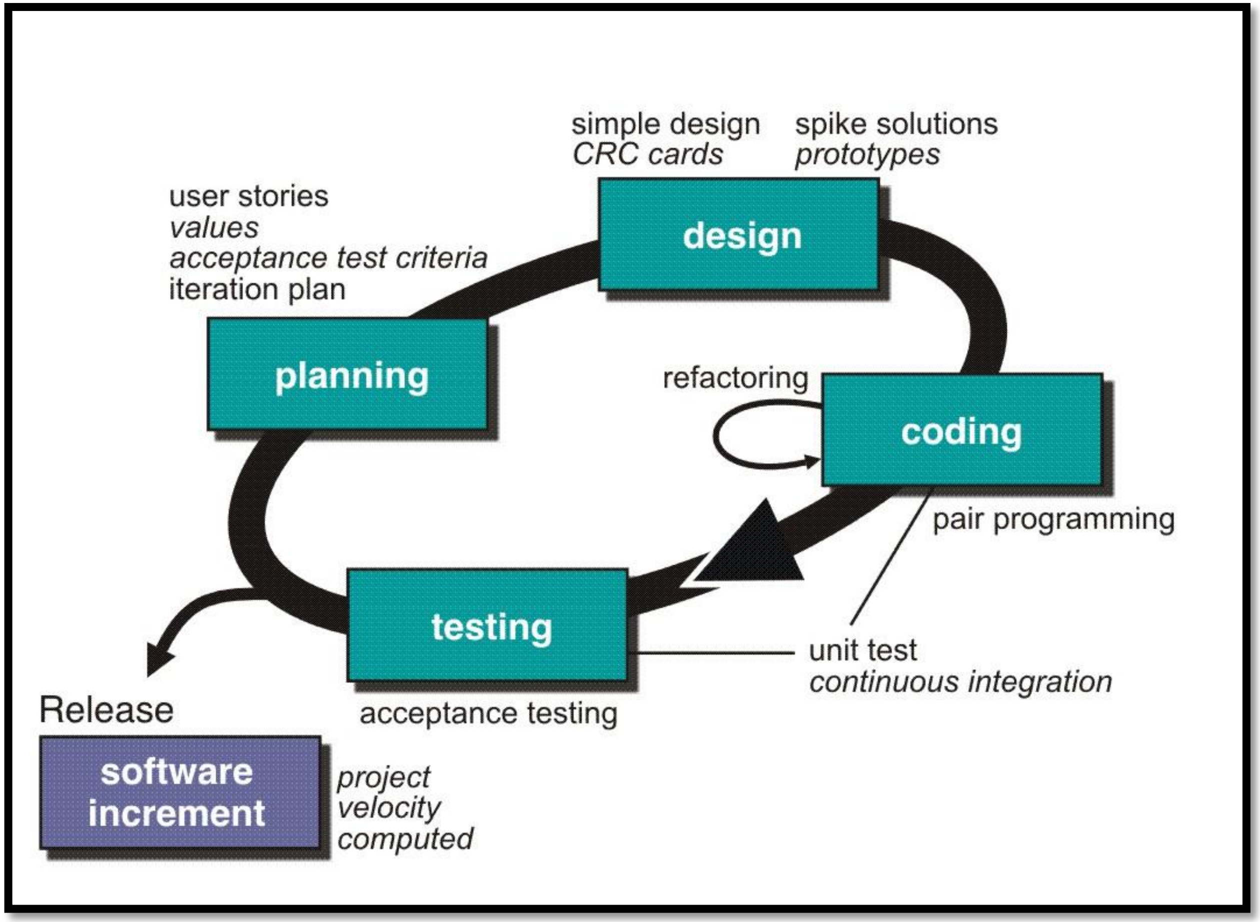
Specialized Process Model: Component Based Development



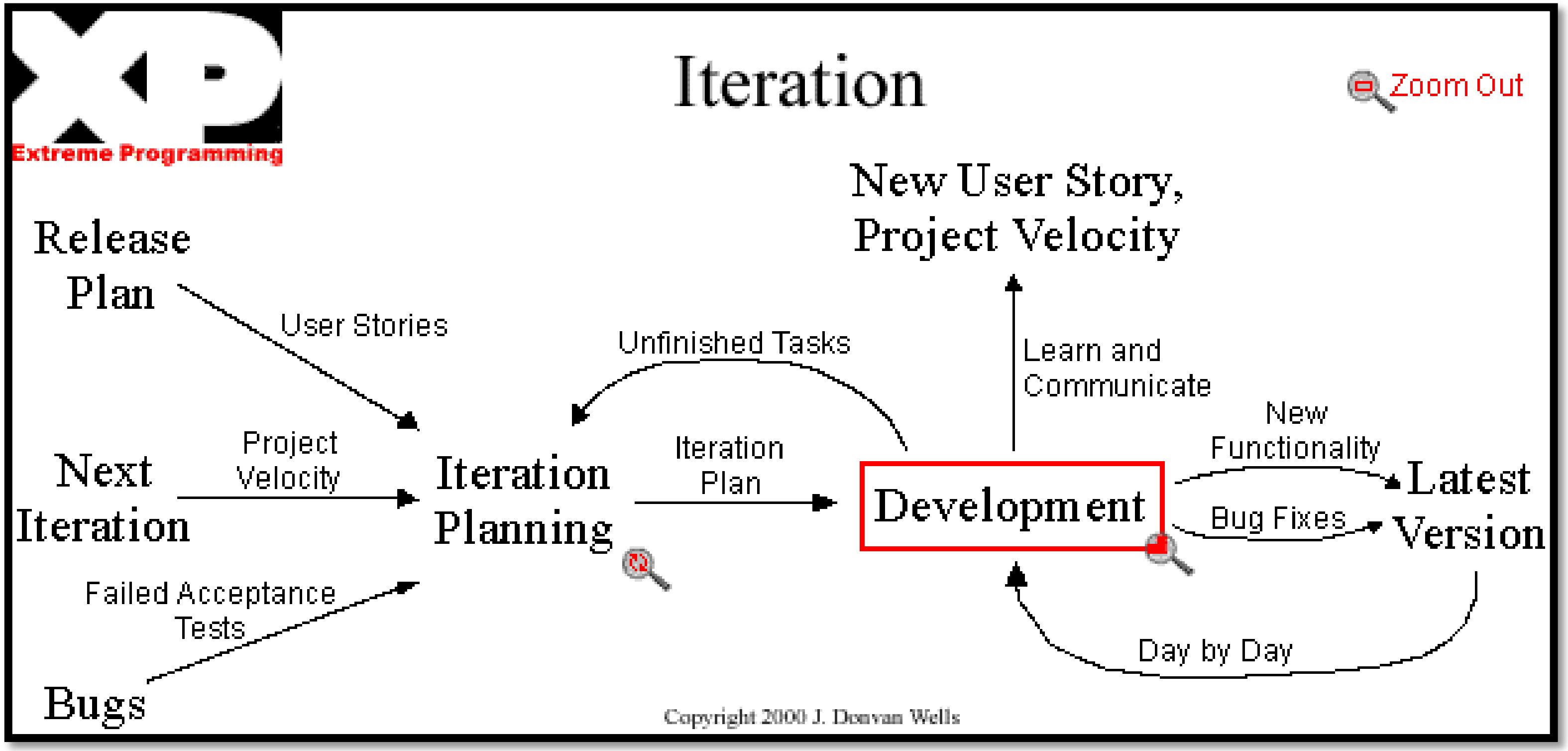
Specialized Process Model: Unified Process



Specialized Process Model: Agile Method (example: Extreme Programming)



Specialized Process Model: Agile Method (example: Extreme Programming)

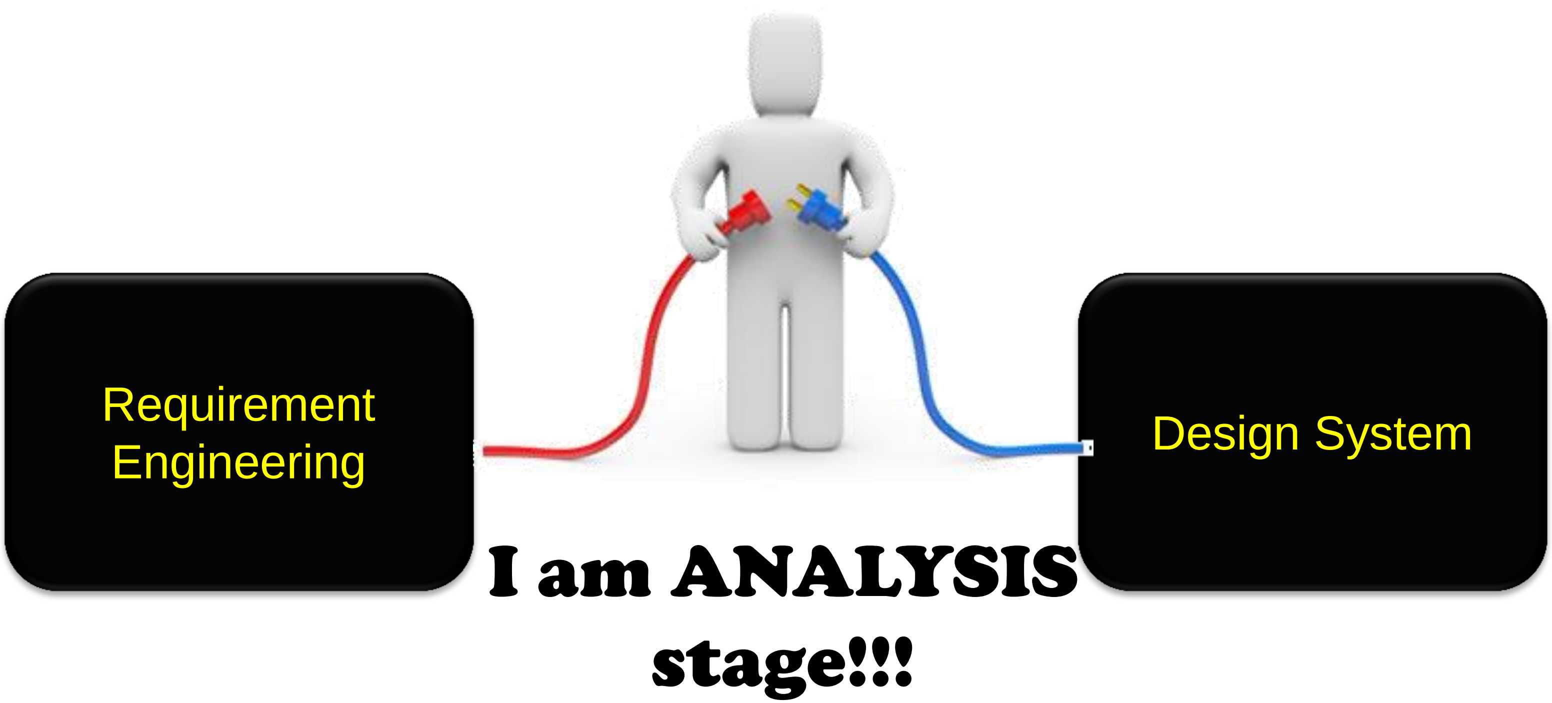


Analysis and Its Principles

What Is Requirement Analysis?

- Requirements analysis
  - specifies software’s ***operational*** characteristics
  - indicates software's ***interface*** with other system elements
  - establishes ***constraints*** that software must meet
- Requirements analysis allows the software engineer (called an *analyst* or *modeler* in this role) to:
  - ***elaborate*** on basic requirements established during earlier requirement engineering tasks
  - build ***models*** that depict user scenarios, functional activities, problem classes and their relationships, system and class behavior, and the flow of data as it is transformed.

What Is Requirement Analysis?



What Is Requirement Analysis?

FOCUS ON WHAT

NOT HOW!!!!

Steps in Requirement Analysis

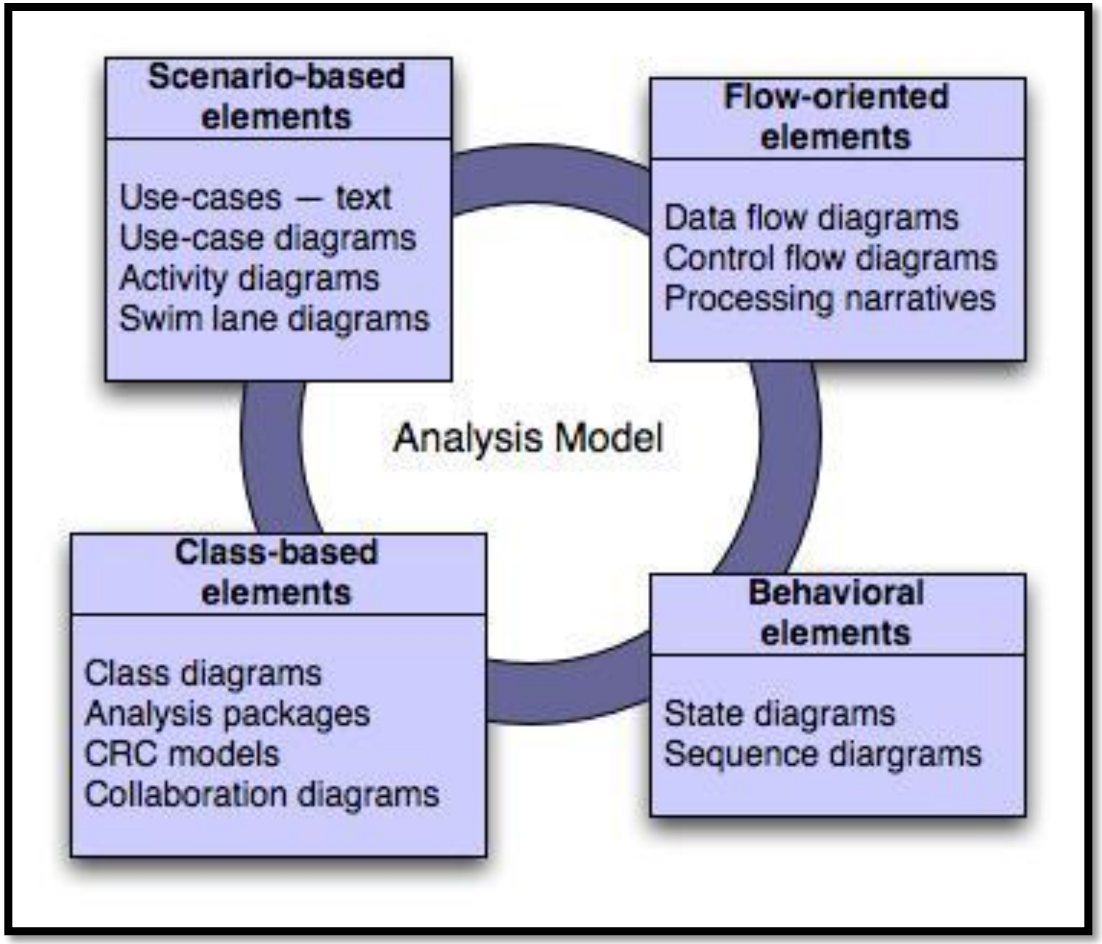
1. Identification
2. Understanding
3. Analysis
4. Reporting

Analysis Modelling Approach

1. Structured Analysis

2. Object-oriented analysis

Analysis Model



Domain Analysis

- Define the ***domain*** to be investigated.
- Collect a representative ***sample*** of applications in the domain.
- ***Analyze*** each application in the sample.
- Develop an analysis model for the ***objects***.
- In terms of data modeling, function/process modeling, behavioral modeling, etc.

Rules of Thumb Analysis

- Focus on requirements that are visible within the problem/business domain
- Each element of the reqs should add to an overall understanding of SW reqs and provide insight into the information domain, function, and behavior of the system
- Delay infrastructure and non functional models until design (bedakan dengan kebutuhan non fungsional)
- Minimize coupling
- Be certain that the reqs model provides value to all stakeholder
- Keep the model as simple as it can be

Data Modeling

- examines ***data objects*** independently of processing
- focuses attention on the ***data domain***
- creates a model at the ***customer***'s level of abstraction
- indicates how data objects ***relate*** to one another

Principles that Guide Practice

- Divide and conquer
- Understand the use of abstraction
- Strive for consistency
- Focus on the transfer of information
- Build software that exhibit effective modularity
- Look for patterns
- When possible, represent problems & solutions from Different perspectives
- Remember that someone will maintain the software

Principles that Guide Process

- Be agile
- Focus on quality in every step
- Be ready to a adapt
- Build an effective team
- Establish mechanisms for comm. & coordination
- Manage change
- Asses risk
- Create work products that provide value for others

Communication Principles (1)

- Listen
- Prepare before you communicate
- Someone should facilitate the activity
- Face-to-face comm. is the best
- Take notes and document decisions
- Strive for collaboration

Communication Principles (2)

- Stay focus; modularize your discussion
- If something is unclear, draw a picture
- (a) once you agree to something, move on. (b) if you can't agree to something, move on. (c) if a feature/function is unclear and cannot be clarified at the moment, move on
- Negotiation is not a game. It works best when both parties win

Modeling Principles (1)

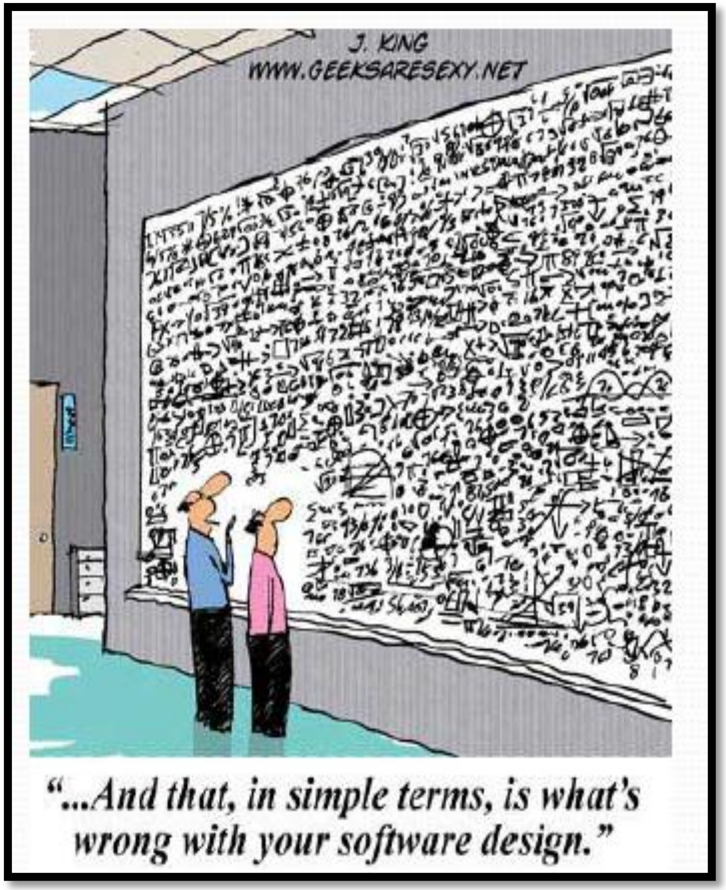
- The primary goal is to build software, not create models
- Don’t create more models than you need
- Strive to produce the simplest model
- Build models in a way that makes them amenable (menerima) to change
- Be able to state an explicit purpose for each model that is created
- Adapt the models you develop to the system at hand

Modeling Principles (2)

- Try to build useful models, but forget about building perfect models
- Don’t become dogmatic about the syntax of the model. If it communicates content succesfully, representation is secondary
- If your instincts tell you a model isn’t right eventhough it seems okay on paper, you probably have reason to be concerned
- Get feedback as soon as you can

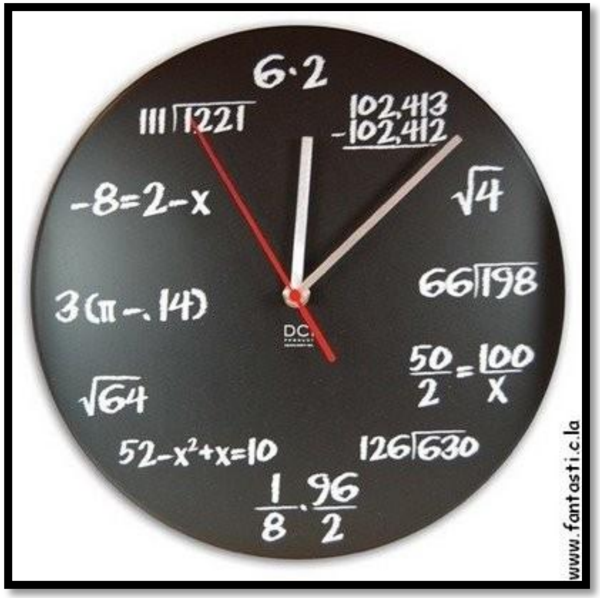
Design and Its Principles

Bad Design



J. KING
WWW.GEEKSARESEXY.NET
"...And that, in simple terms, is what's wrong with your software design."

NO SMOKING
IN THIS AREA
COMPLIES WITH ALL ORDINANCES & STATUTES



6·2
111|1221
102,413
-102,412
√4
66|198
50/2 = 100/X
126|630
1/8 · 96/2
52-x²+x=10
√64
3(π-.14)
-8=2-x
www.fantasti.c.la

Good Design

- Implement all of the explicit requirements
- Readable and understandable for coder/tester
- Provide a complete picture of the software:

  data, functional, behavior

But How to Make Good Design?

Good Design: Technical Criteria (1)

- Architecture: (1) recognizable styles, (2) good design characteristic component, (3) can be implemented in the evolutionary fashion
- Modular
- Contain distinct representation of data, architecture, interfaces, and components
- Data structures recognizable data patterns

Good Design: Technical Criteria (2)

- Components with independent functional characteristics
- Interfaces to simplify connection between components and the external environment
- Repeatable method and is derived by information from analysis
- Notation effectively communicate its meaning

Design Principles (1)

- The design process should not suffer from ‘tunnel vision.’
- The design should be traceable to the analysis model.
- The design should not reinvent the wheel.
- The design should “minimize the intellectual distance” [DAV95] between the software and the problem as it exists in the real world.
- The design should exhibit uniformity and integration.
- The design should be structured to accommodate change.

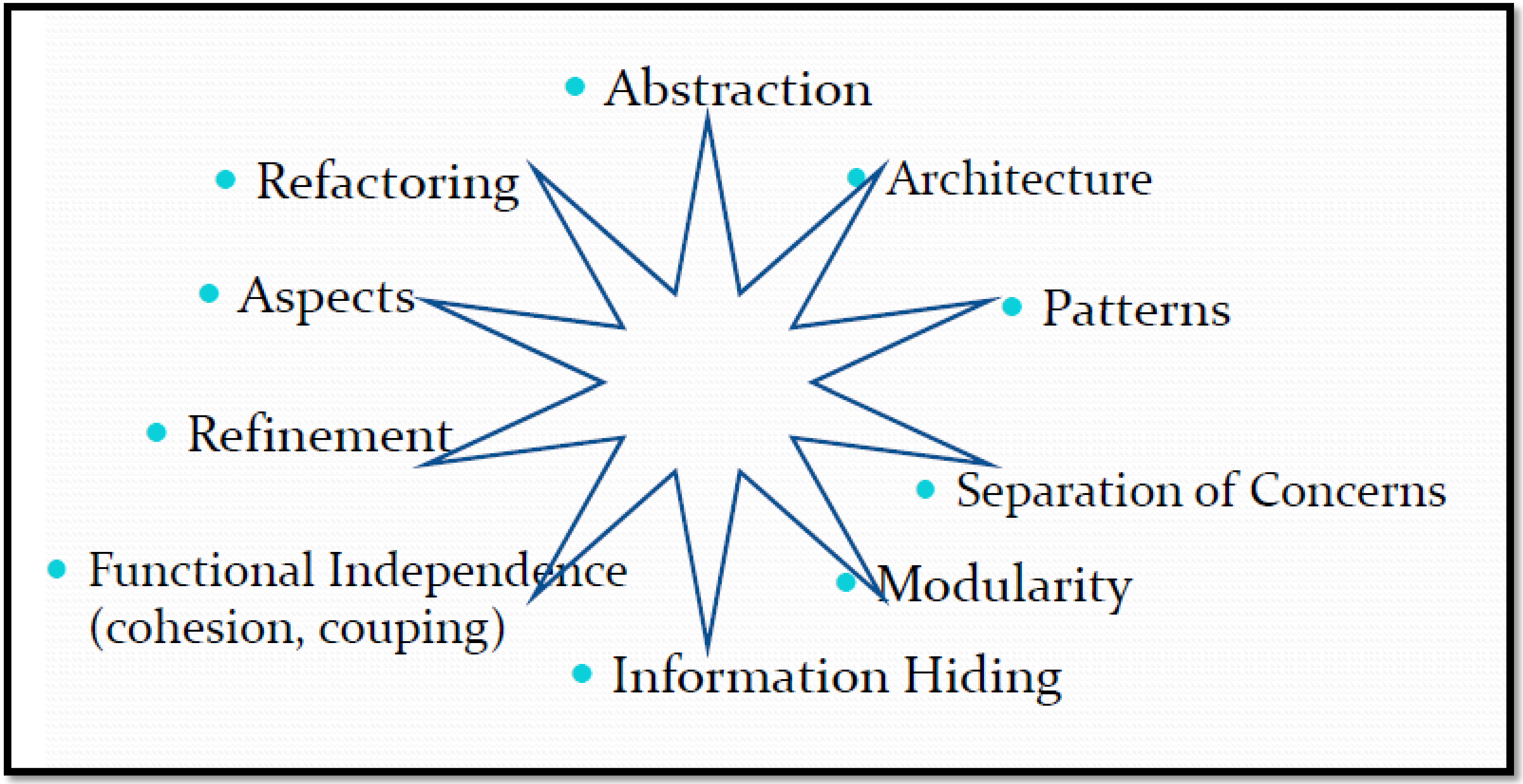
Design Principles (2)

- The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered.
- Design is not coding, coding is not design.
- The design should be assessed for quality as it is being created, not after the fact.
- The design should be reviewed to minimize conceptual (semantic) errors.

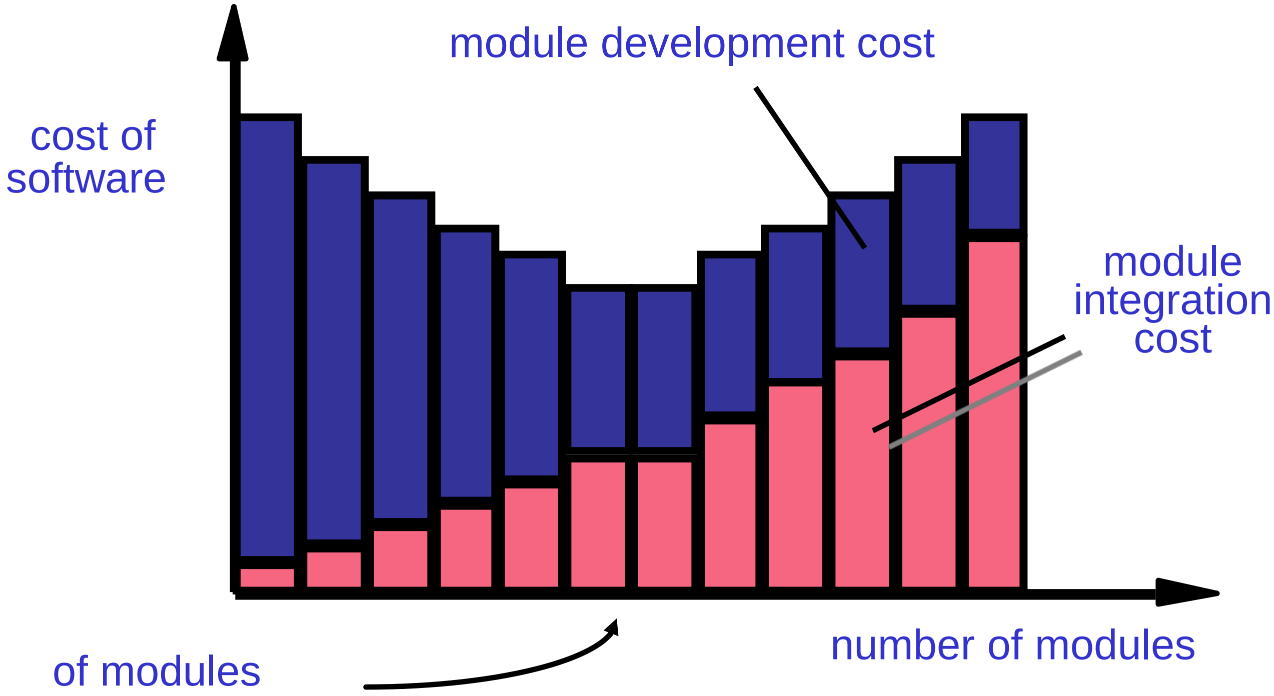
Design Concepts

- abstraction — data, procedure, control
- architecture — the overall structure of the software
- patterns — “conveys the essence” of a proven design solution
- modularity — compartmentalization of data and function
- information hiding — controlled interfaces
- functional independence — high cohesion and low coupling
- refinement — elaboration of detail for all abstractions
- refactoring — improve design without effecting behavior

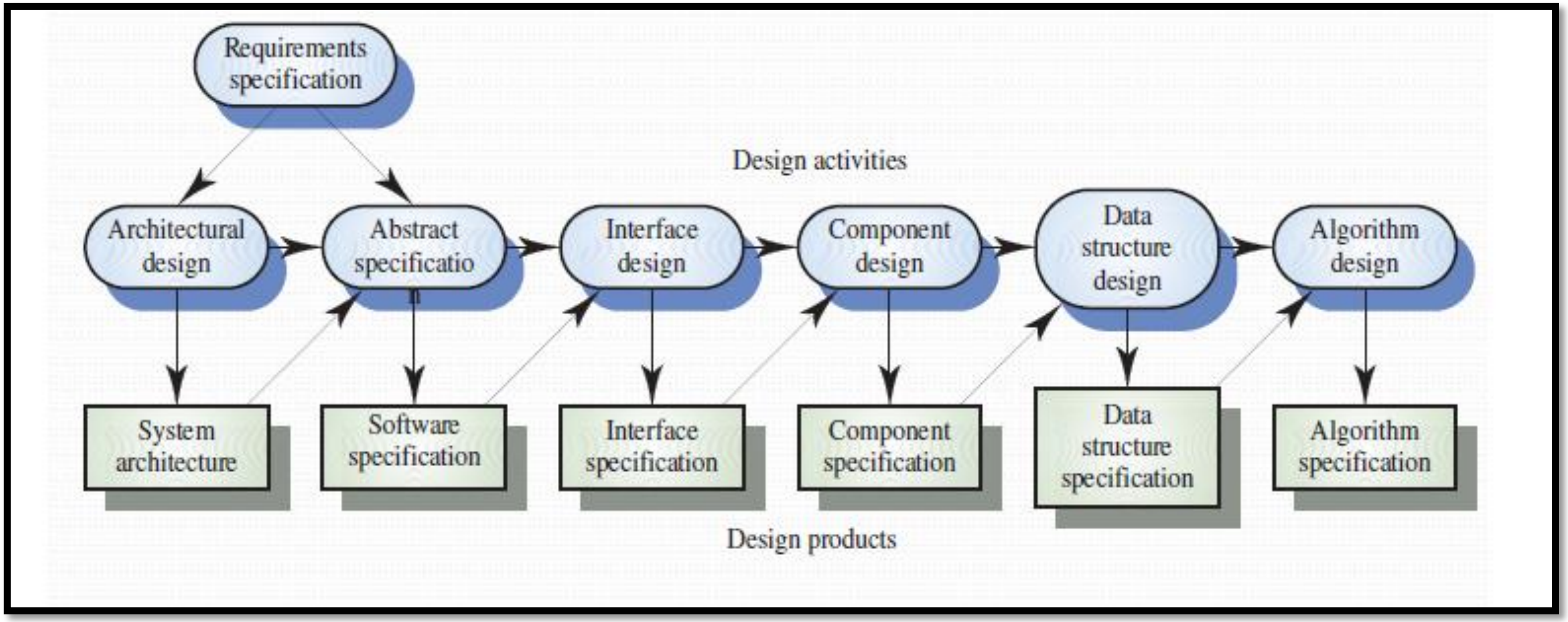
Design Concepts



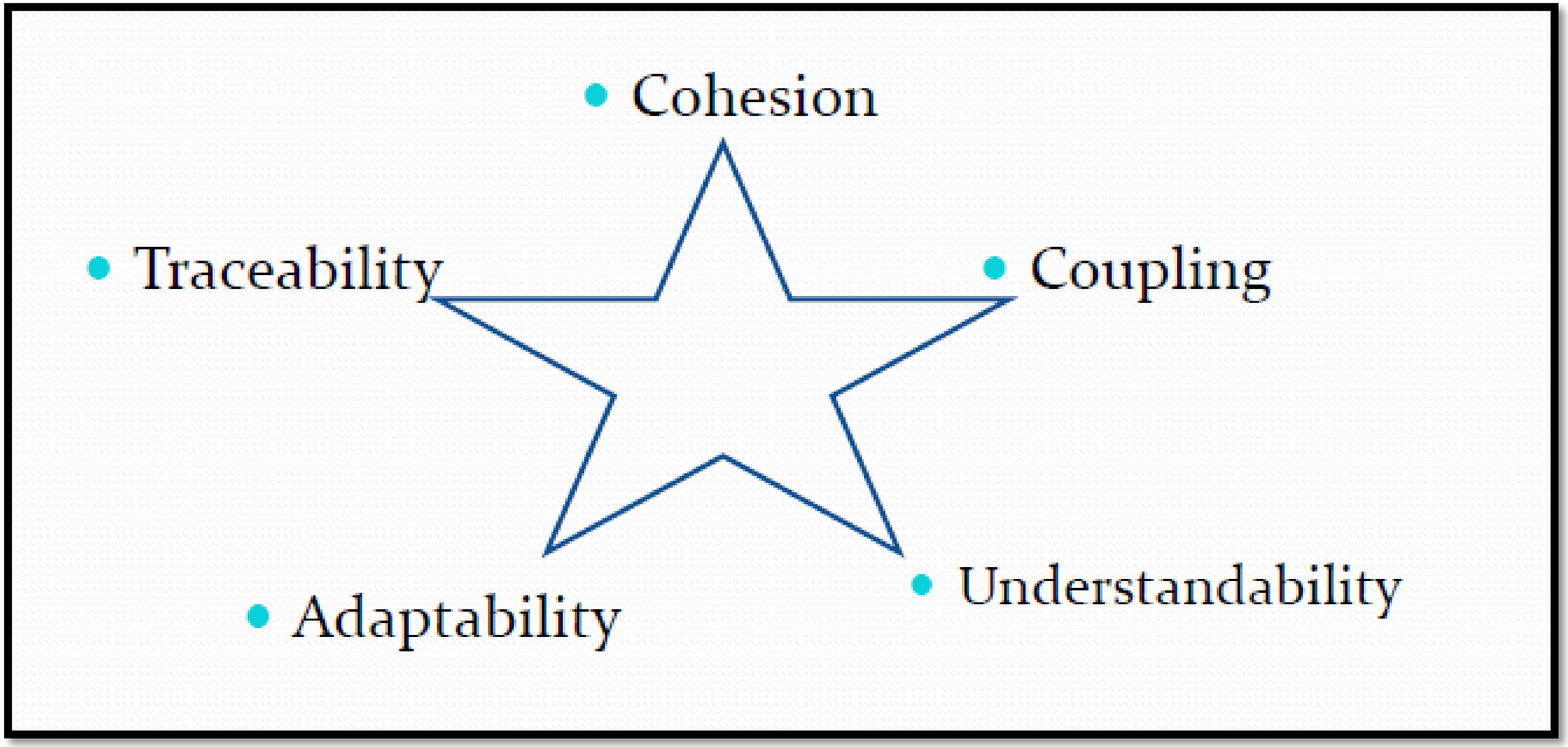
Modularity Trade Off



Phases in The Design Process



Design Quality Attributes



Cohesion

- A measure of how well a component 'fits together'
- A component should implement a single logical entity or function
- Cohesion is a desirable design component attribute as when a change has to be made, it is localized in a single cohesive component
- Various levels of cohesion have been identified

Cohesion Level (1)

- Coincidental cohesion (weak)
  - Parts of a component are simply bundled together
- Logical association (weak)
  - Components which perform similar functions are grouped
    - For example:
      - output text to screen
      - output line to printer
      - output record to file
    - Seems ok
    - Problem is it carries out a range of similar but different actions
    - No single well defined action

Cohesion Level (2)

- Temporal cohesion (weak)

  Components which are activated at the same time are grouped

  For example:

  - clear screen
  - open file
  - Initialise total

  again not related

  solution is to make initialization module all otherspecialised modules:

  - call init_terminal
  - call init_files
  - call init_calculations

Cohesion Level (3)

- Procedural cohesion (weak)

  The elements in a component make up a single control sequence

- Communicational cohesion (medium)

  All the elements of a component operate on the same data e.g. display and log temperature

- Sequential cohesion (medium)

  The output for one part of a component is the input to another part

Cohesion Level (4)

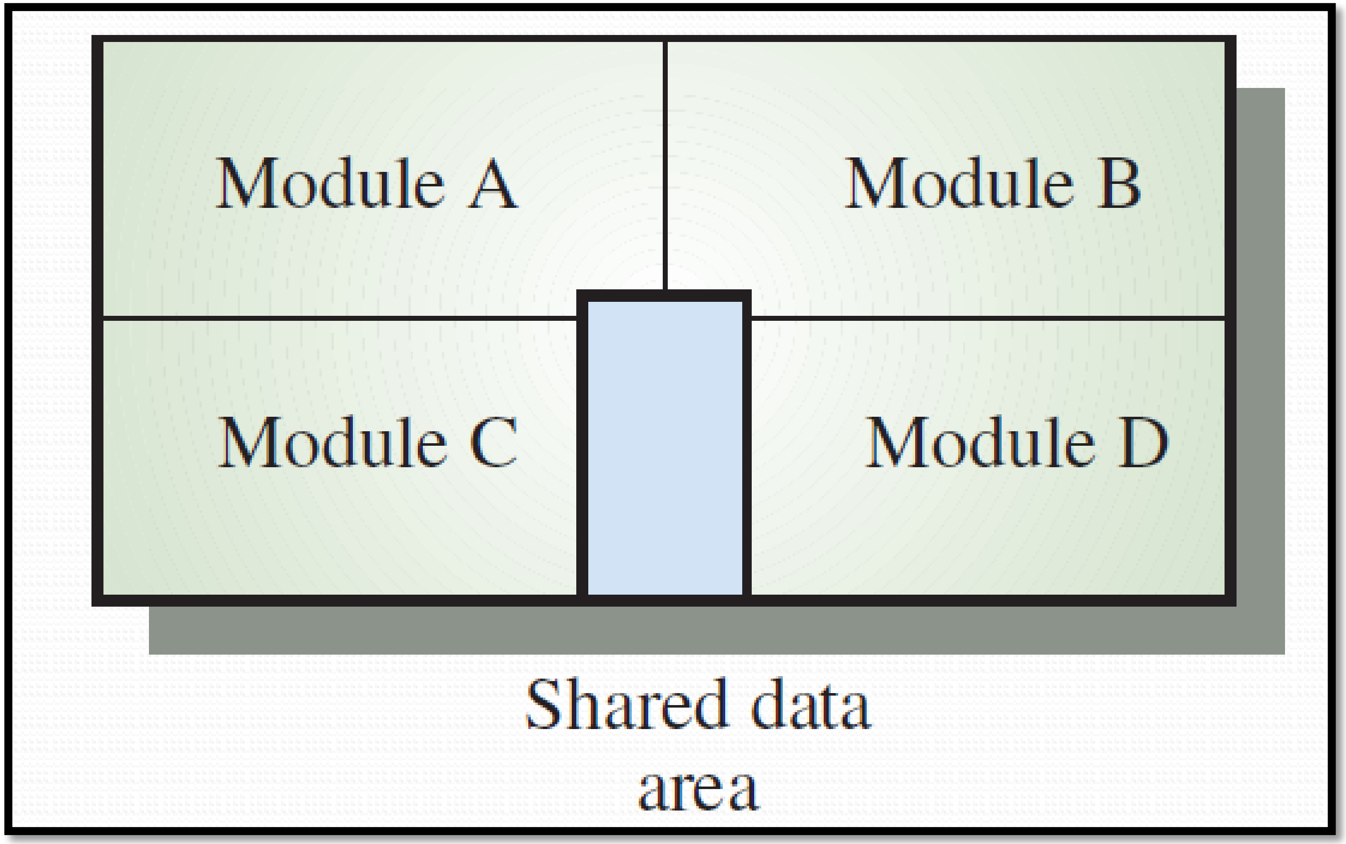
- Functional cohesion (strong)
  - optimal type of cohesion
  - performs a single well-defined action on a single data object
  - e.g. calculate average
  - Each part of a component is necessary for the execution of a single function

- Object cohesion (strong)
  - Each operation provides functionality which allows object attributes to be modified or inspected

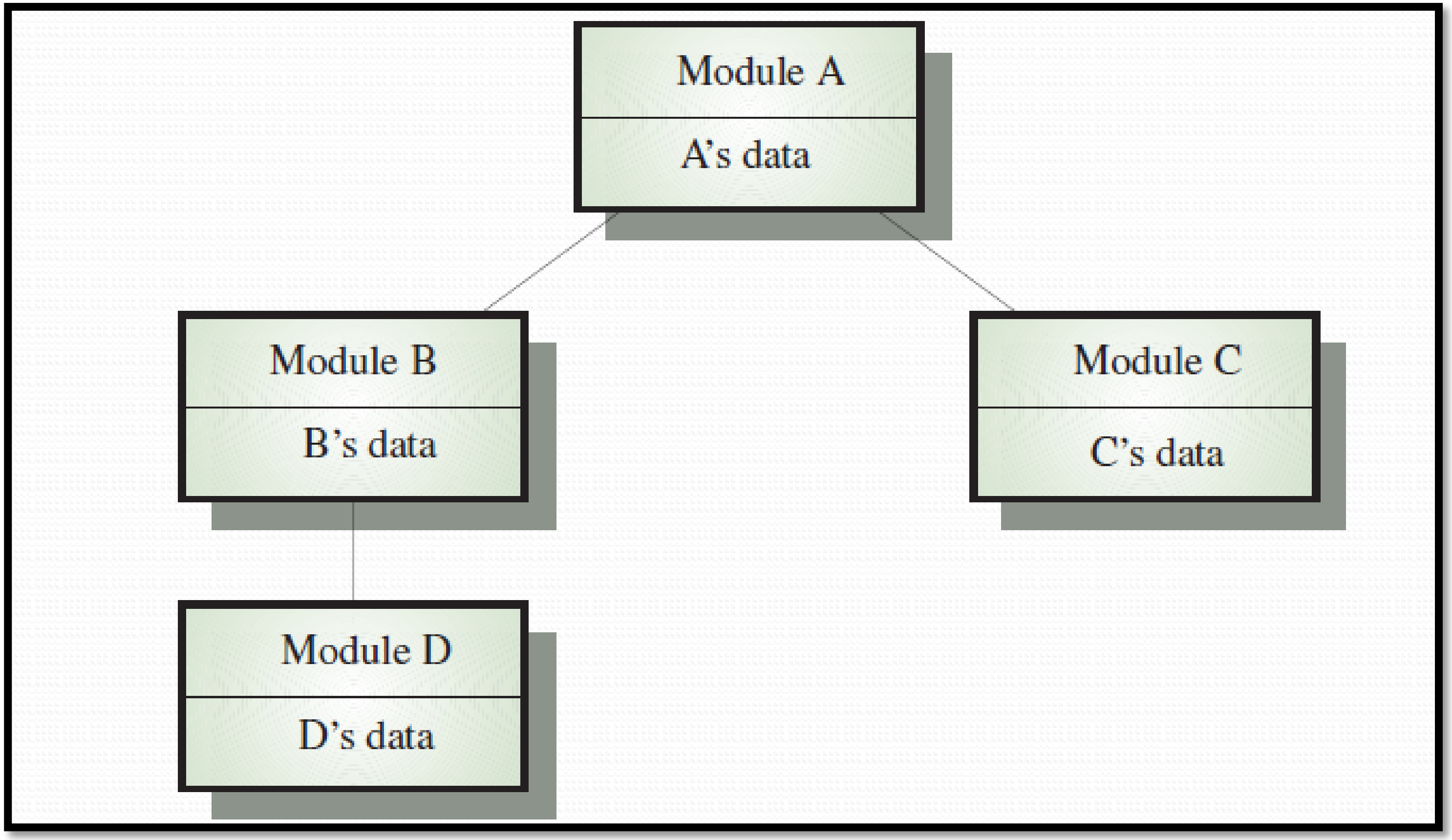
Coupling

- A measure of the strength of the inter-connections between system components
- Loose coupling means component changes are unlikely to affect other components
- Shared variables or control information exchange lead to tight coupling
- Loose coupling can be achieved by state decentralization (as in objects) and component communication via parameters or message passing

Tight Coupling



Loose Coupling



Object Oriented Analysis and Design (OOAD)

Object Oriented Concepts

Key concepts:

- Classes and objects
- Attributes and operations
- Encapsulation and instantiation
- Inheritance

Class

- object-oriented thinking begins with the definition of a class, often defined as:

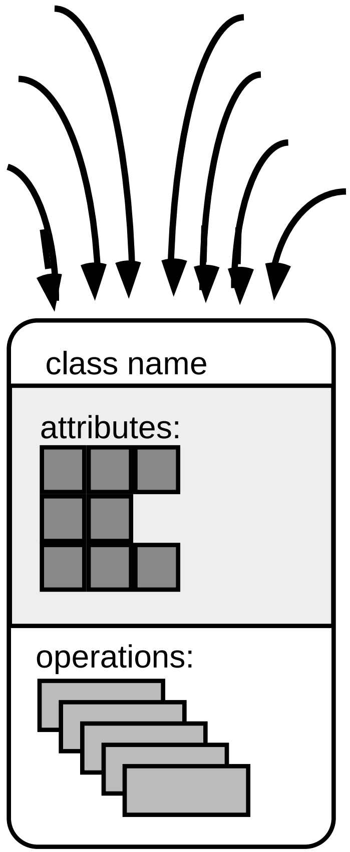
  template, generalized description, “blueprint” ...

  describing a collection of similar items

- a metaclass (also called a superclass) establishes a hierarchy of classes

- once a class of items is defined, a specific instance of the class can be identified

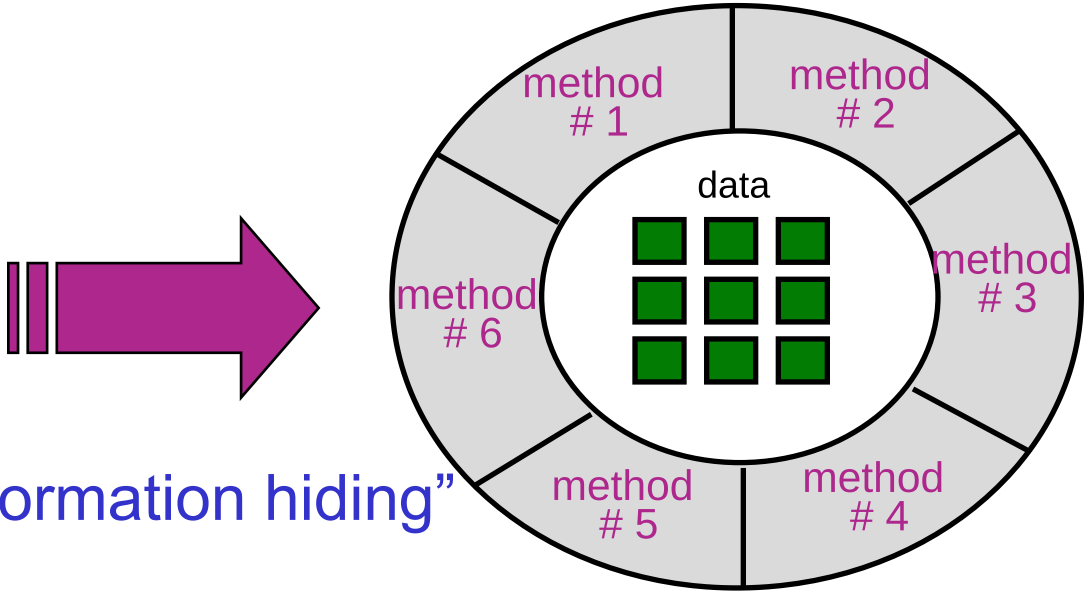
Class



- *external entities* (printer, user, sensor)
- *things* (e.g, reports, displays, signals)
- *occurrences or events* (e.g., interrupt, alarm)
- *roles* (e.g., manager, engineer, salesperson)
- *organizational units* (e.g., division, team)
- *places* (e.g., manufacturing floor)
- *structures* (e.g., employee record)

Encapsulation (Information Hiding)

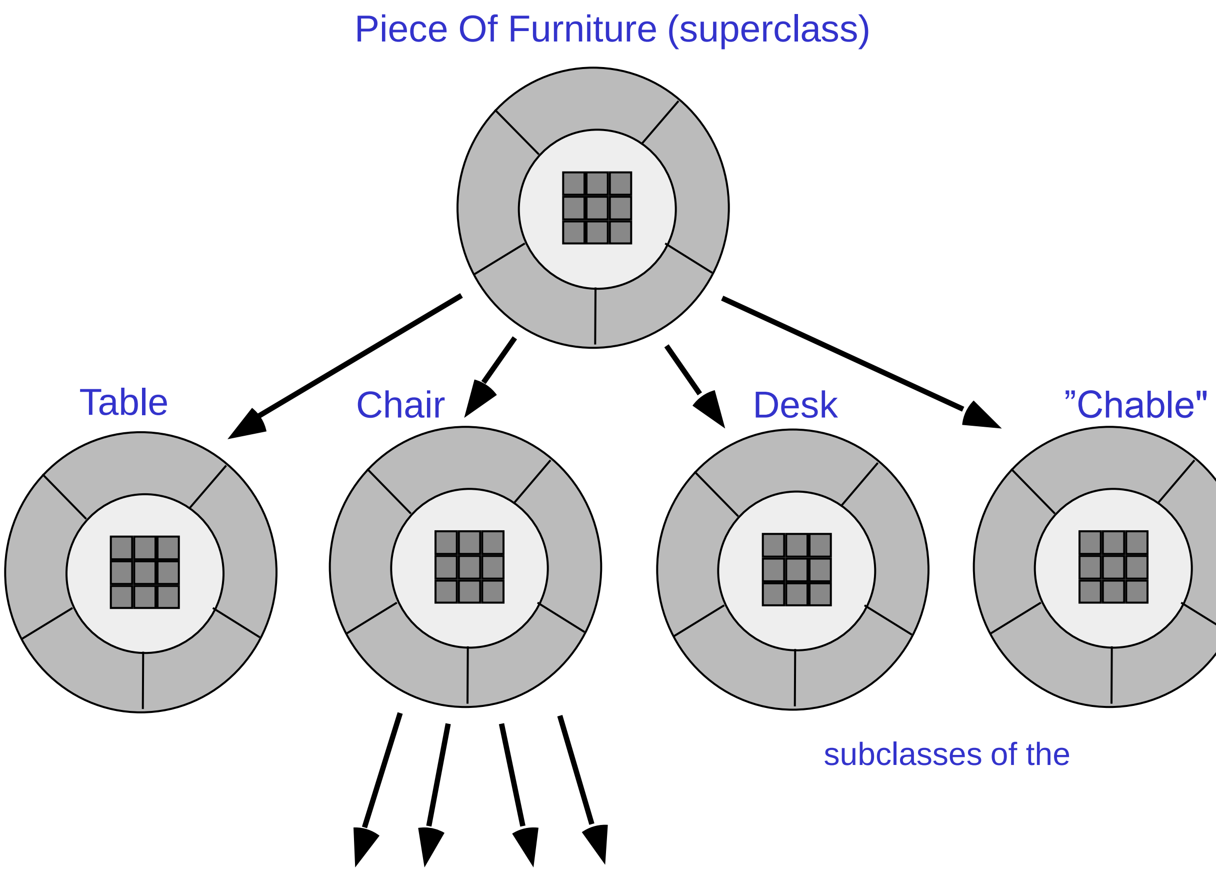
The object encapsulates both data and the logical procedures required to manipulate the data



Achieves "information hiding"

A method is invoked via message passing.
An executable procedure that is encapsulated in a class and is designed to operate on one or more data attributes that are defined as part of the class.

Class Hierarchy



Scenario Based Modeling (Use Case)

"[Use-cases] are simply an aid to defining what exists outside the system (actors) and what should be performed by the system (use-cases)."

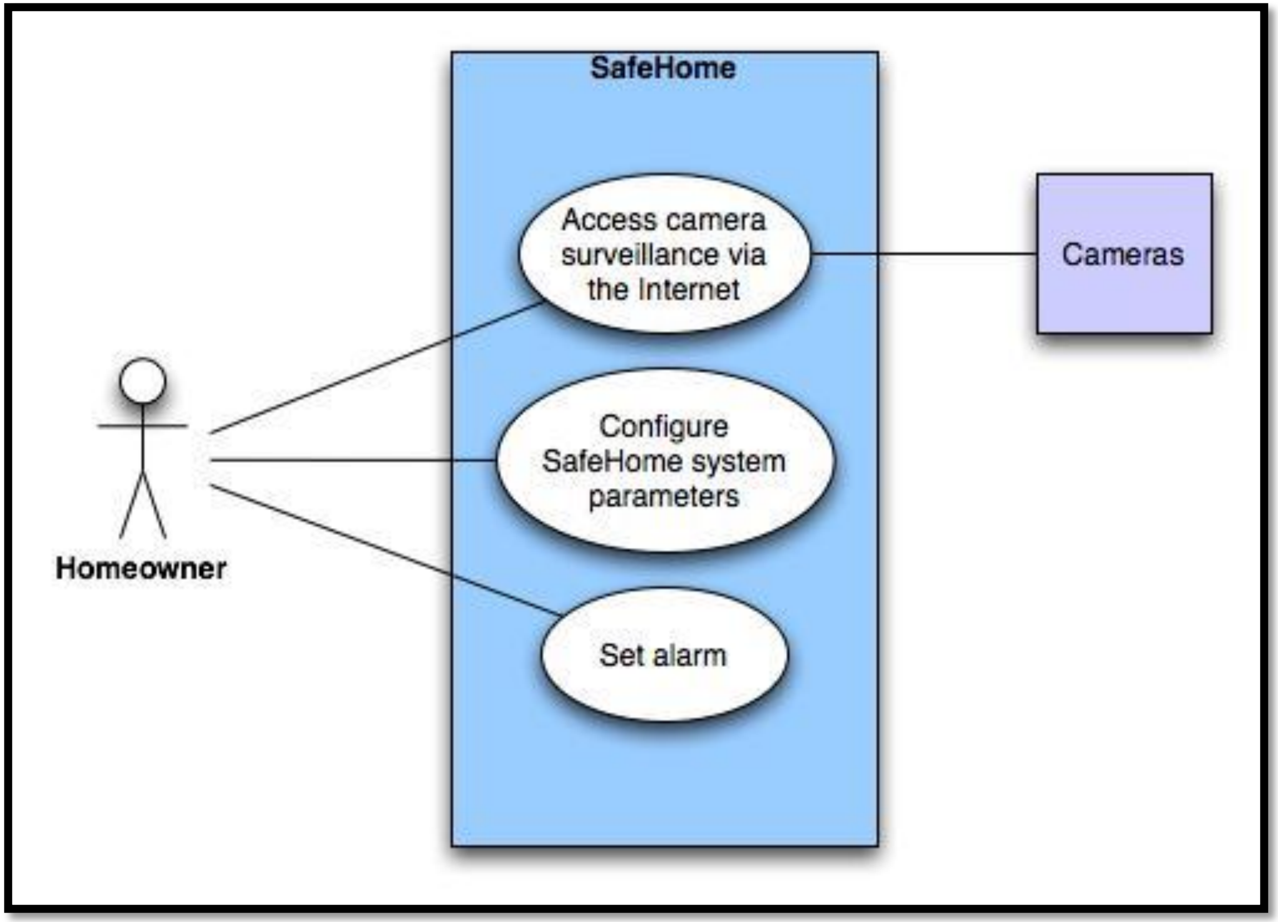
Ivar Jacobson

- a scenario that describes a "thread of usage" for a system
- *actors* represent roles people or devices play as the system functions
- *users* can play a number of different roles for a given scenario

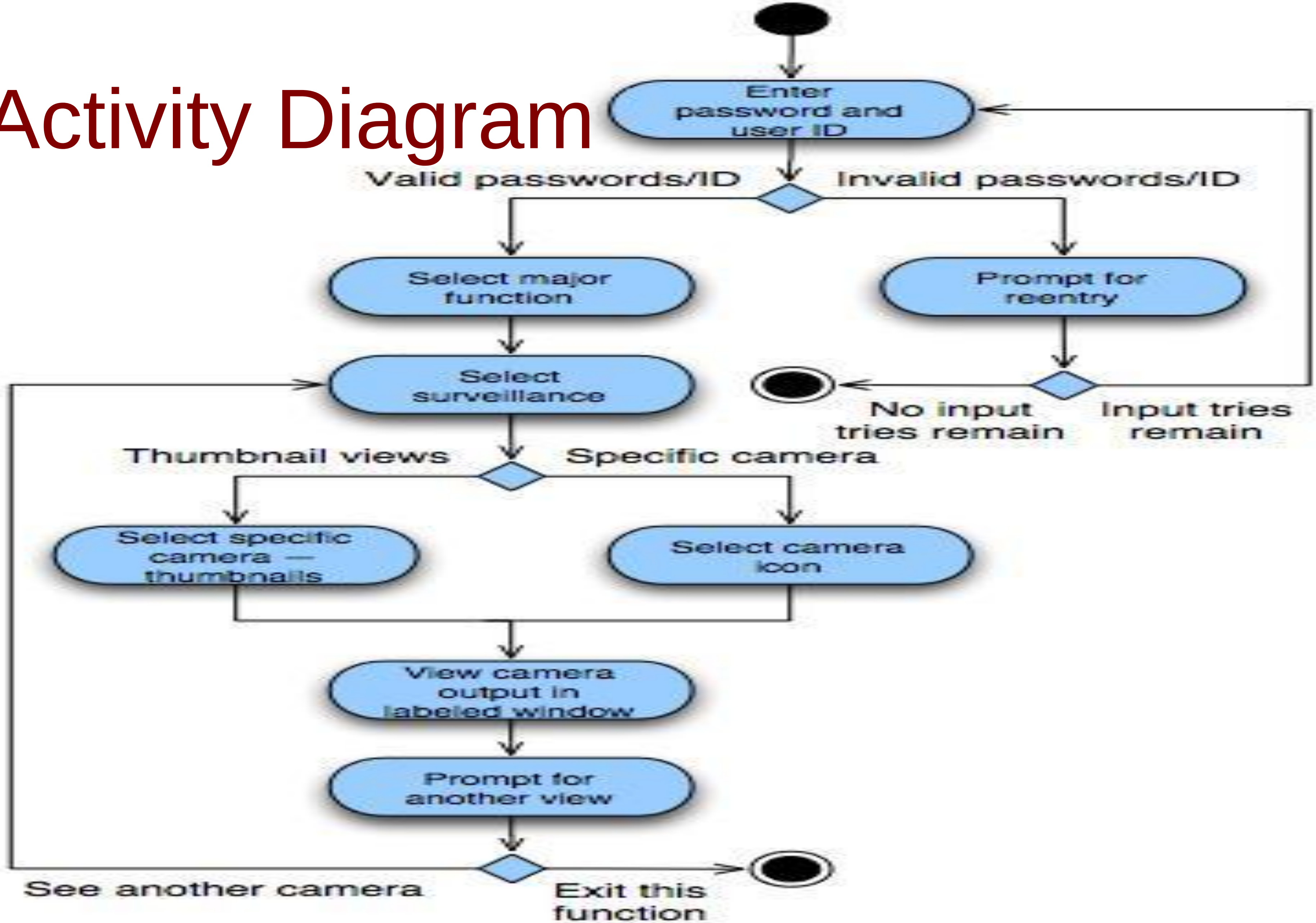
Developing Use Case

- What are the main tasks or functions that are performed by the actor?
- What system information will the 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?

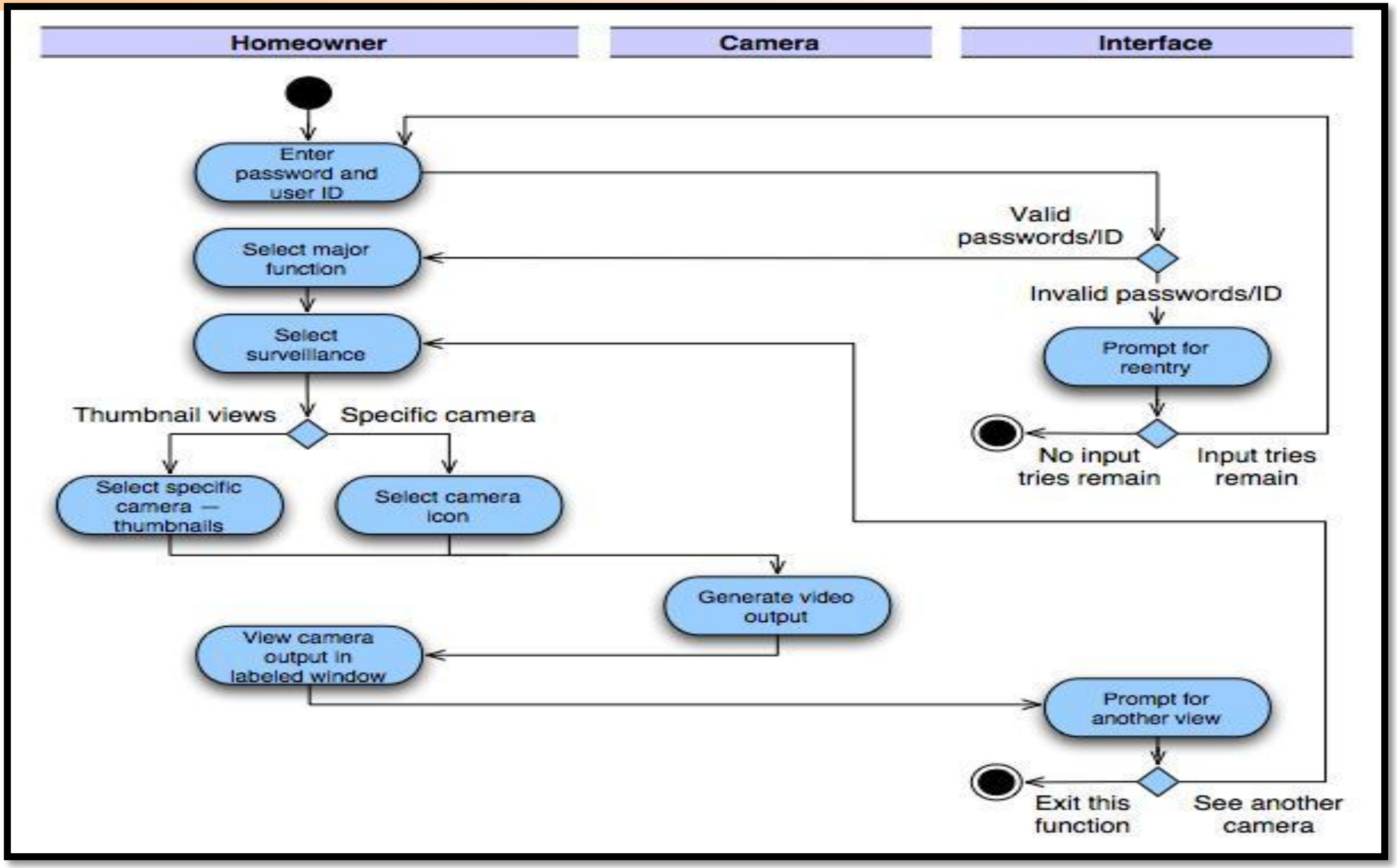
Use Case Diagram



Activity Diagram



Swimlane Diagram



Class Based Modeling

- External entities that produce or consume information
- Things that are part of the information domain
- Occurrences or events
- Roles played by people who interact with the system
- Organizational units
- Places that establish context
- Structures that define a class of objects

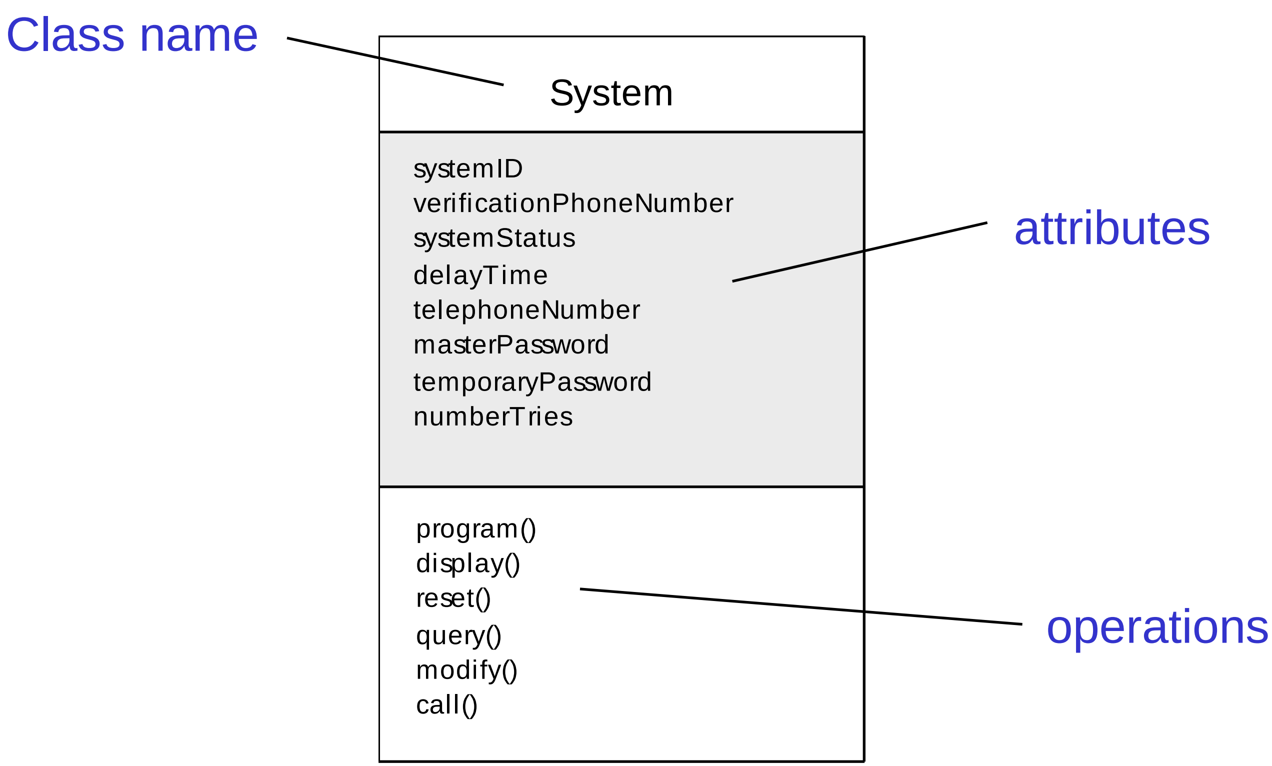
Class Selection Criteria

- Retained information
- Needed services
- Multiple attributes
- Common attributes
- Common operations
- Essential requirements

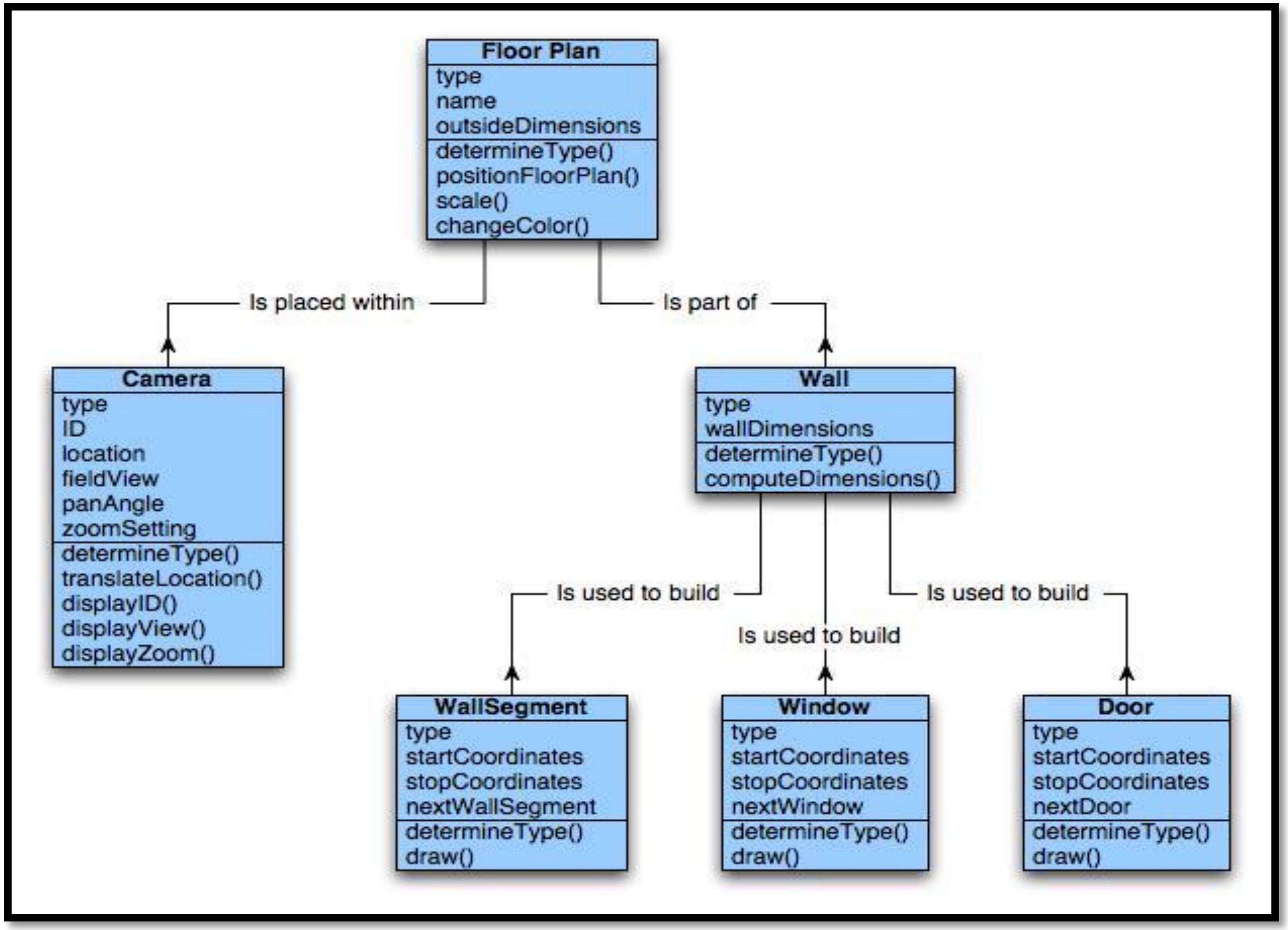
Identifying Class

Potential class	Classification	Accept / Reject
homeowner	role; external entity	reject: 1, 2 fail
sensor	external entity	accept
control panel	external entity	accept
installation	occurrence	reject
(security) system	thing	accept
number, type	not objects, attributes	reject: 3 fails
master password	thing	reject: 3 fails
telephone number	thing	reject: 3 fails
sensor event	occurrence	accept
audible alarm	external entity	accept: 1 fails
monitoring service	organizational unit; ee	reject: 1, 2 fail

Class Diagram



Class Diagram



CRC Modeling

Class: FloorPlan	
Description	
Responsibility	**Collaborator**
Defines floor plan name/type	
Manages floor plan positioning	
Scales floor plan for display	
Incorporates walls, doors, windows	Wall
Shows position of video cameras	Camera

Class Responsibilities

- Distribute system intelligence across classes.
- State each responsibility as generally as possible.
- Put information and the behavior related to it in the same class.
- Localize information about one thing rather than distributing it across multiple classes.
- Share responsibilities among related classes, when appropriate.

Class Types

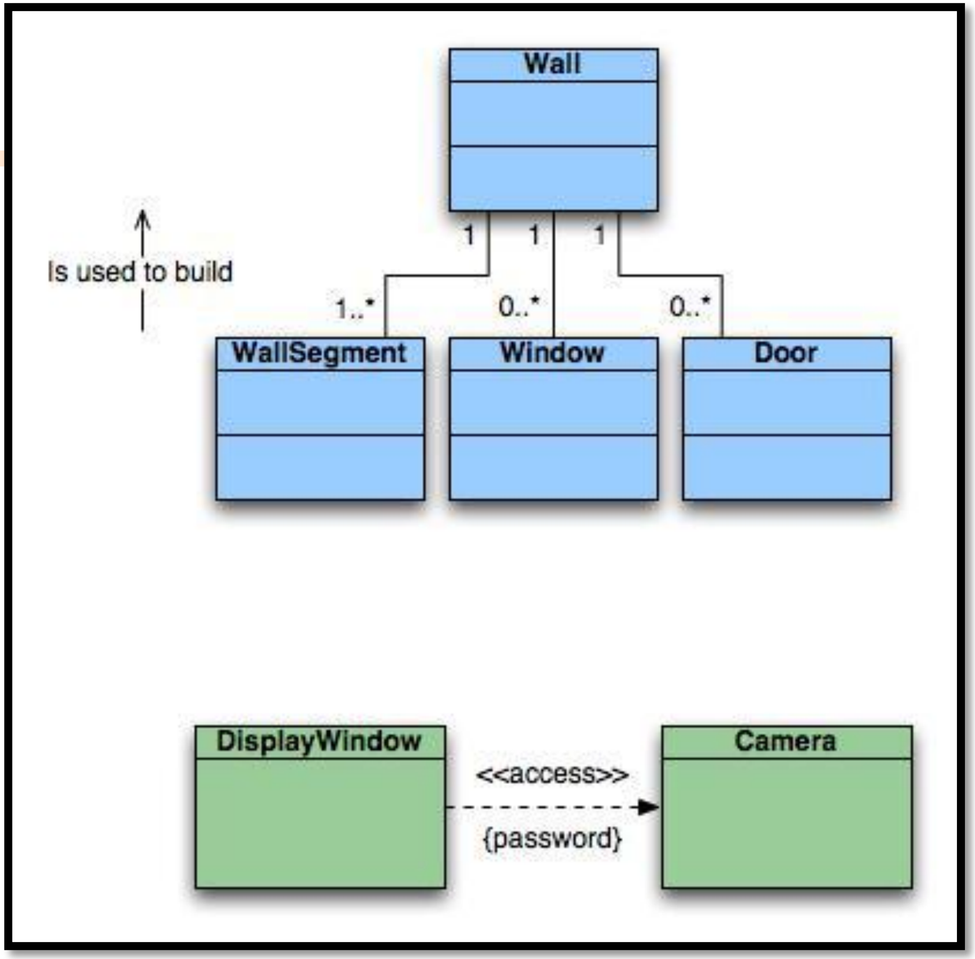
- *Entity classes*, also called *model* or *business* classes, are extracted directly from the statement of the problem (e.g., FloorPlan and Sensor).
- *Boundary classes* are used to create the interface (e.g., interactive screen or printed reports) that the user sees and interacts with as the software is used.
- *Controller classes* manage a “unit of work” [UML03] from start to finish. That is, controller classes can be designed to manage
  - the creation or update of entity objects;
  - the instantiation of boundary objects as they obtain information from entity objects;
  - complex communication between sets of objects;
  - validation of data communicated between objects or between the user and the application.

Class Collaboration

Relationships between classes:

- is-part-of — used when classes are part of an aggregate class.
- has-knowledge-of — used when one class must acquire information from another class.
- depends-on — used in all other cases

Class Diagram



Top: Multiplicity
Bottom: Dependencies

Behavioral Modeling

The behavioral model indicates how software will respond to external events or stimuli. To create the model, the analyst must perform the following steps:

- Evaluate all use-cases to fully understand the sequence of interaction within the system.
- Identify events that drive the interaction sequence and understand how these events relate to specific objects.
- Create a sequence for each use-case.
- Build a state diagram for the system.
- Review the behavioral model to verify accuracy and consistency.

State Representation

- In the context of behavioral modeling, two different characterizations of states must be considered:
  - the state of each class as the system performs its function and the state of the system as observed from the outside as the system performs its function
- The state of a class takes on both passive and active characteristics [CHA93].

  A *passive state* is simply the current status of all of an object's attributes.

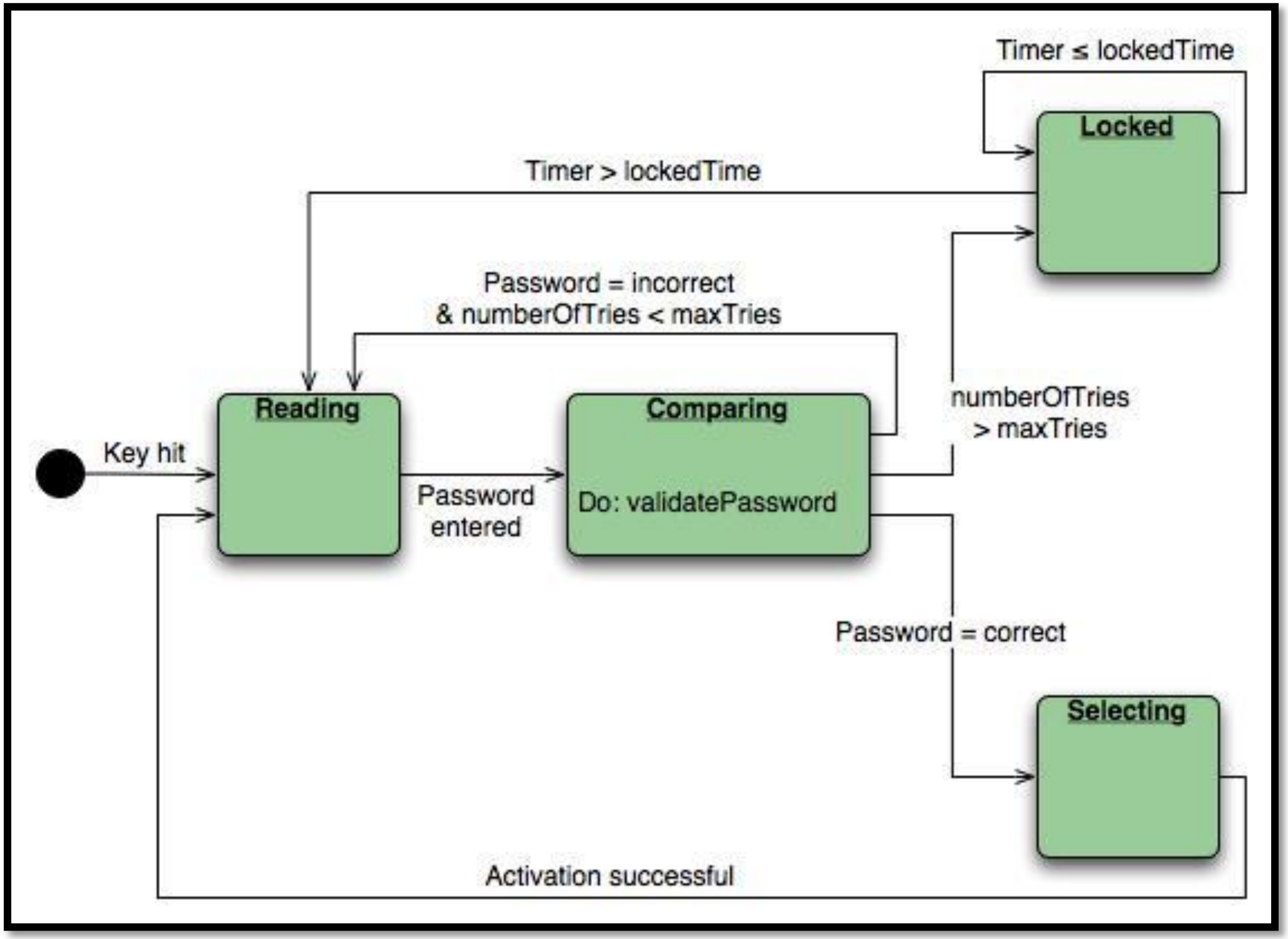
  The *active state* of an object indicates the current status of the object as it undergoes a continuing transformation or processing.

Identifying State

A use-case is examined for *points of information exchange*.

The homeowner uses the keypad to key in a four-digit password. The password is compared with the valid password stored in the system. If the password in incorrect, the control panel will beep once and reset itself for additional input. If the password is correct, the control panel awaits further action.

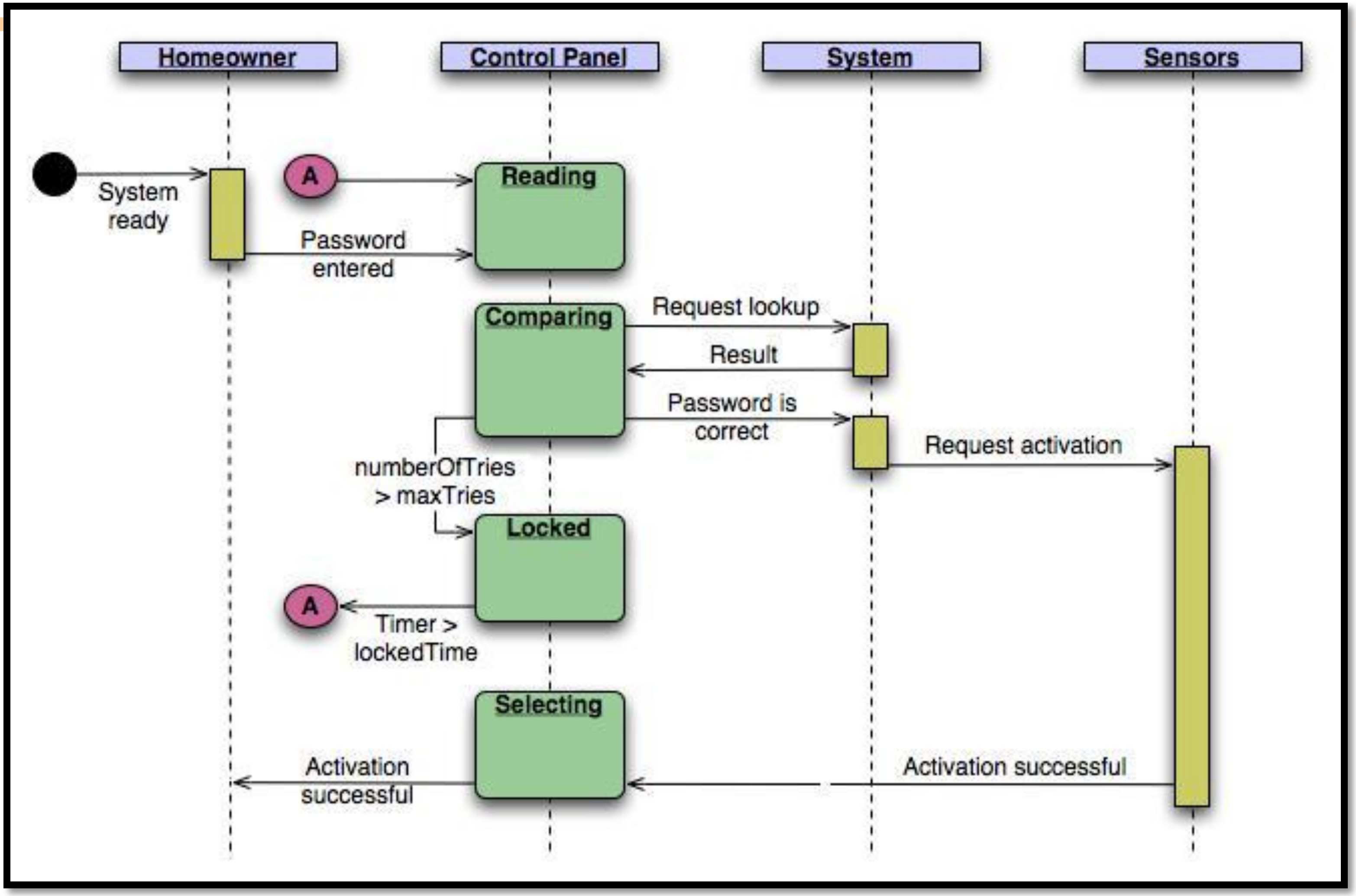
State Diagram



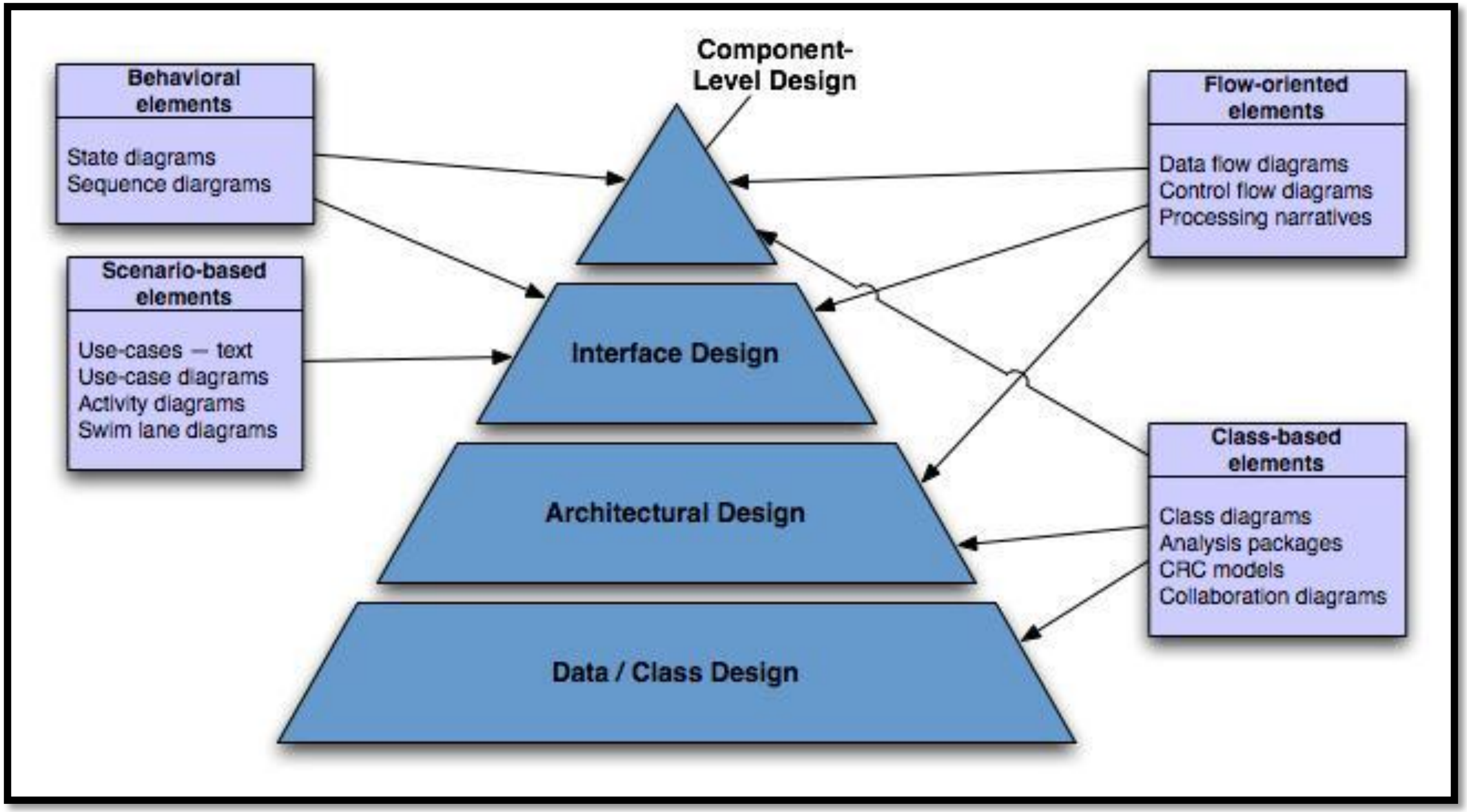
The State of The System

- State — a set of observable circum-stances that characterizes the behavior of a system at a given time
- state transition — the movement from one state to another
- Event — an occurrence that causes the system to exhibit some predictable form of behavior
- Action — process that occurs as a consequence of making a transition

Sequence Diagram



Analysis into Design



Design System Elements

- Data elements
  - Data model --> data structures
  - Data model --> database architecture
- Architectural elements
  - Application domain
  - Analysis classes, their relationships, collaborations and behaviors are transformed into design realizations
  - Patterns and “styles” (Chapter 10)
- Interface elements
  - the user interface (UI)
  - external interfaces to other systems, devices, networks or other producers or consumers of information
  - internal interfaces between various design components.
- Component elements
- Deployment elements

Interface Elements

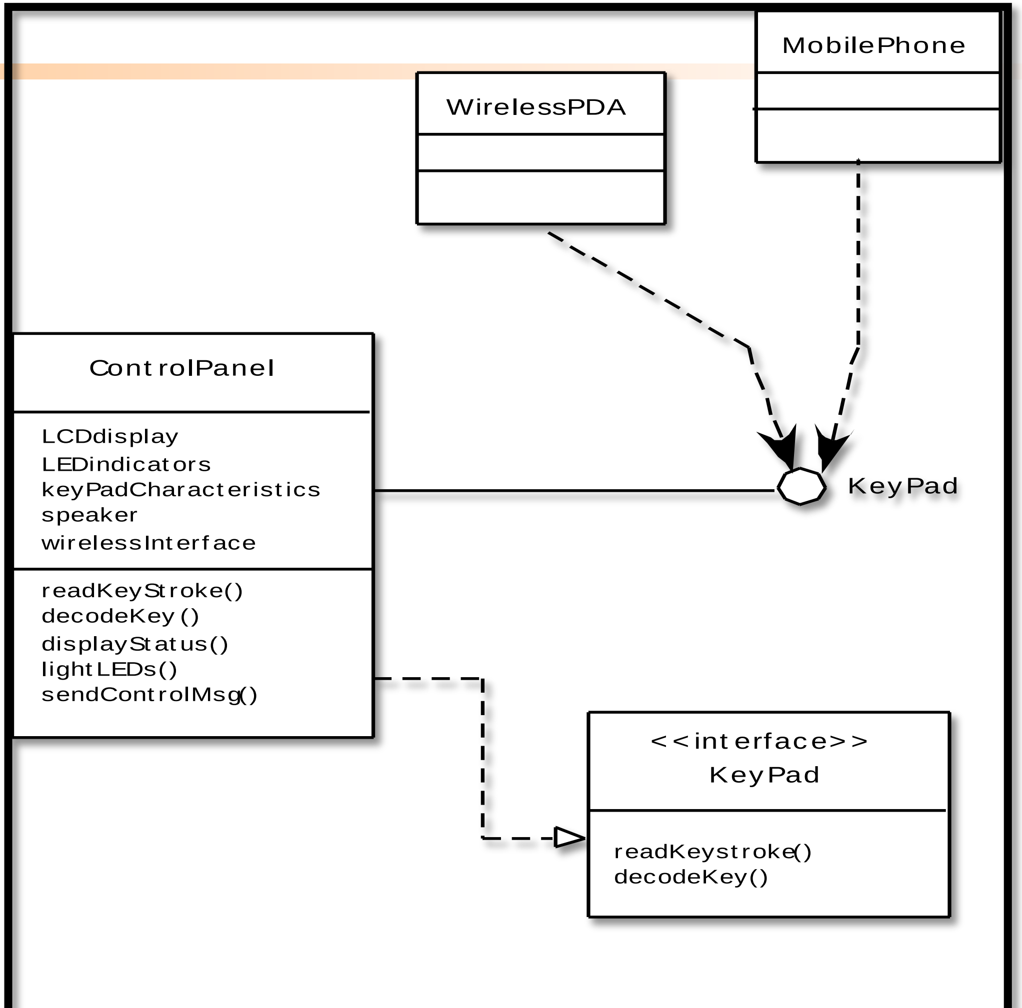
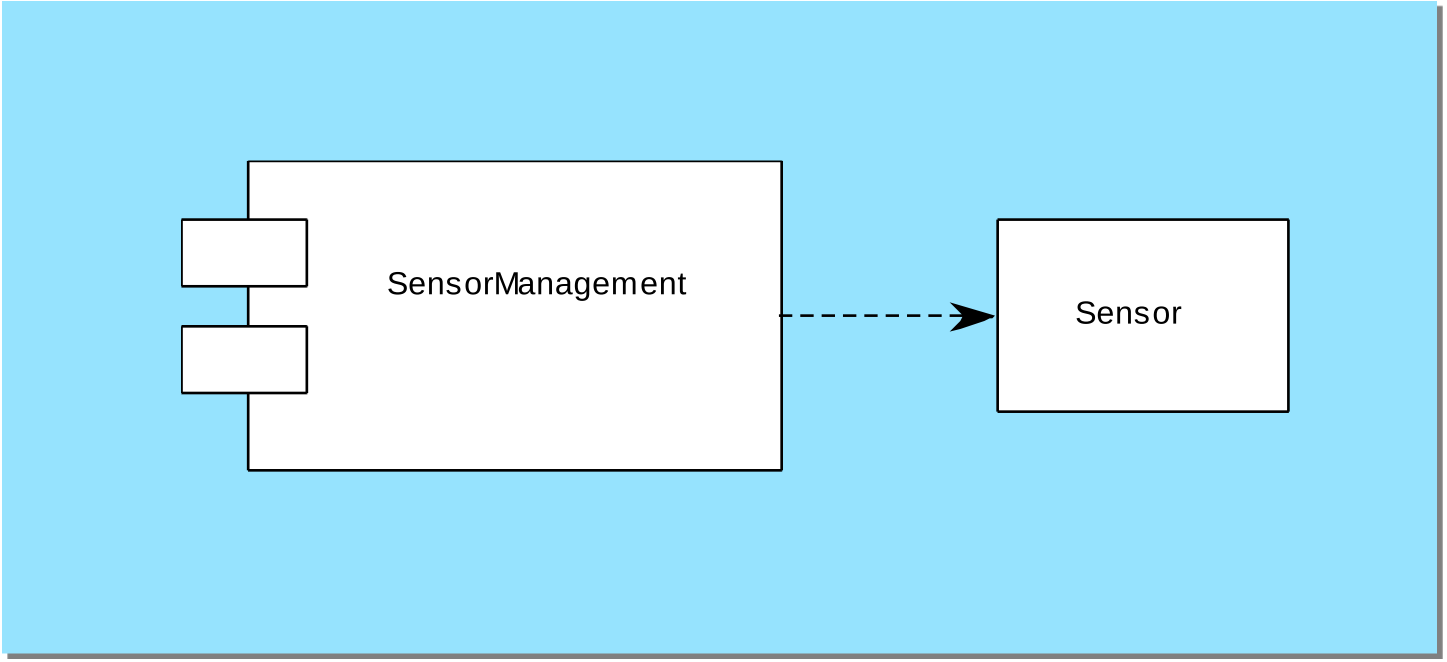
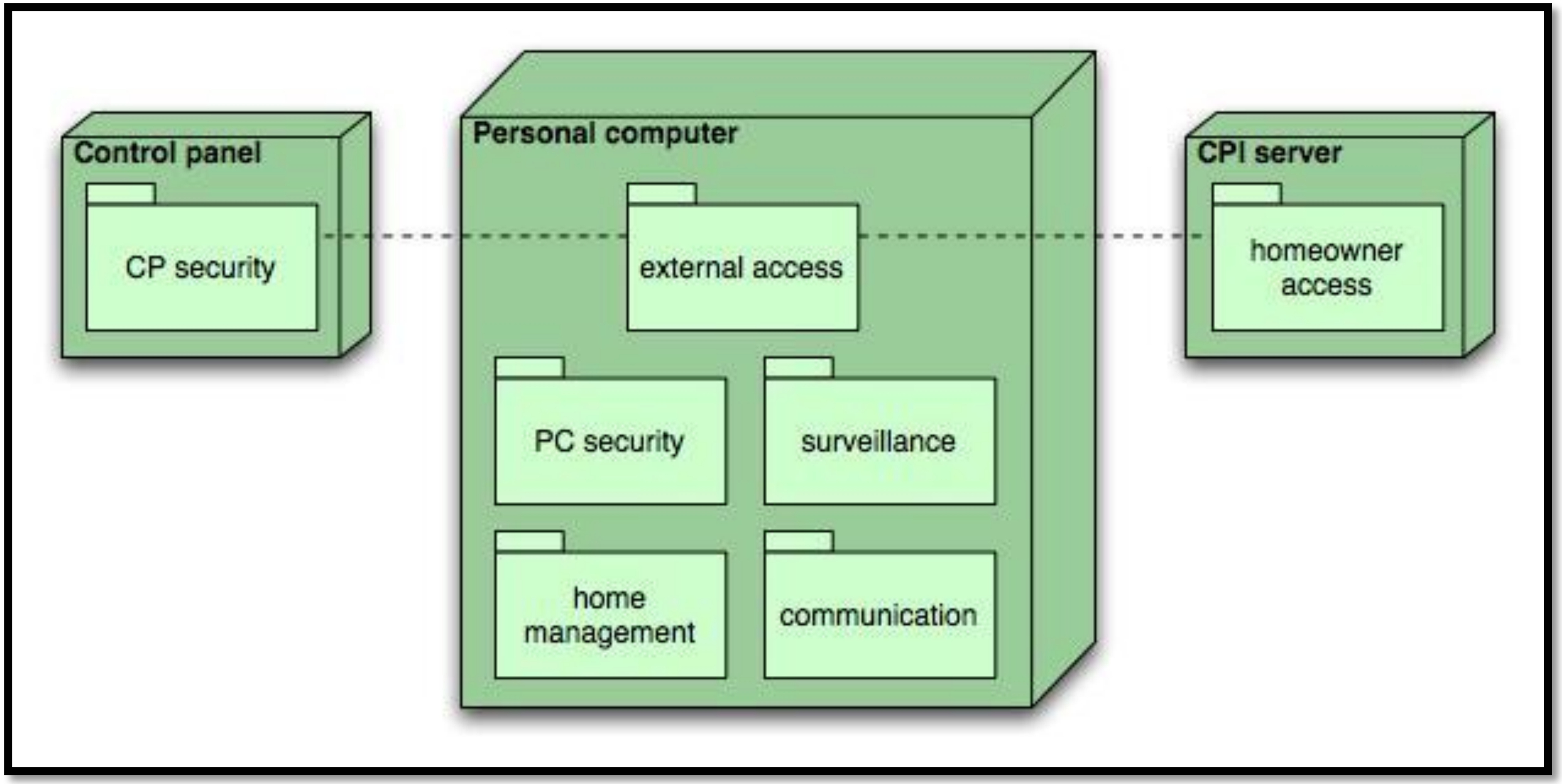


Figure 9.6 UML interface representation for **ControlPanel**

Component Elements



Deployment Diagram



References

1. Roger S. Presmann, Software Engineering, 6th edition.
2. Kendall, System Analysis and Design, 7th edition.
3. Ian Sommerville, Software Engineering, 8th Edition
4. PPT of Roger S. Pressman (chung and zheng)
5. PPT of Kendall
6. Saiful Akbar, Handouts PPL – ITB, 2011