Unit II

Requirement Analysis and Project Estimation:
Requirement Elicitation, Software Requirement Specification (SRS).

Requirement Models: Scenario Based Models, Class Based Models, Behavioural Models and Flow Models**.**

Software Project Estimation: LOC, FP, Empirical Estimation Models COCOMO I COCOMO II, Specialized Estimation Techniques.

Chapter 7 Requirements Engineering
Chapter 8 Building Analysis Model
Chapter 22: 22.2.1, 22.2.2, 23.6.1 to 23.6.4, 23.8, 23.9 and COCOMO pdf

Requirements Engineering

- 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
- Elicitation—elicit requirements from all stakeholders
- Elaboration—create an analysis model that identifies data, function and behavioral requirements
- Negotiation—agree on a deliverable system that is realistic for developers and customers
- Specification—can be any one (or more) of the following:
  - A written document
  - A set of graphical models
  - A formal mathematical model
  - A collection of user scenarios (use-cases)
  - A prototype

Requirements Engineering

- 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.
- Requirements management – set of activities to identify, control, track requirements and changes to requirements at any time
  - Features traceability table
  - Source traceability table
  - Dependency traceability table
  - Subsystem traceability table
  - Interface traceability table

	A01	A02	A03	A04	Aii
R01			✓		
R02		✓			✓
R03			✓	✓	
R04	✓		✓		✓
Rnn		✓		✓	✓

Generic traceability table

Requirements Engineering

- Inception
- Elicitation
- Elaboration
- Negotiation
- Specification
- Validation
- 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 – context free 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?
- Second questions – better understanding of the problem
  - How would you characterize a "good" solution?
  - What problem(s) this solution address
  - What is business environment of this solution?
  - Will a special performance constraint affect the approach of the solution?
- Third questions – effectiveness of the communication
  - Are you the right person? Are your answers official
  - Are my questions relevant
  - Can anyone else provide additional information?
  - Should I be asking you any more questions

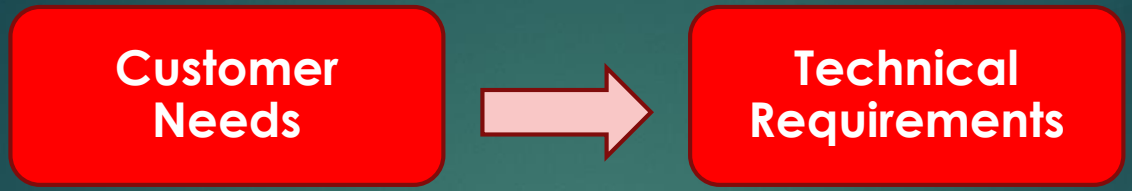
Requirements Engineering

- Inception
- Elicitation
- Elaboration
- Negotiation
- Specification
- Validation
- Requirements management

Eliciting Requirements

- Collaborative Requirements Gathering
  - 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

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
  - Three types of requirements
    - Normal requirements: requirements reflecting objectives and goals of the product
    - Expected requirements: requirements that are implicit to the product
    - Exciting requirements: requirements beyond customers expectations

Eliciting Requirements

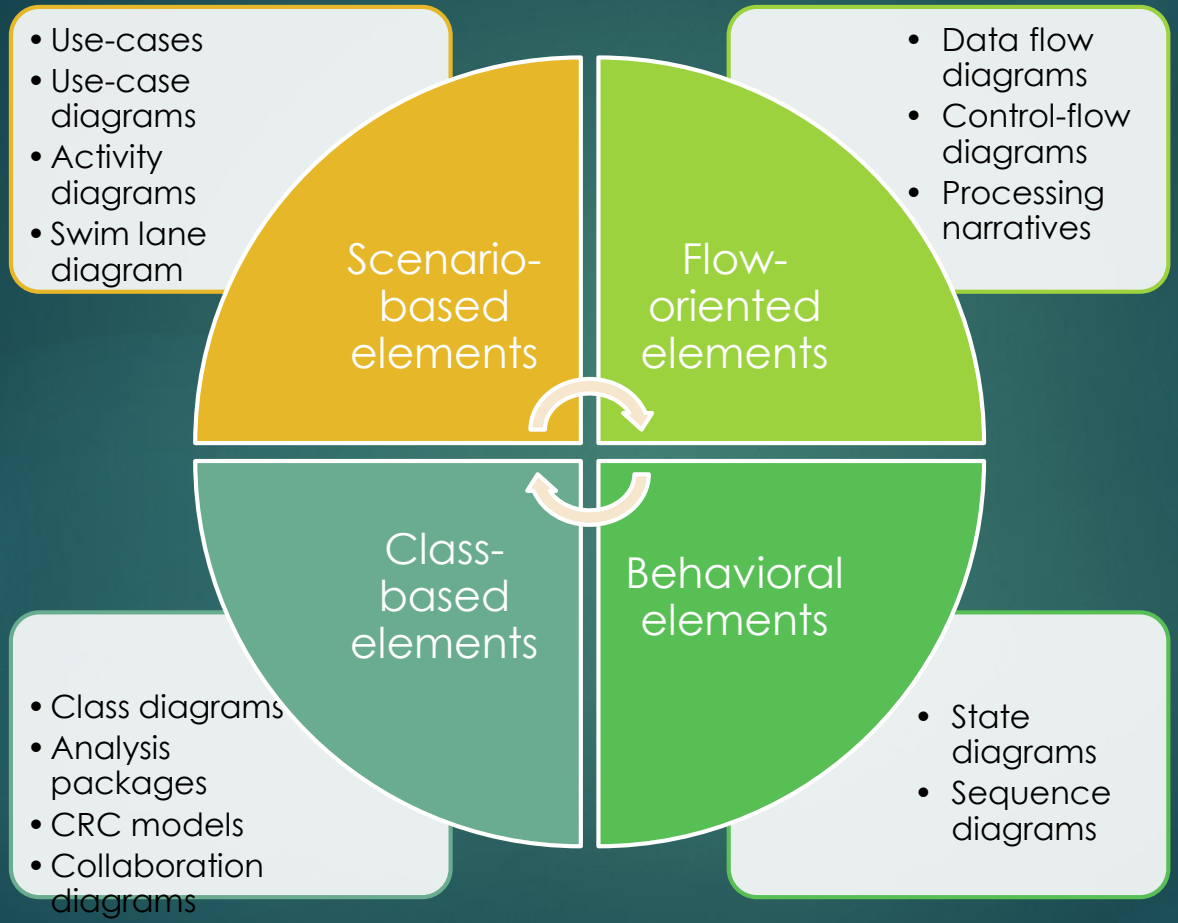
- User Scenarios
  - Creating scenarios that identify a thread of usage for the system (use-cases)
  - Use-cases provide a description of how the system will be used.
- Elicitation Work Products
  - a statement of need and feasibility.
  - a bounded statement of scope for the system or product.
  - a 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.

Requirements Engineering

- Inception
- Elicitation
- Elaboration
- Negotiation
- Specification
- Validation
- Requirements management

Building the Analysis Model

- Analysis model provides a description of required informational, functional and behavioral domains of a system
- 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


• Use-cases
• Use-case diagrams
• Activity diagrams
• Swim lane diagram
Scenario-based elements
Flow-oriented elements
• Data flow diagrams
• Control-flow diagrams
• Processing narratives
Class-based elements
Behavioral elements
• Class diagrams
• Analysis packages
• CRC models
• Collaboration diagrams
• State diagrams
• Sequence diagrams

Requirements Engineering

- Inception
- Elicitation
- Elaboration
- Negotiation
- Specification
- Validation
- Requirements management

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"

Requirements Engineering

- Inception
- Elicitation
- Elaboration
- Negotiation
- Specification
- Validation
- Requirements management

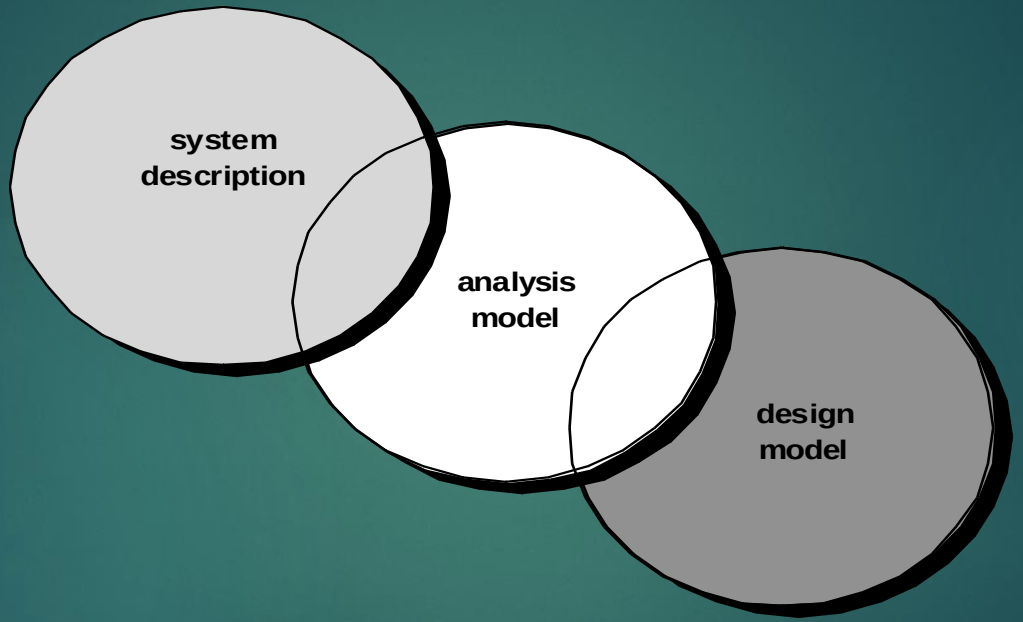
Validating Requirements

- 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?
- 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?

Requirements 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.

A Bridge



Rules of Thumb

- The model should focus on requirements that are visible within the problem or business domain. The level of abstraction should be relatively high.
- Each element of the analysis model should add to an overall understanding of software requirements and provide insight into the information domain, function and behavior of the system.
- Delay consideration of infrastructure and other non-functional models until design.
- Minimize coupling throughout the system.
- Be certain that the analysis model provides value to all stakeholders.
- Keep the model as simple as it can be.

Domain Analysis

Software domain analysis is the identification, analysis, and specification of common requirements from a specific application domain, typically for reuse on multiple projects within that application domain . . . [Object-oriented domain analysis is] the identification, analysis, and specification of common, reusable capabilities within a specific application domain, in terms of common objects, classes, subassemblies, and frameworks . . .

Donald Firesmith

Domain Analysis

- Define the domain to be investigated. E.g banking, avionics, embedded software etc
- Collect a representative sample of applications in the domain.
- Analyze each application in the sample.
- Develop an analysis model for the objects.
- E.g. find or create analysis classes, common features / functions that can be reused if applicable

Analysis Modeling

- Structured analysis – considers data and the process that transform the data.
- Object oriented analysis – definition of classes and their collaborations with one another

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

What is a Data Object?

Object —something that is described by a set of attributes (data items) and that will be manipulated within the software (system)

- each instance of an object (e.g., a book) can be identified uniquely (e.g., ISBN #)
- each plays a necessary role in the system
  i.e., the system could not function without
  access to instances of the object
- each is described by attributes that are
  themselves data items

Typical Objects

- *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, warehouse)
- *structures* (e.g., employee record, a file)

Data Objects and Attributes

A data object contains a set of attributes that act as an aspect, quality, characteristic, or descriptor of the object

object: automobile

attributes:
- **make**
- **model**
- **body type**
- **price**
- **color**

What is a Relationship?

relationship —indicates “connectedness”;

a "fact" that must be "remembered"
by the system and cannot or is not computed
or derived mechanically

- several instances of a relationship can exist
- objects can be related in many different ways

Cardinality and modality
Cardinality – no. of objects that can participate in a relationship

Modality – if the relationship is mandatory then modality is 1, else 0

ERD Notation

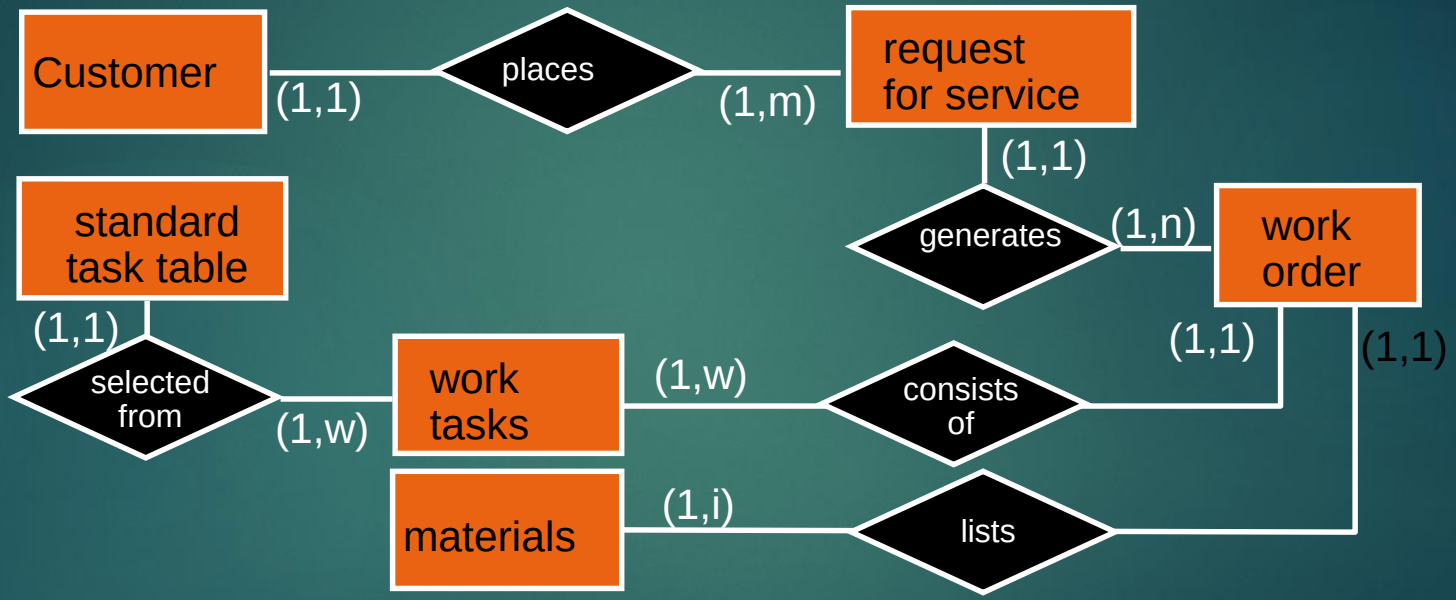
One common form:

Another common form:

Building an ERD

- Level 1—model all data objects (entities) and their "connections" to one another
- Level 2—model all entities and relationships
- Level 3—model all entities, relationships, and the attributes that provide further depth

The ERD: An Example



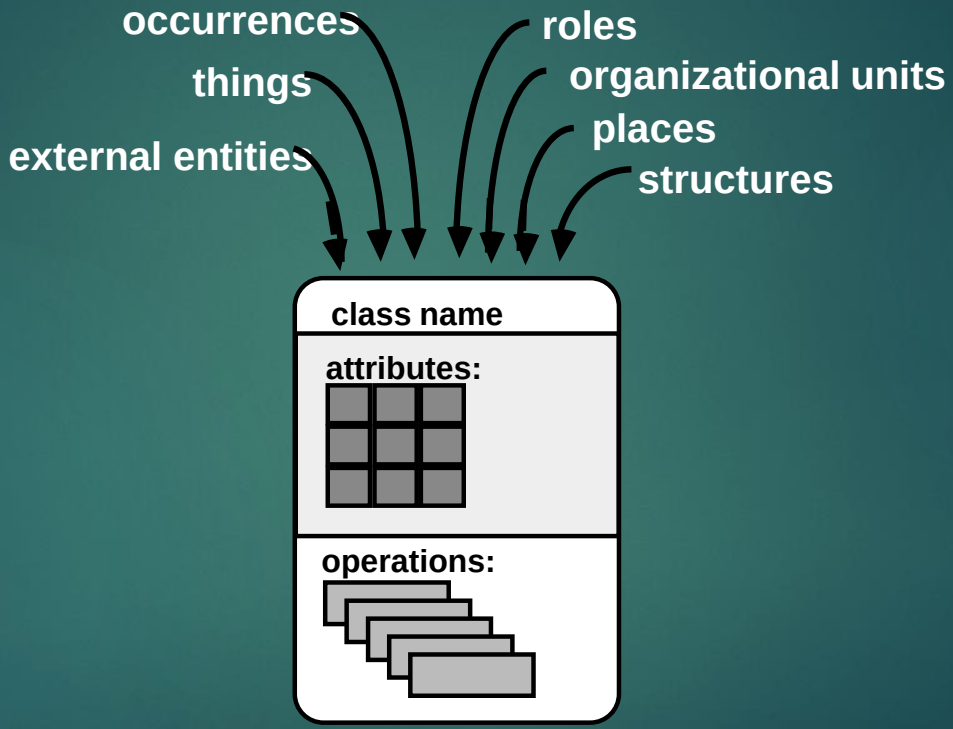
Object-Oriented Concepts

- Must be understood to apply class-based elements of the analysis model
- Key concepts:
  - Classes and objects
  - Attributes and operations
  - Encapsulation and instantiation
  - Inheritance

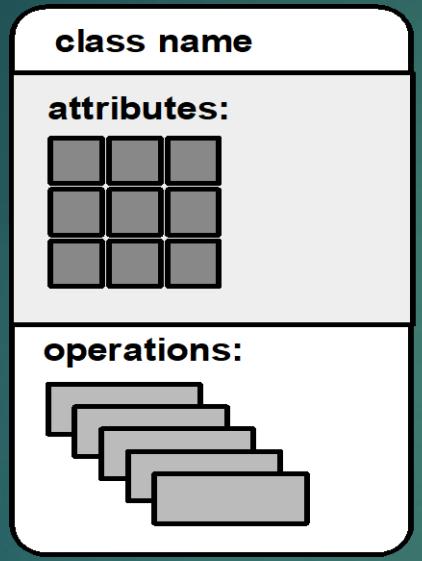
Classes

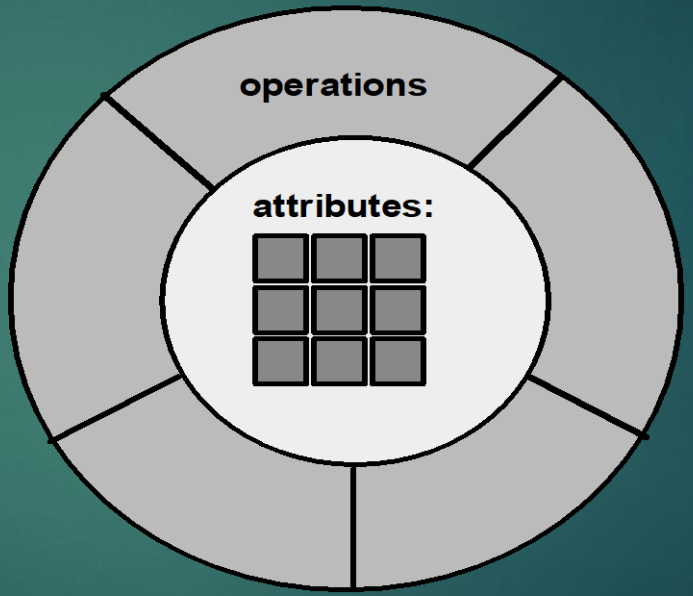
- object-oriented thinking begins with the definition of a class, often defined as:
  - template
  - generalized description
  - "blueprint" ... describing a collection of similar items
- a metaclass (also called a superclass) establishes a hierarchy of classes
- once a class of items is defined, a specific instance of the class can be identified

What is a Class?



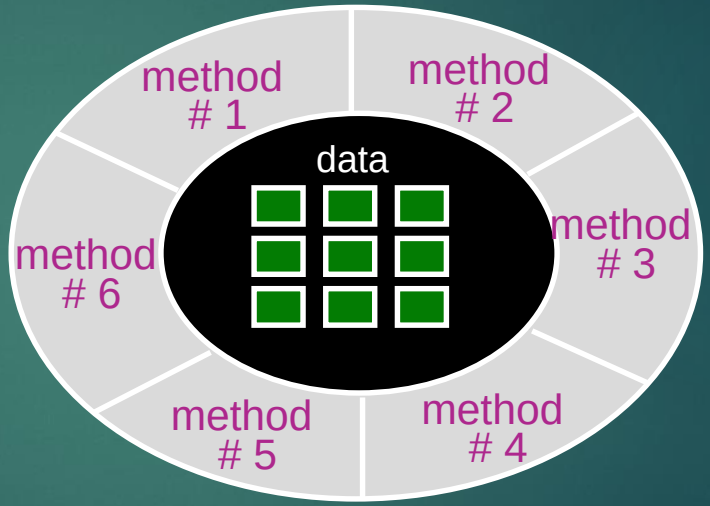
Building a Class





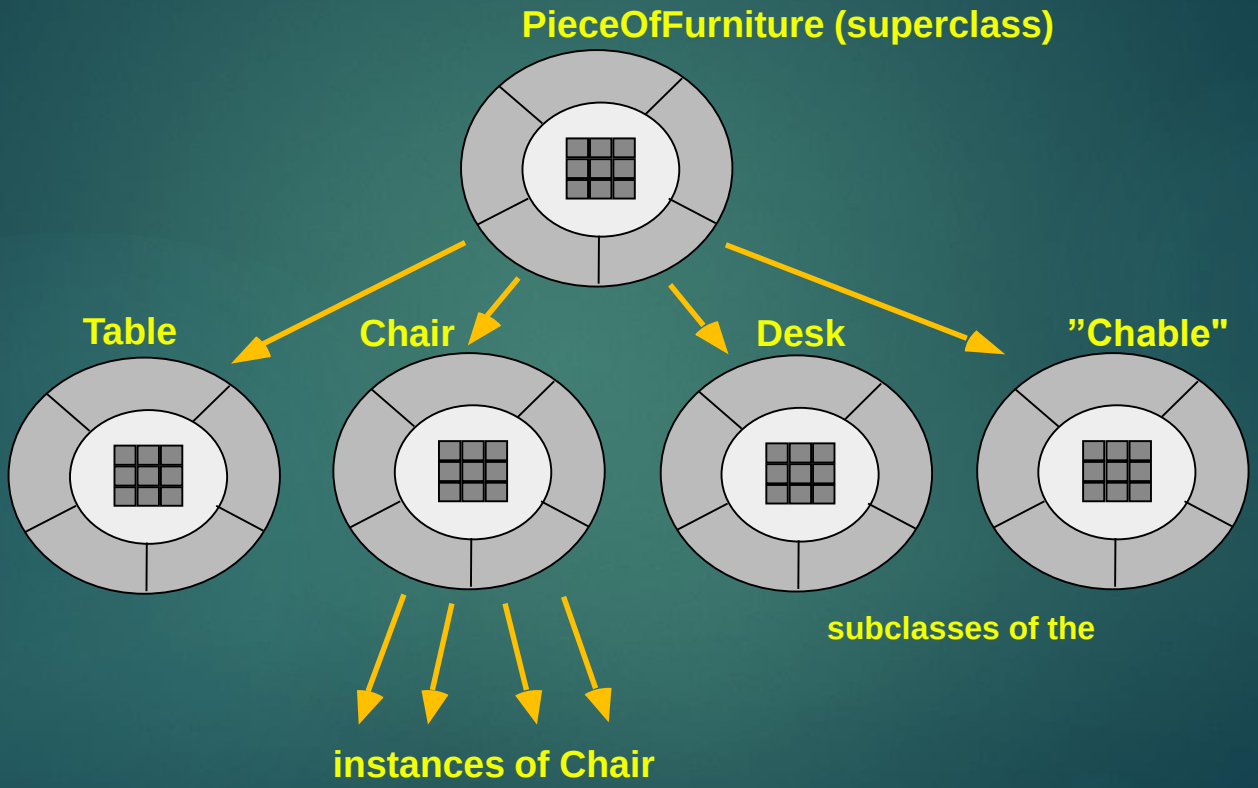
Encapsulation/Hiding

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



Achieves “information hiding”

Class Hierarchy

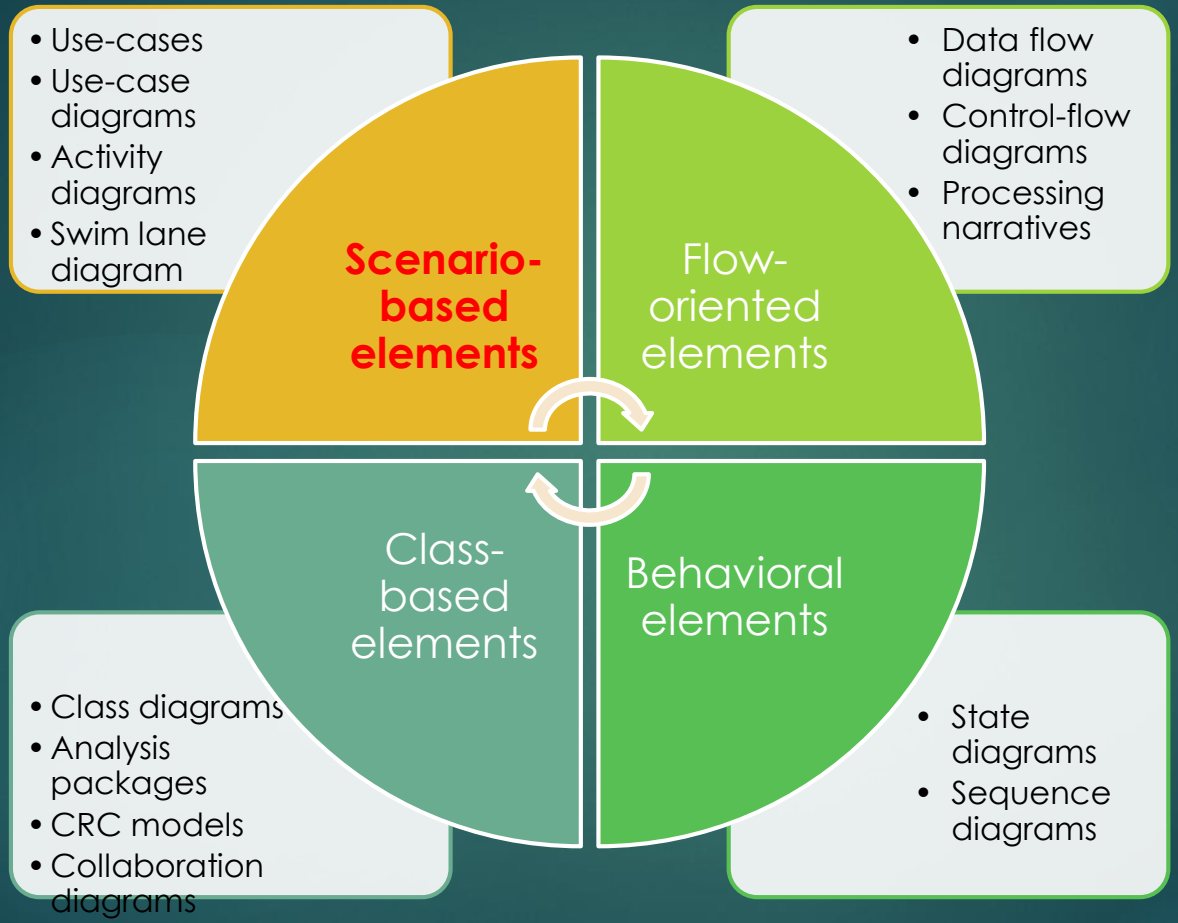


Methods (a.k.a. Operations, Services)

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.

A method is invoked via message passing.

Building an Analysis Model


• Use-cases
• Use-case diagrams
• Activity diagrams
• Swim lane diagram
Scenario-based elements
Flow-oriented elements
• Data flow diagrams
• Control-flow diagrams
• Processing narratives
Class-based elements
Behavioral elements
• Class diagrams
• Analysis packages
• CRC models
• Collaboration diagrams
• State diagrams
• Sequence diagrams

Scenario-Based Modeling

"[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

(1) What should we write about?

(2) How much should we write about it?

(3) How detailed should we make our description?

(4) How should we organize the description?

Use-Cases

- 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

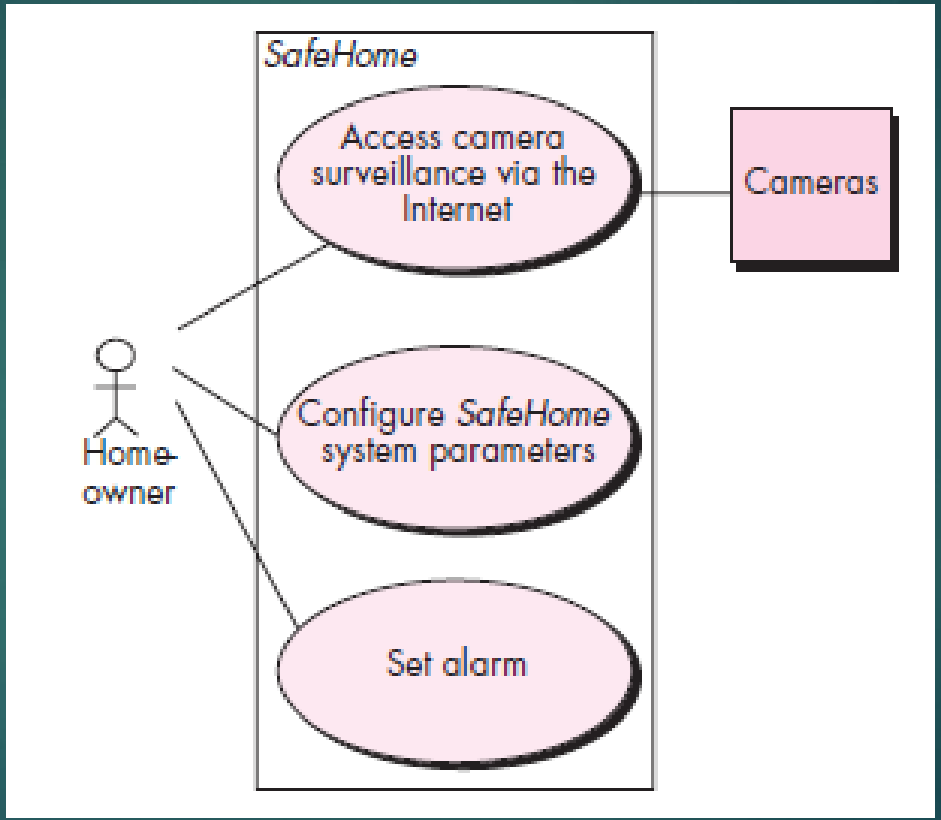
Use-Cases

- 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?

Developing a Use-Case

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



Use Case Template for Surveillance

Use case: Access camera surveillance via the Internet—display camera views (ACS-DCV)

Iteration: 2, last modification: January 14 by V. Raman.

Primary actor: Homeowner.

Goal in context: To view output of camera placed throughout the house from any remote location via the Internet.

Preconditions: System must be fully configured; appropriate user ID and passwords must be obtained.

Trigger: The homeowner decides to take a look inside the house while away.

Scenario:

1. The homeowner logs onto the *SafeHome Products* website.
2. The homeowner enters his or her user ID.
3. The homeowner enters two passwords (each at least eight characters in length).
4. The system displays all major function buttons.
5. The homeowner selects the "surveillance" from the major function buttons.
6. The homeowner selects "pick a camera."
7. The system displays the floor plan of the house.
8. The homeowner selects a camera icon from the floor plan.
9. The homeowner selects the "view" button.
10. The system displays a viewing window that is identified by the camera ID.
11. The system displays video output within the viewing window at one frame per second.

Exceptions:

1. ID or passwords are incorrect or not recognized—see use case **Validate ID and passwords.**
2. Surveillance function not configured for this system—system displays appropriate error message; see use case **Configure surveillance function.**
3. Homeowner selects "View thumbnail snapshots for all camera"—see use case **View thumbnail snapshots for all cameras.**
4. A floor plan is not available or has not been configured—display appropriate error message and see use case **Configure floor plan.**
5. An alarm condition is encountered—see use case **Alarm condition encountered.**

Priority: Moderate priority, to be implemented after basic functions.

When available: Third increment.

Frequency of use: Moderate frequency.

Channel to actor: Via PC-based browser and Internet connection.

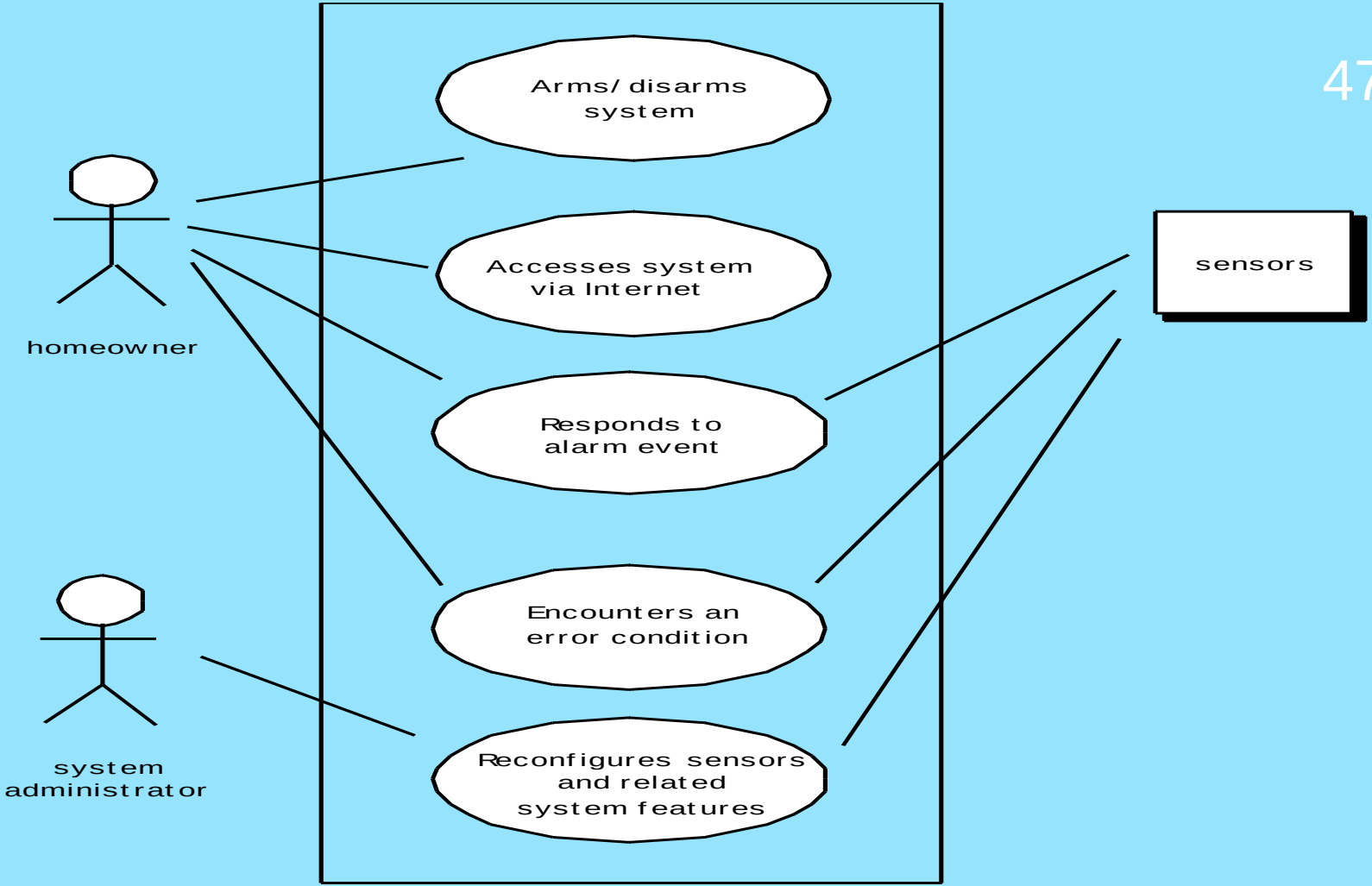
Secondary actors: System administrator, cameras.

Channels to secondary actors:

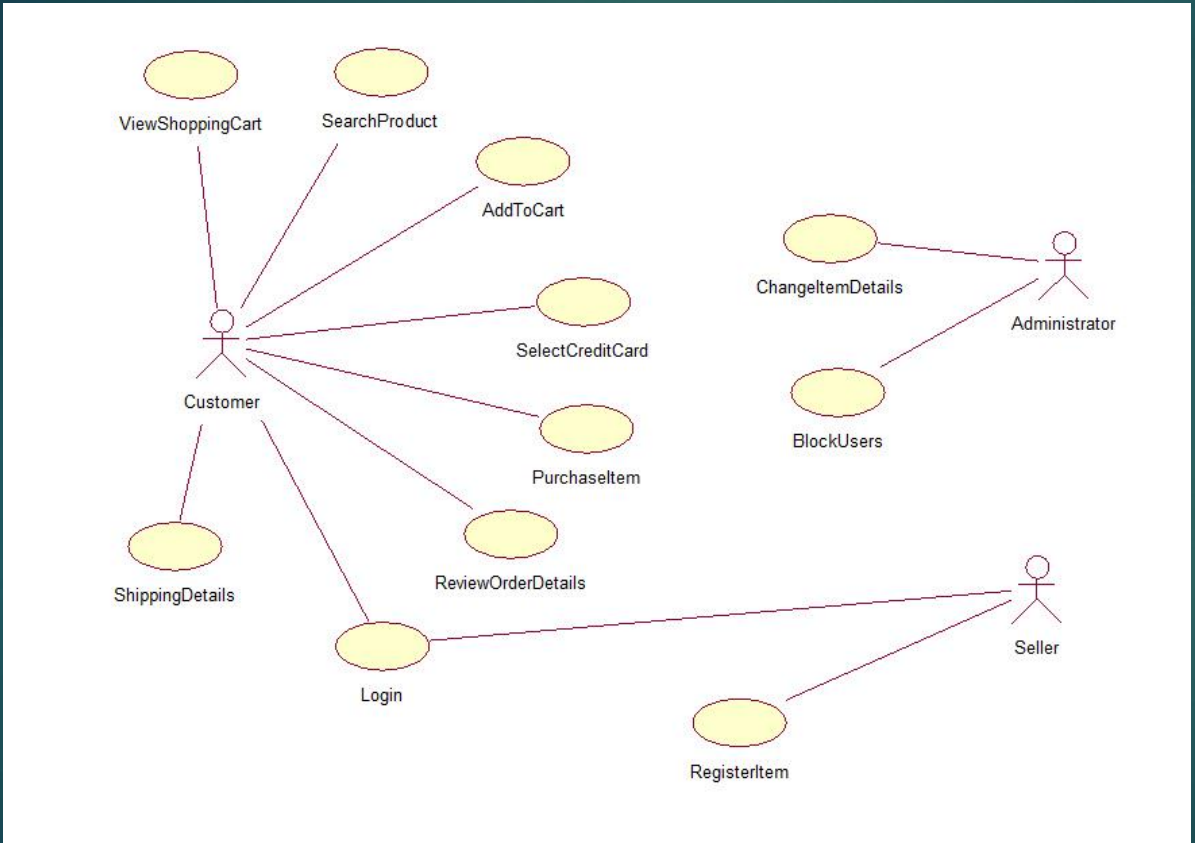
1. System administrator: PC-based system.
2. Cameras: wireless connectivity.

Open issues:

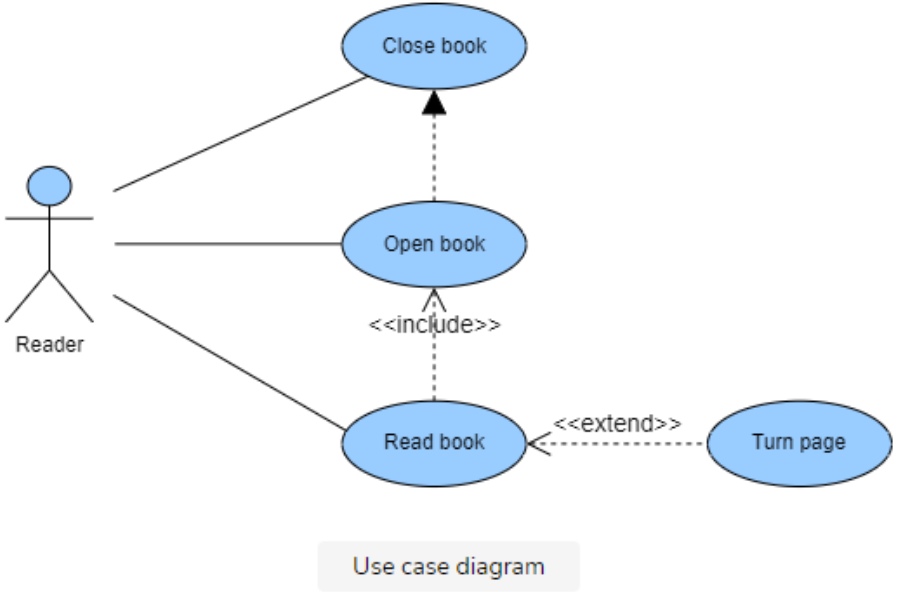
1. What mechanisms protect unauthorized use of this capability by employees of *SafeHome Products*?
2. Is security sufficient? Hacking into this feature would represent a major invasion of privacy.
3. Will system response via the Internet be acceptable given the bandwidth required for camera views?
4. Will we develop a capability to provide video at a higher frames-per-second rate when high-bandwidth connections are available?


Arms/ disarms system
Accesses system via Internet
Responds to alarm event
Encounters an error condition
Reconfigures sensors and related system features
homeowner
system administrator
sensors

Use-case Online book shop



Use-case diagrams



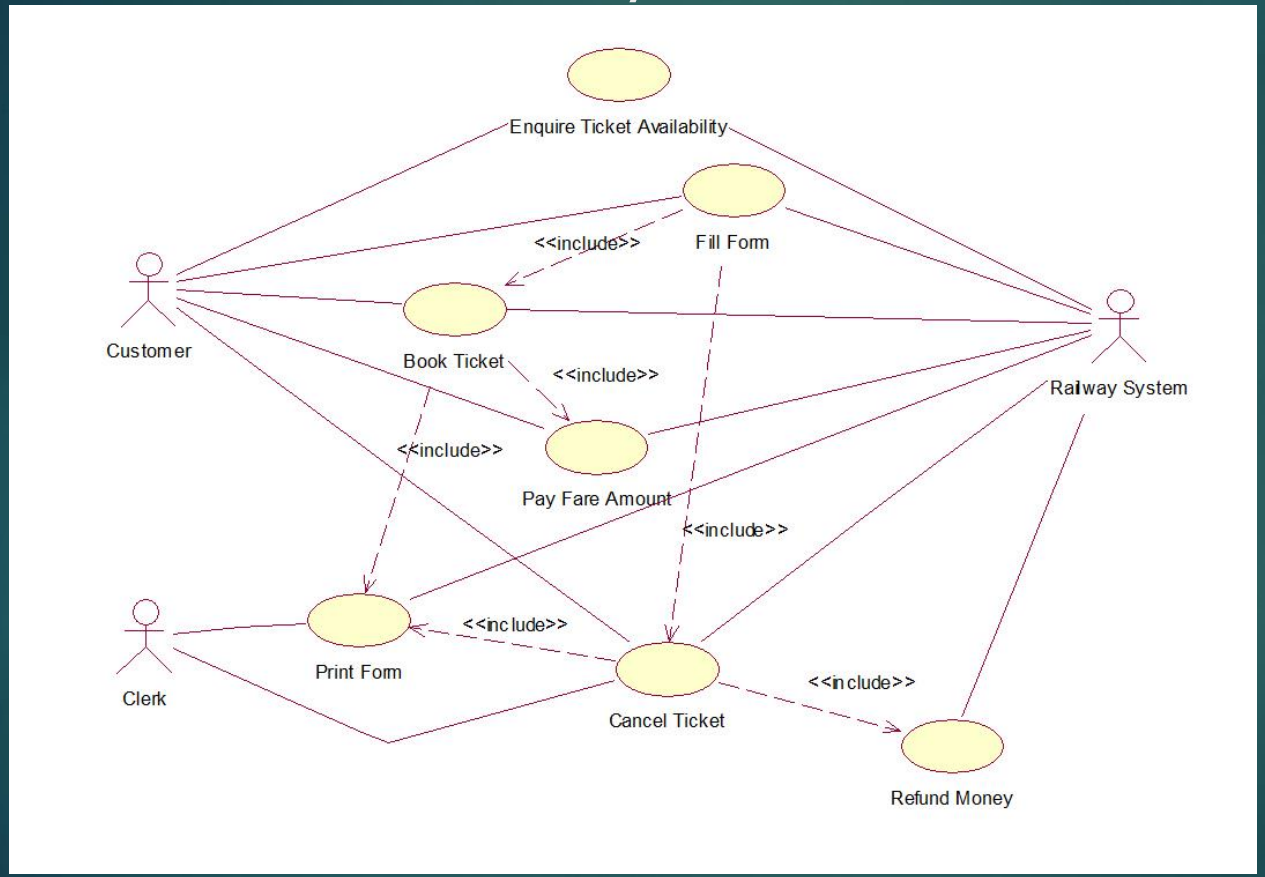
Use case diagram

•**Extend relationship:** The use case is optional and comes after the base use case. It is represented by a dashed arrow in the direction of the base use case with the notation <<extend>>.

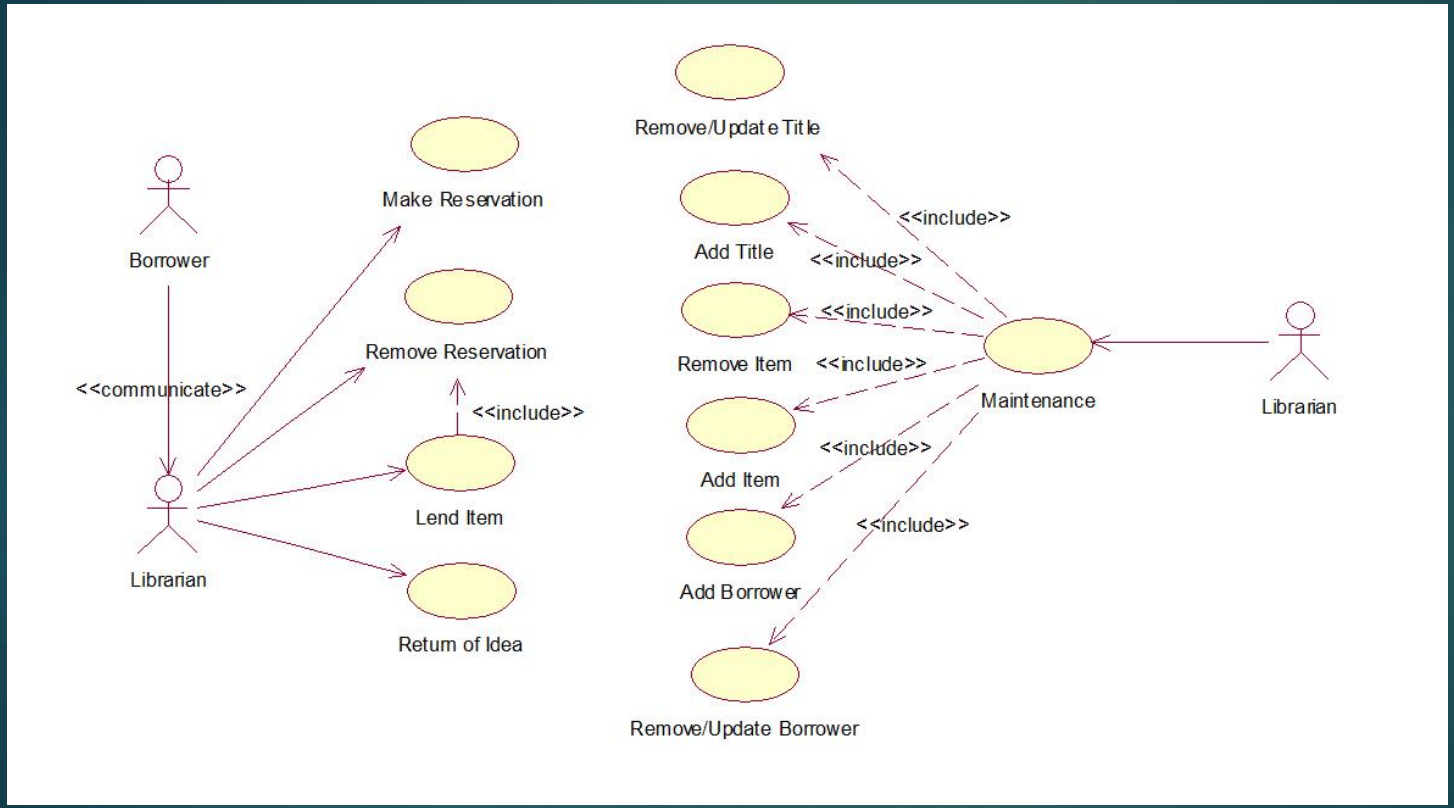
•**Include relationship:** The use case is mandatory and part of the base use case. It is represented by a dashed arrow in the direction of the included use case with the notation <<include>>.

In the above use case diagram, the use case “Read book” includes the use case “Open book”. If a reader reads the book, she must open it too, as it is mandatory for the base use case (read book). The use case “read book” extends to the use case “turn page”, which means that turning the page while reading the book is optional. The base use case in this scenario (read book) is complete without the extended use case.

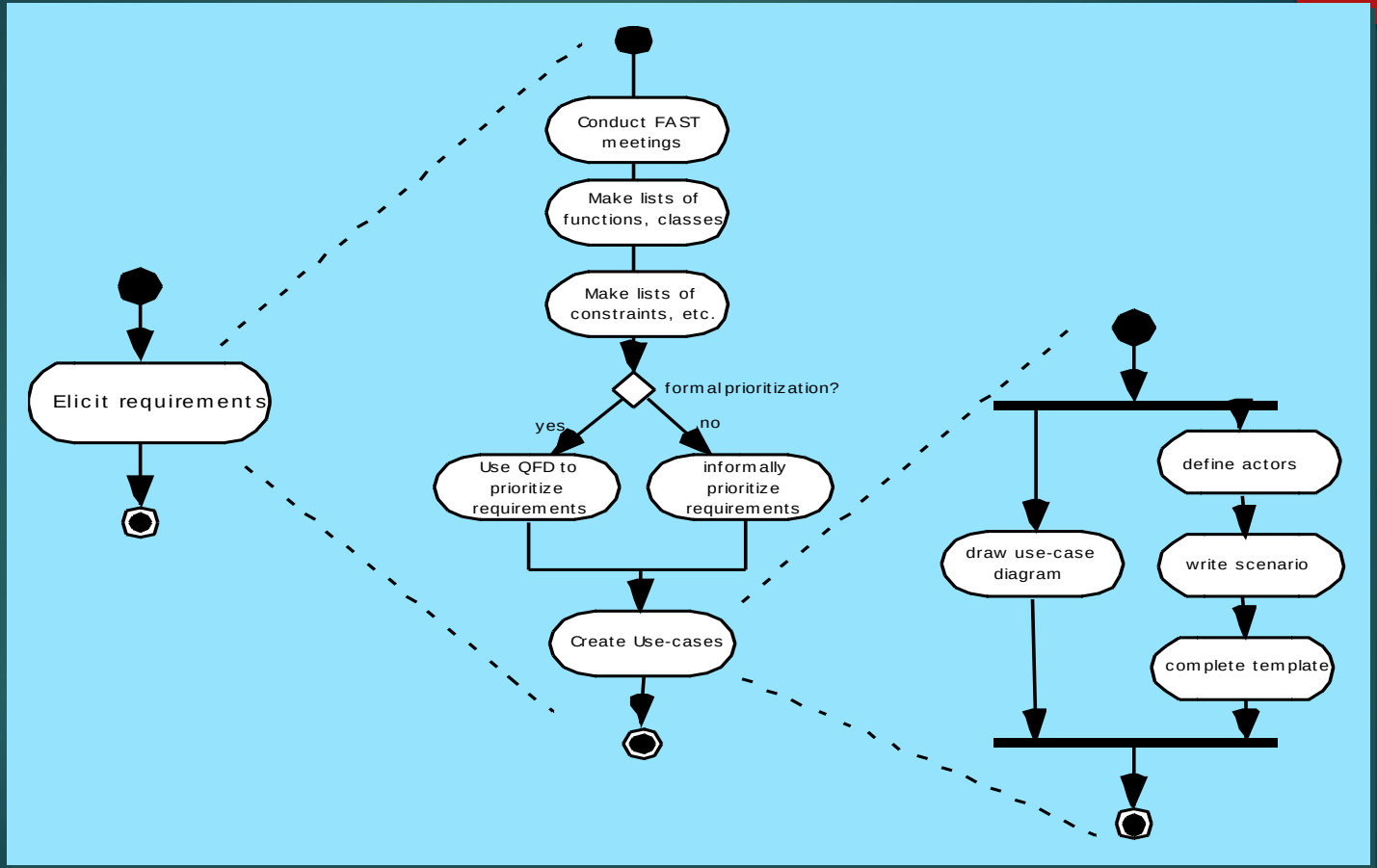
Use-case Railway reservation



Use-case Library Management

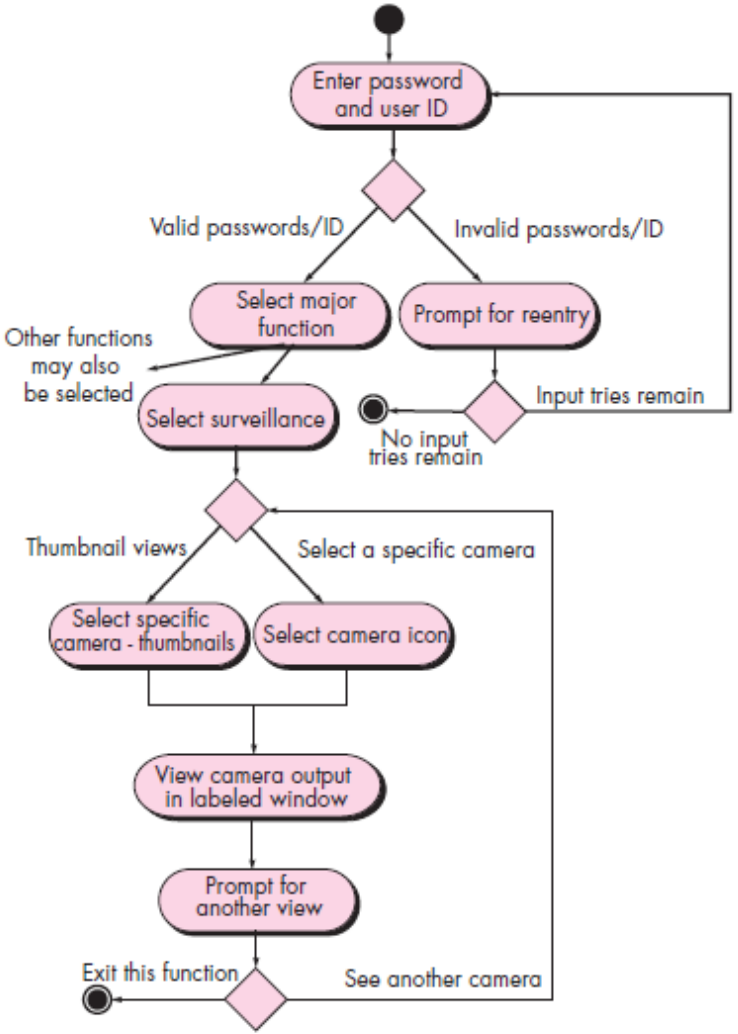


Activity diagram



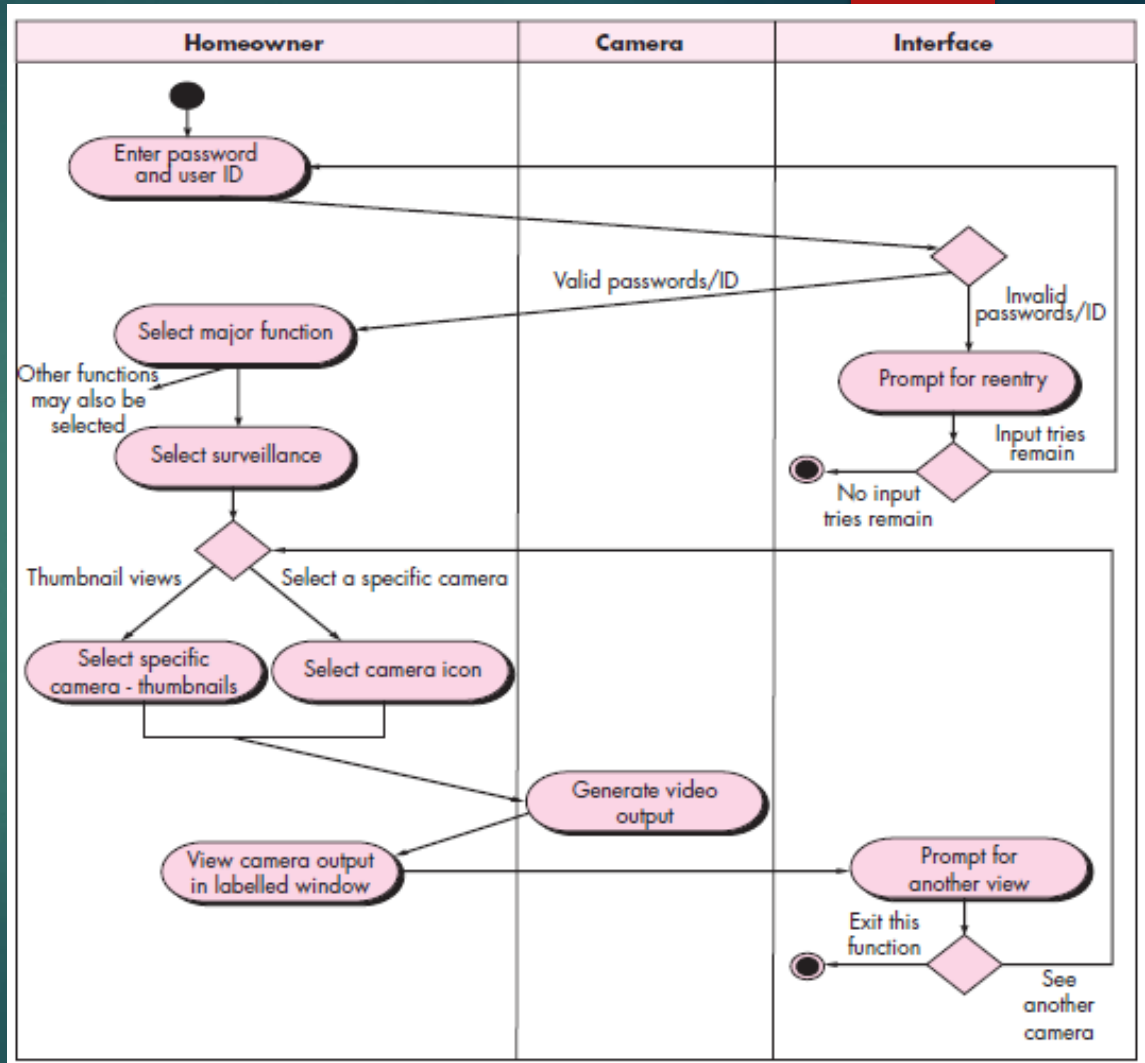
Activity Diagram

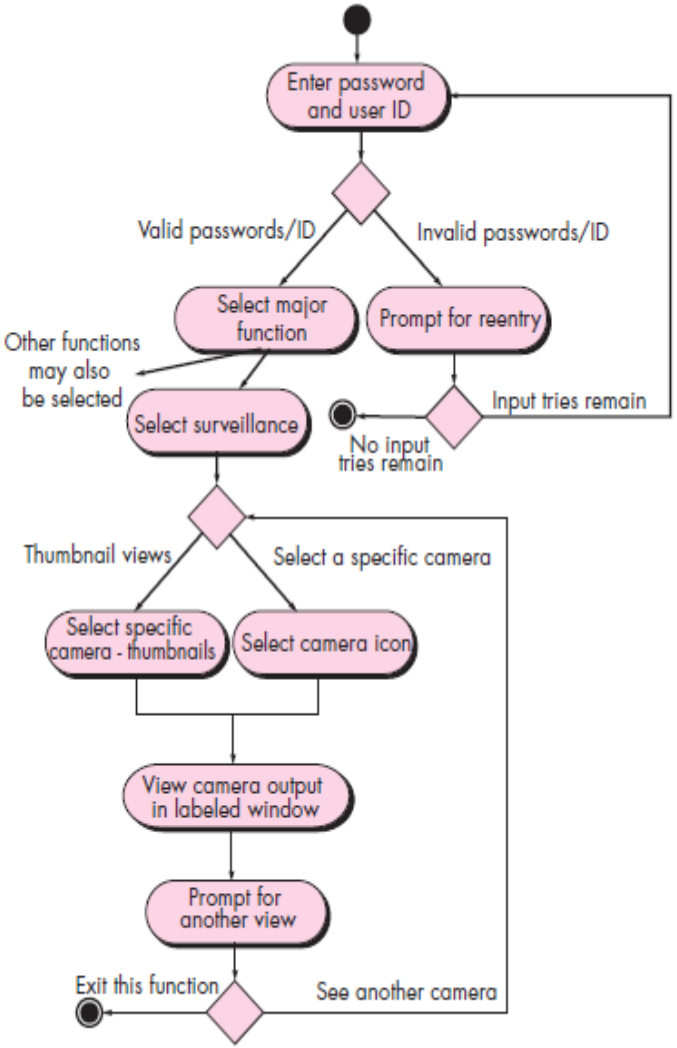
Supplements the use-case by providing a diagrammatic representation of procedural flow

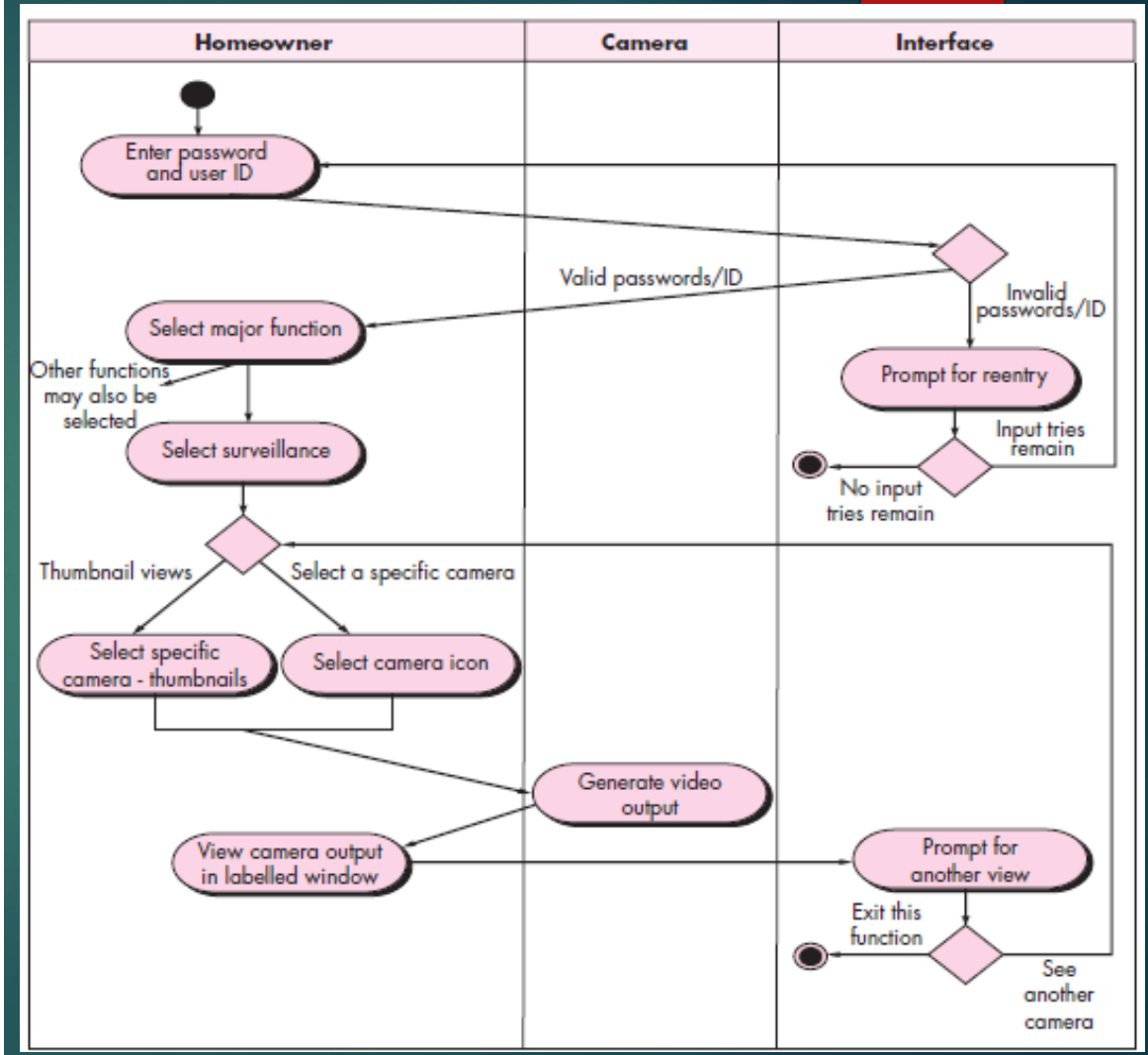


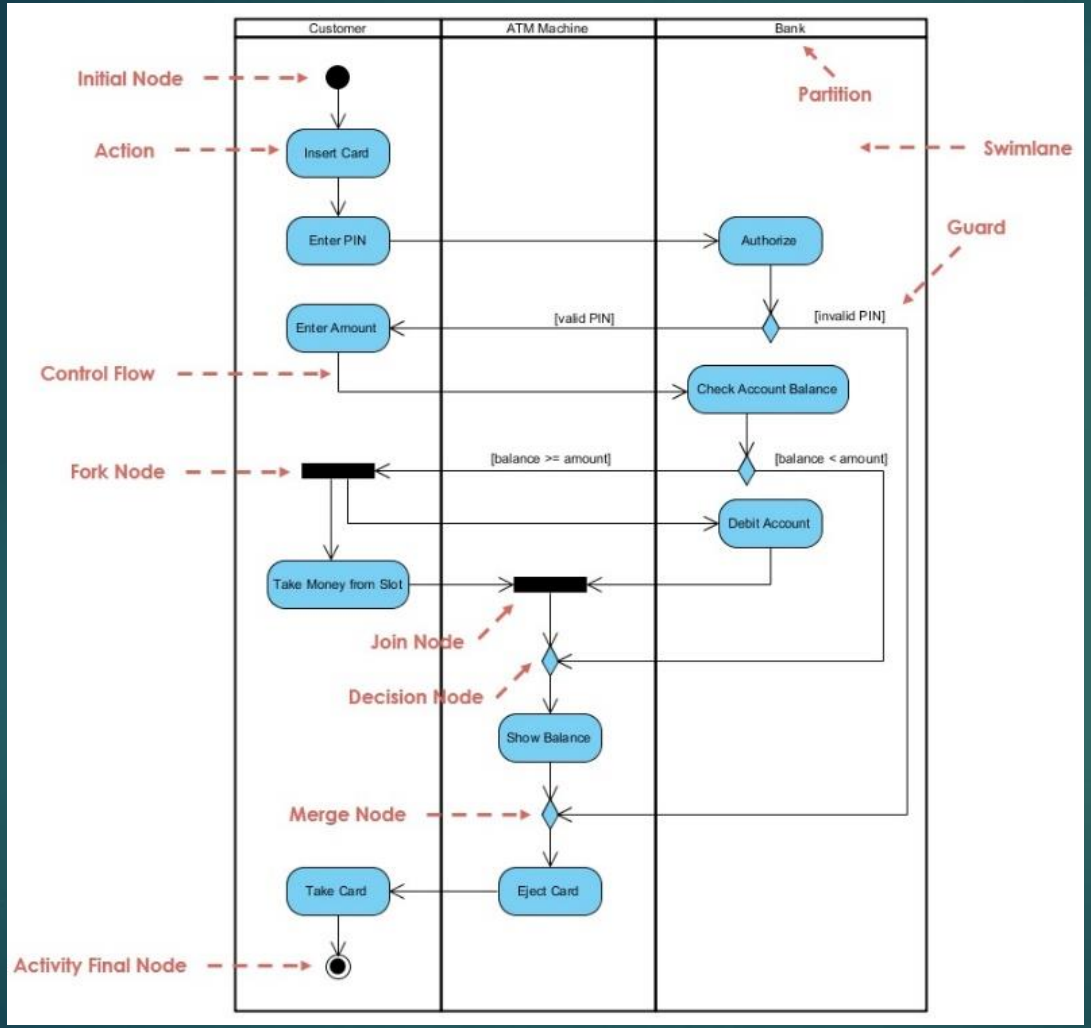
Swimlane Diagrams

Allows the modeler to represent the flow of activities described by the use-case and at the same time indicate which actor (if there are multiple actors involved in a specific use-case) or analysis class has responsibility for the action described by an activity rectangle




Enter password and user ID
Valid passwords/ID
Invalid passwords/ID
Select major function
Prompt for reentry
Other functions may also be selected
Select surveillance
Input tries remain
No input tries remain
Thumbnail views
Select a specific camera
Select specific camera - thumbnails
Select camera icon
View camera output in labeled window
Prompt for another view
Exit this function
See another camera


Homeowner
Camera
Interface
Enter password and user ID
Valid passwords/ID
Invalid passwords/ID
Select major function
Prompt for reentry
Other functions may also be selected
Input tries remain
Select surveillance
No input tries remain
Thumbnail views
Select a specific camera
Select specific camera - thumbnails
Select camera icon
Generate video output
View camera output in labelled window
Prompt for another view
Exit this function
See another camera



Kiran Bhowmick

Analysis Patterns

Pattern name: A descriptor that captures the essence of the pattern.

Intent: Describes what the pattern accomplishes or represents

Motivation: A scenario that illustrates how the pattern can be used to address the problem.

Forces and context: A description of external issues (forces) that can affect how the pattern is used and also the external issues that will be resolved when the pattern is applied.

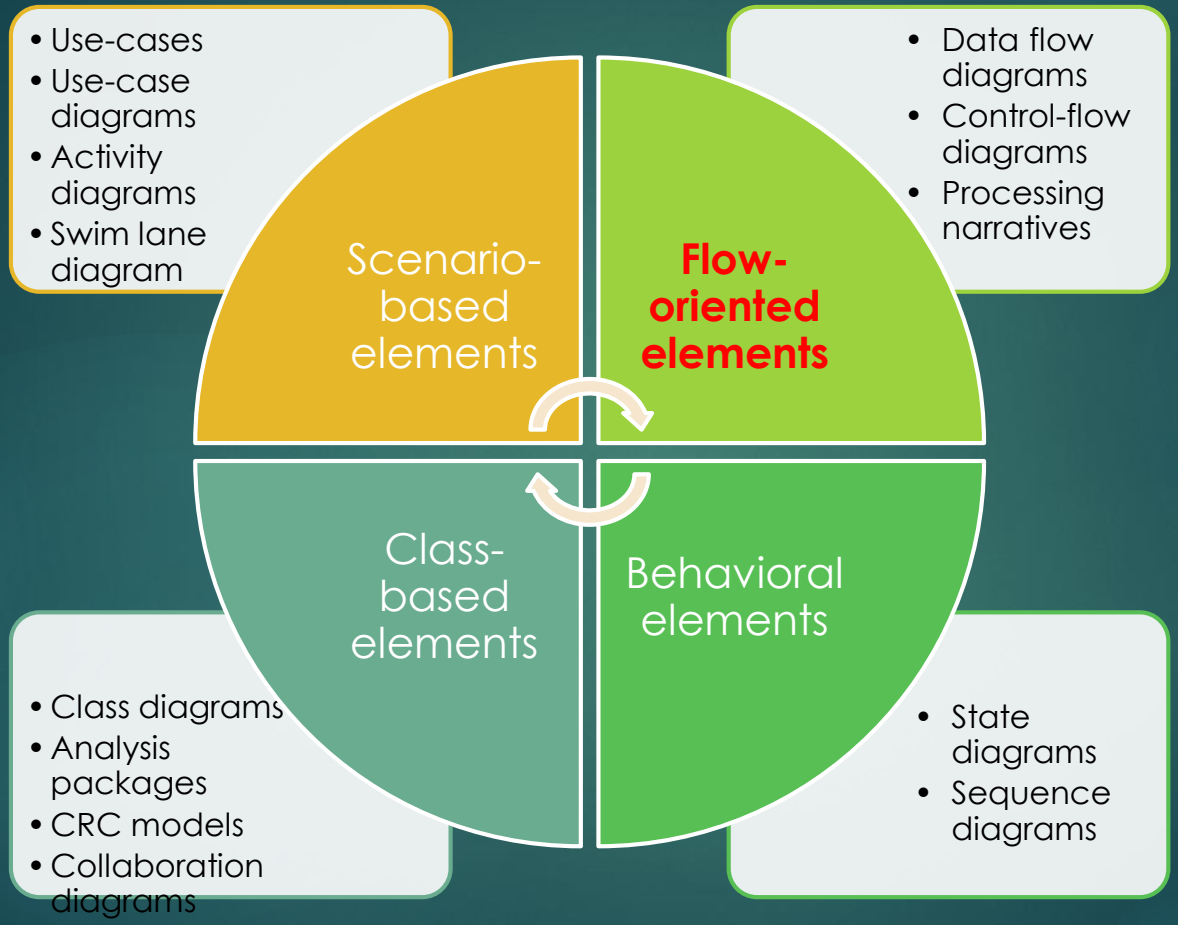
Solution: A description of how the pattern is applied to solve the problem with an emphasis on structural and behavioral issues.

Consequences: Addresses what happens when the pattern is applied and what trade-offs exist during its application.

Design: Discusses how the analysis pattern can be achieved through the use of known design patterns.

Known uses: Examples of uses within actual systems.

Related patterns: On e or more analysis patterns that are related to the named pattern because (1) it is commonly used with the named pattern; (2) it is structurally similar to the named pattern; (3) it is a variation of the named pattern.


• Use-cases
• Use-case diagrams
• Activity diagrams
• Swim lane diagram
Scenario-based elements
Flow-oriented elements
• Data flow diagrams
• Control-flow diagrams
• Processing narratives
Class-based elements
Behavioral elements
• Class diagrams
• Analysis packages
• CRC models
• Collaboration diagrams
• State diagrams
• Sequence diagrams

Flow-Oriented Modeling

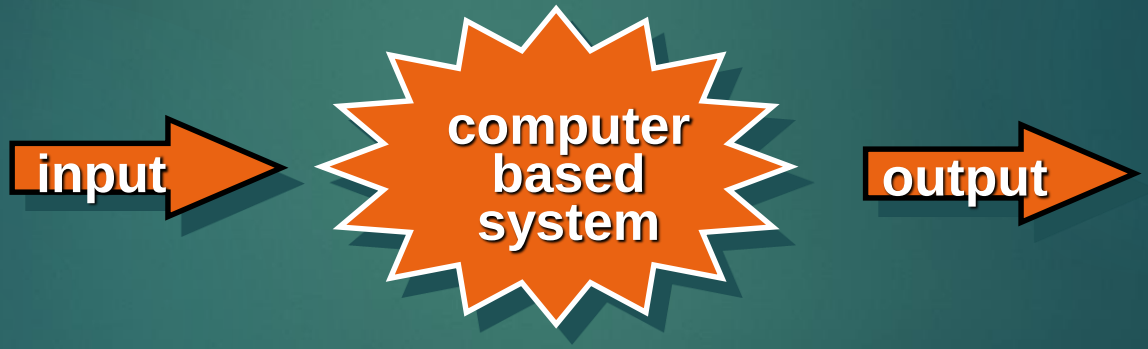
Represents how data objects are transformed as they move through the system

A data flow diagram (DFD) is the diagrammatic form that is used

Considered by many to be an ‘old school’ approach, flow-oriented modeling continues to provide a view of the system that is unique—it should be used to supplement other analysis model elements

The Flow Model

Every computer-based system is an information transform ....



Flow Modeling Notation

external entity

process

data flow

data store

External Entity

A producer or consumer of data

Examples: a person, a device, a sensor

Another example: computer-based system

Data must always originate somewhere and must always be sent to something

Process

- **A data transformer (changes input to output)**

Examples: compute taxes, determine area, format report, display graph

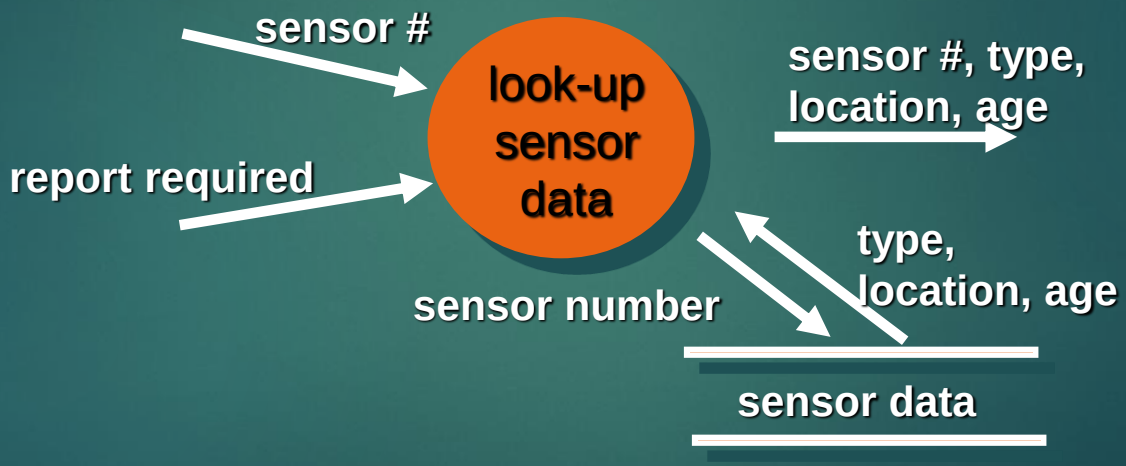
Data must always be processed in some way to achieve system function

Data Flow

Data flows through a system, beginning as input and be transformed into output.

Data Stores

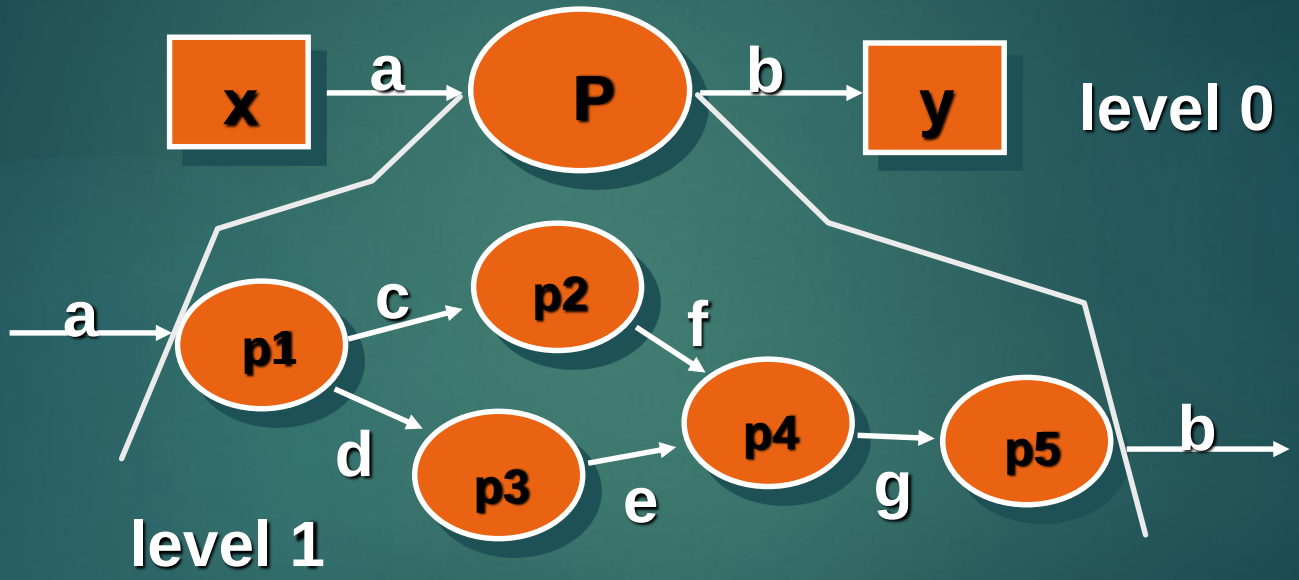
Data is often stored for later use.



Data Flow Diagramming: Guidelines

- all icons must be labeled with meaningful names
- the DFD evolves through a number of levels of detail
- always begin with a context level diagram (also called level 0)
- always show external entities at level 0
- always label data flow arrows
- do not represent procedural logic

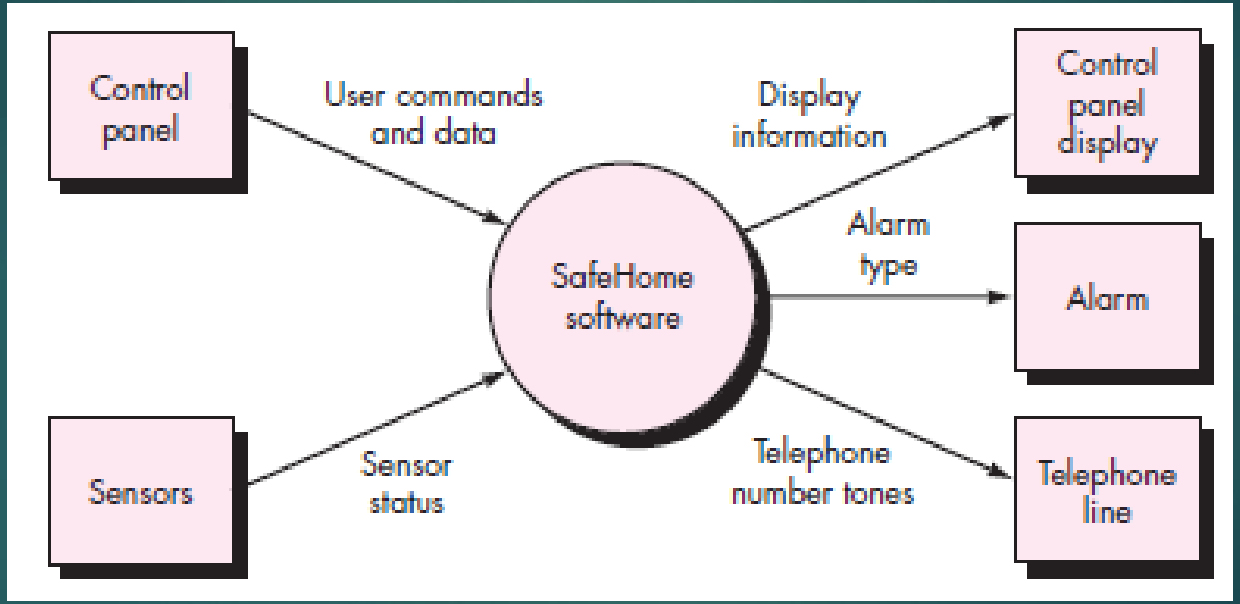
The Data Flow Hierarchy



Constructing a DFD

- review the data model to isolate data objects and use a grammatical parse to determine "operations"
- determine external entities (producers and consumers of data)
- create a level 0 DFD

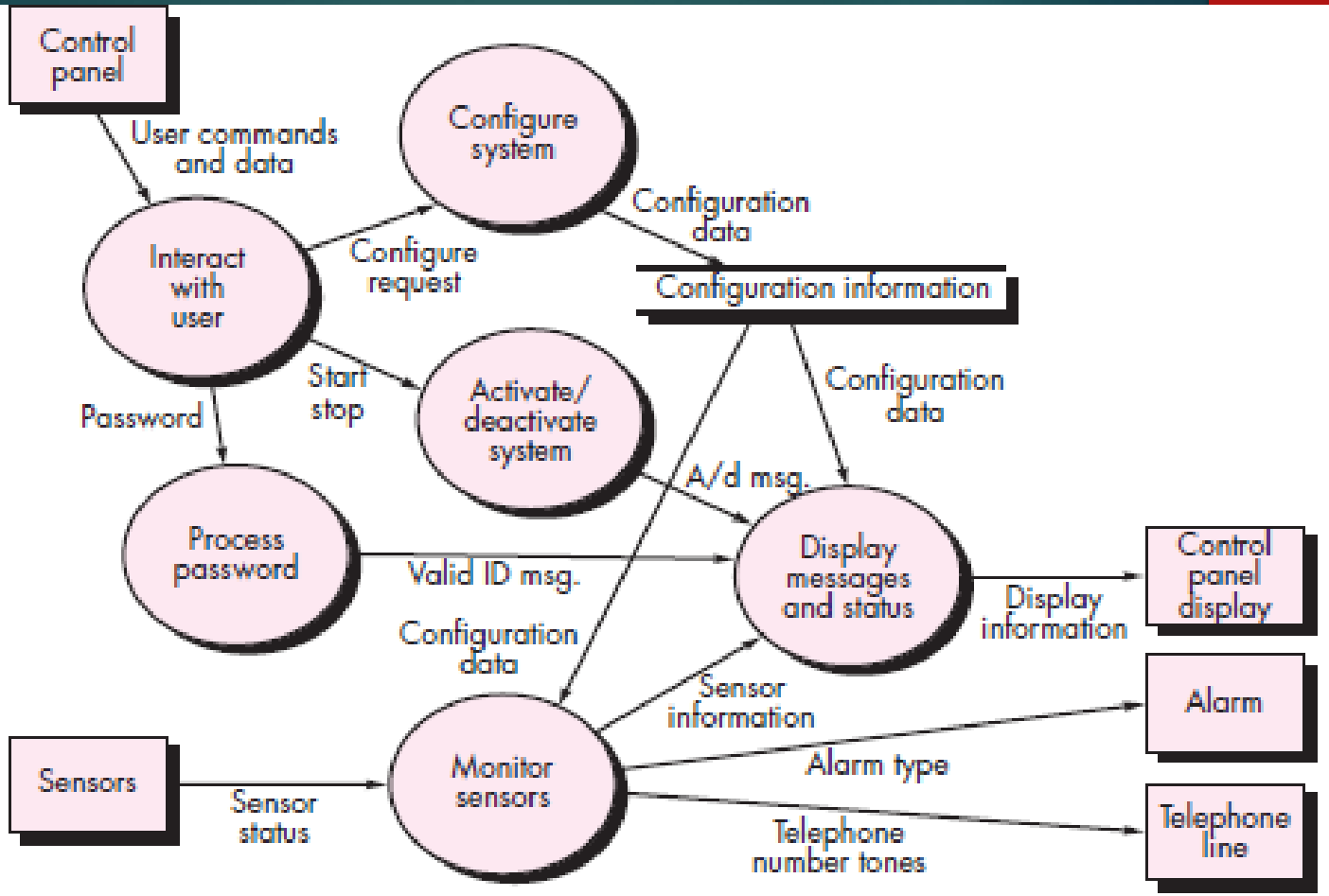
Level 0 DFD Example



Constructing a DFD

- write a narrative describing the transform
- parse to determine next level transforms
- “balance” the flow to maintain data flow continuity
- develop a level 1 DFD
- use a 1:5 (approx.) expansion ratio
- performing a grammatical parse on the processing narrative for a bubble at any DFD level, you can generate much useful information about how to proceed with the refinement to the next level. Using this information, a level 1 DFD can be designed

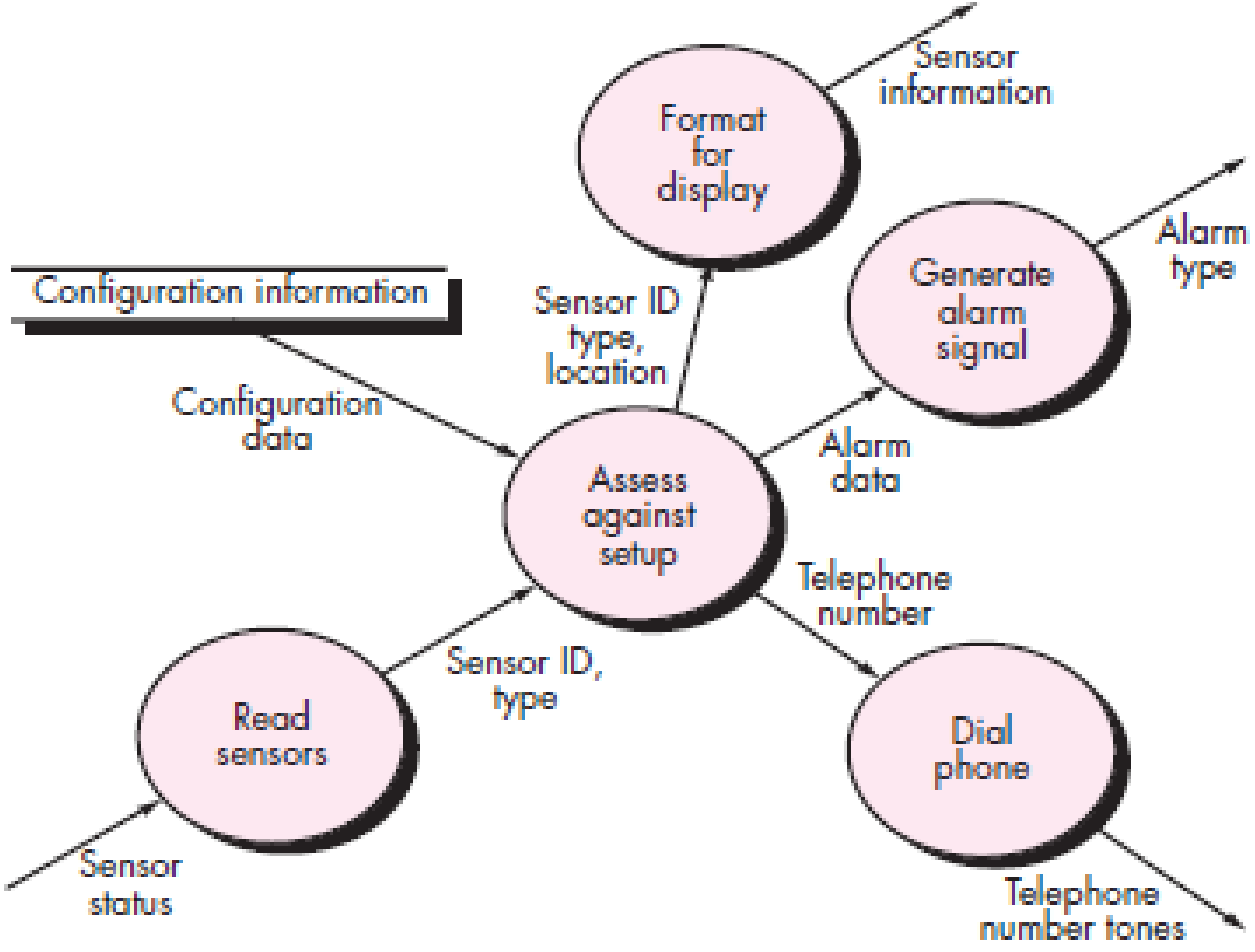
Level 1 DFD for *SafeHome* security function



Constructing a DFD

- The processes represented at DFD level 1 can be further refined into lower levels. For example, the process monitor sensors can be refined into a level 2 DFD
- The refinement of DFDs continues until each bubble performs a simple function. That is, until the process represented by the bubble performs a function that would be easily implemented as a program component

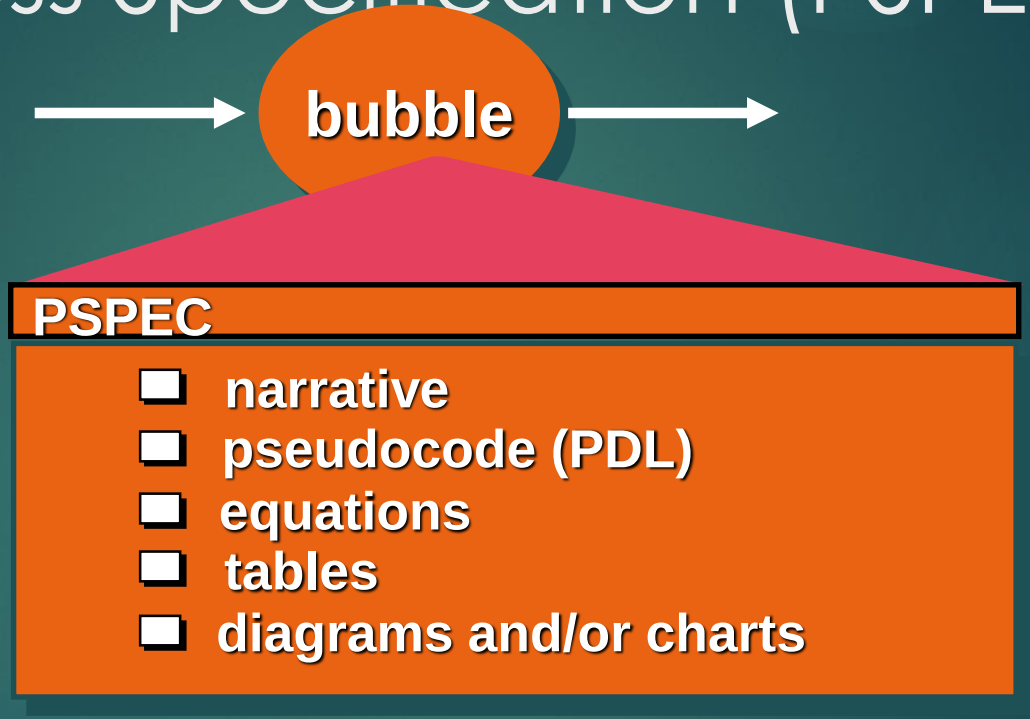
Level 2 DFD that refines the *monitor sensors* process



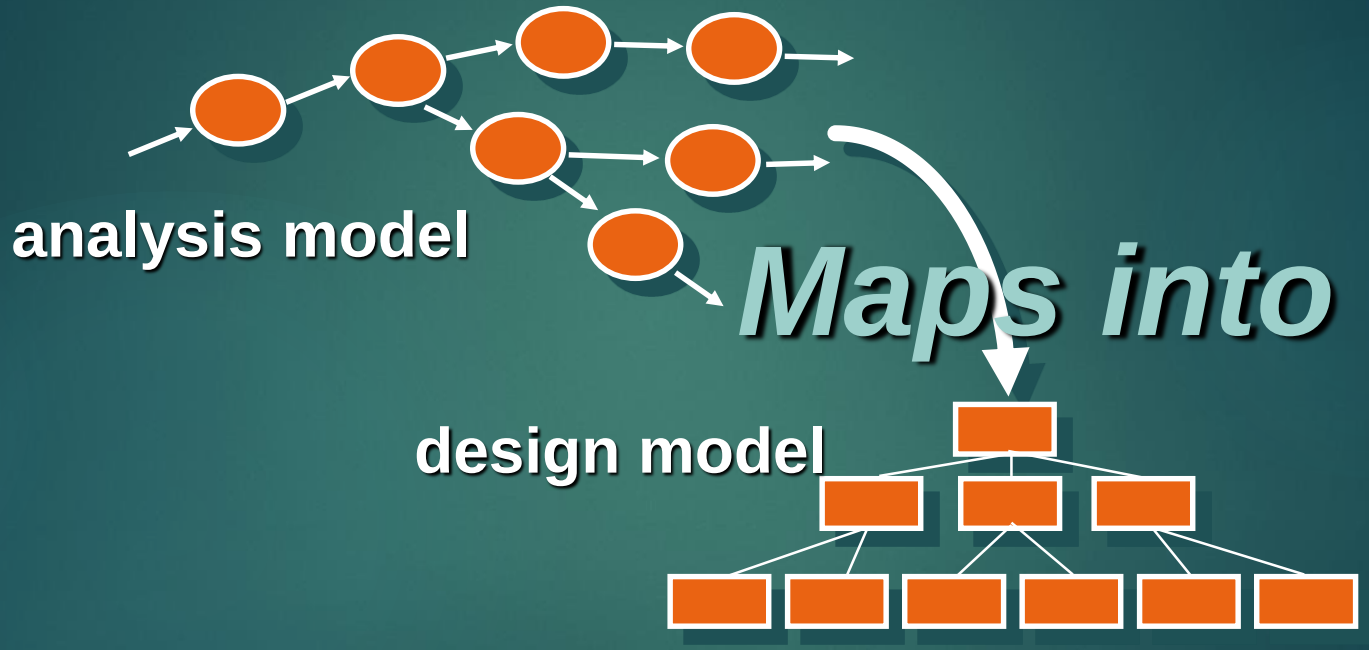
Flow Modeling Notes

- each bubble is refined until it does just one thing
- the expansion ratio decreases as the number of levels increase
- most systems require between 3 and 7 levels for an adequate flow model
- a single data flow item (arrow) may be expanded as levels increase (data dictionary provides information)

Process Specification (PSPEC)



DFDs: A Look Ahead

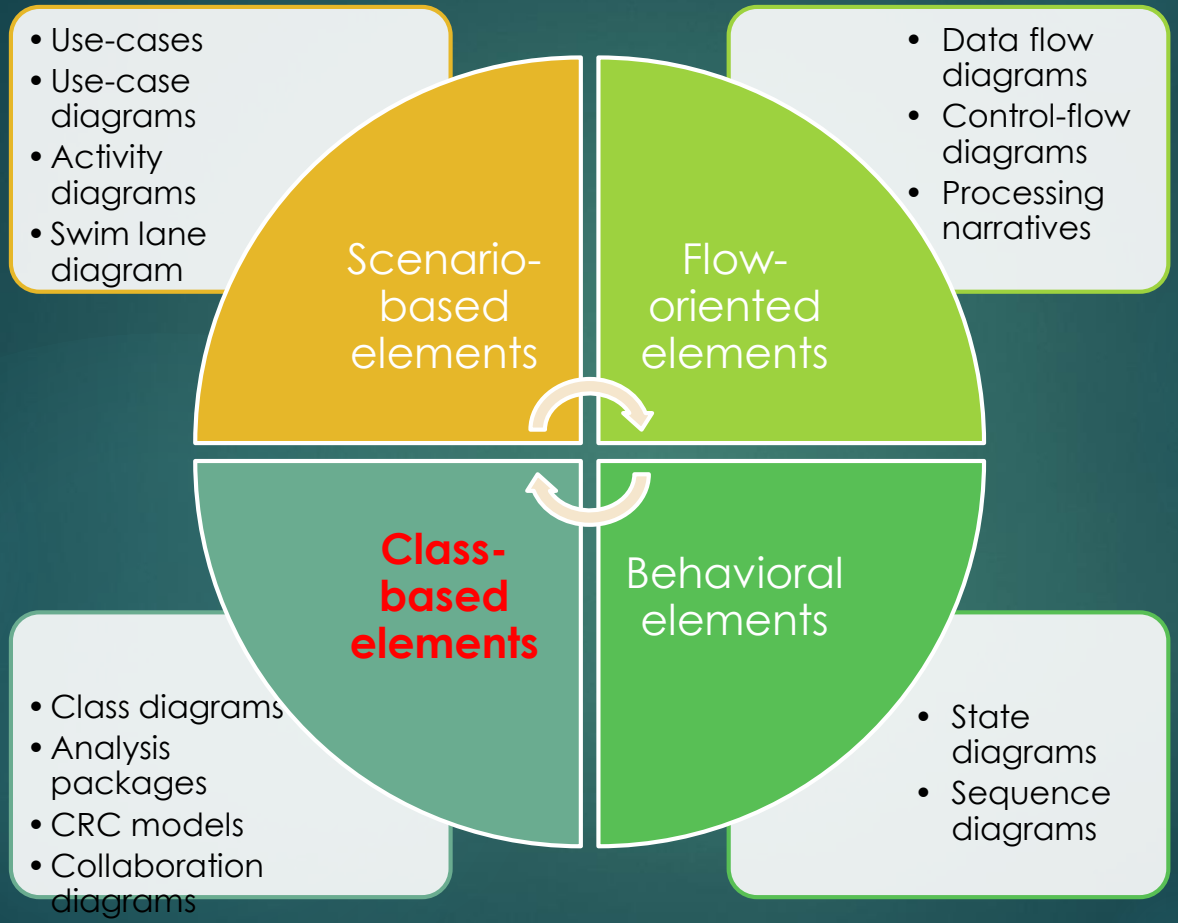


Control Flow Diagrams

- Represents “events” and the processes that manage events
- An “event” is a Boolean condition that can be ascertained by:
  - listing all sensors that are "read" by the software.
  - listing all interrupt conditions.
  - listing all "switches" that are actuated by an operator.
  - listing all data conditions.
  - recalling the noun/verb parse that was applied to the processing narrative, review all "control items" as possible CSPEC inputs/outputs.

The Control Model

- the control flow diagram is "superimposed" on the DFD and shows events that control the processes noted in the DFD
- control flows—events and control items—are noted by dashed arrows
- a vertical bar implies an input to or output from a control spec (CSPEC) — a separate specification that describes how control is handled
- a dashed arrow entering a vertical bar is an input to the CSPEC
- a dashed arrow leaving a process implies a data condition
- a dashed arrow entering a process implies a control input read directly by the process
- control flows do not physically activate/deactivate the processes—this is done via the CSPEC


• Use-cases
• Use-case diagrams
• Activity diagrams
• Swim lane diagram
Scenario-based elements
Flow-oriented elements
• Data flow diagrams
• Control-flow diagrams
• Processing narratives
Class-based elements
Behavioral elements
• Class diagrams
• Analysis packages
• CRC models
• Collaboration diagrams
• State diagrams
• Sequence diagrams

Class-Based Modeling

- Identify analysis classes by examining the problem statement
- Use a "grammatical parse" to isolate potential classes
- Identify the attributes of each class
- Identify operations that manipulate the attributes

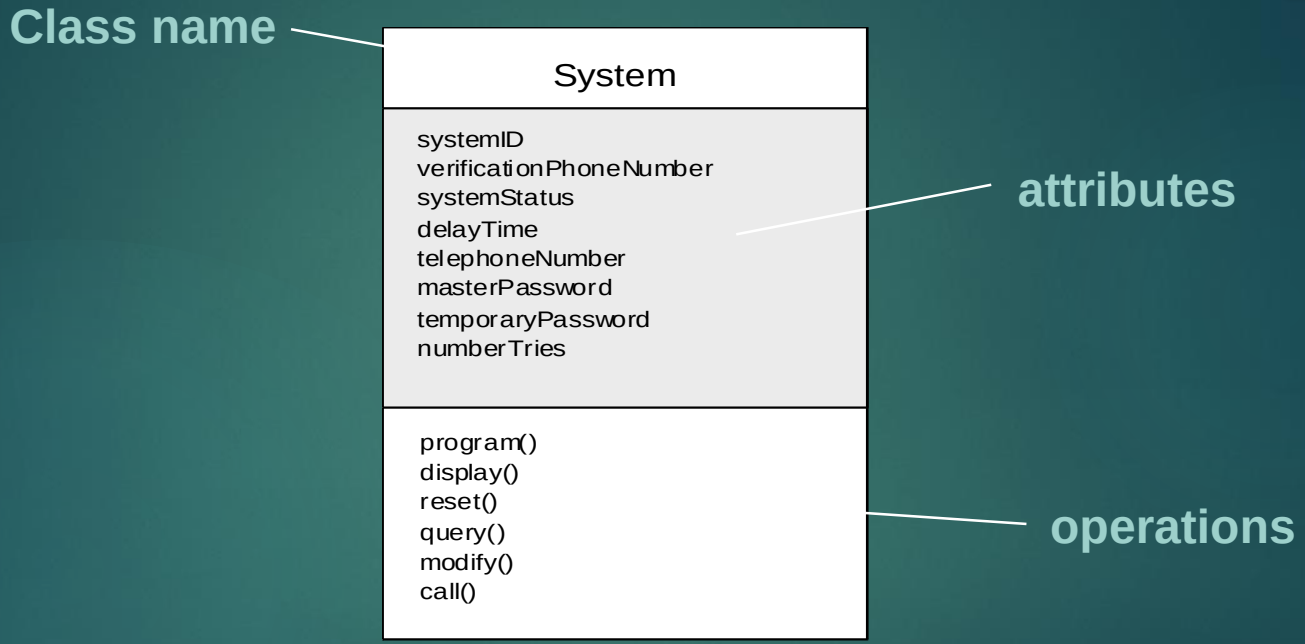
Analysis Classes

- *External entities* (e.g., other systems, devices, people) that produce or consume information to be used by a computer-based system.
- *Things* (e.g, reports, displays, letters, signals) that are part of the information domain for the problem.
- *Occurrences or events* (e.g., a property transfer or the completion of a series of robot movements) that occur within the context of system operation.
- *Roles* (e.g., manager, engineer, salesperson) played by people who interact with the system.
- *Organizational units* (e.g., division, group, team) that are relevant to an application.
- *Places* (e.g., manufacturing floor or loading dock) that establish the context of the problem and the overall function of the system.
- *Structures* (e.g., sensors, four-wheeled vehicles, or computers) that define a class of objects or related classes of objects.

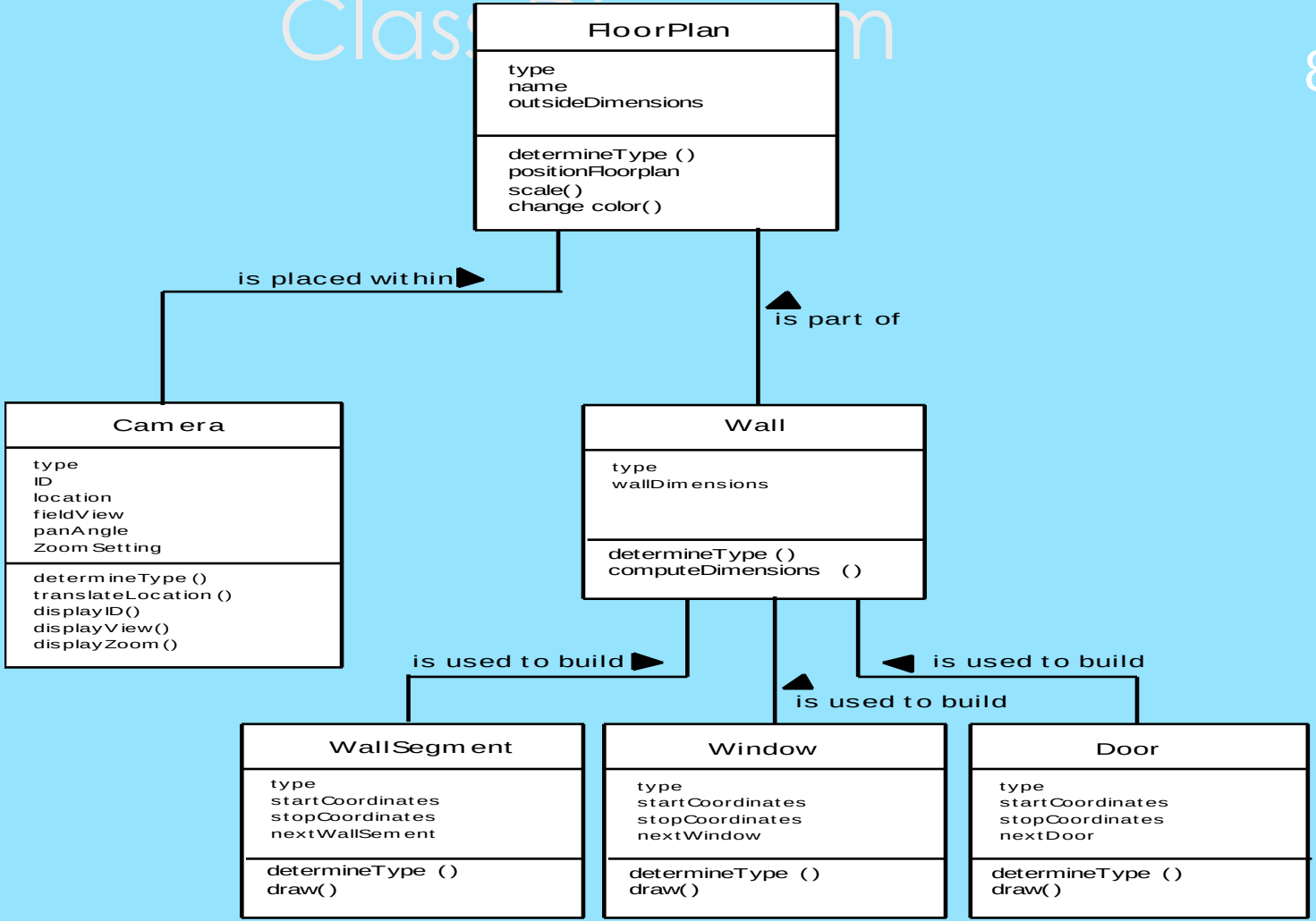
Selecting Classes—Criteria

- retained information
- needed services
- multiple attributes
- common attributes
- common operations
- essential requirements

Class Diagram



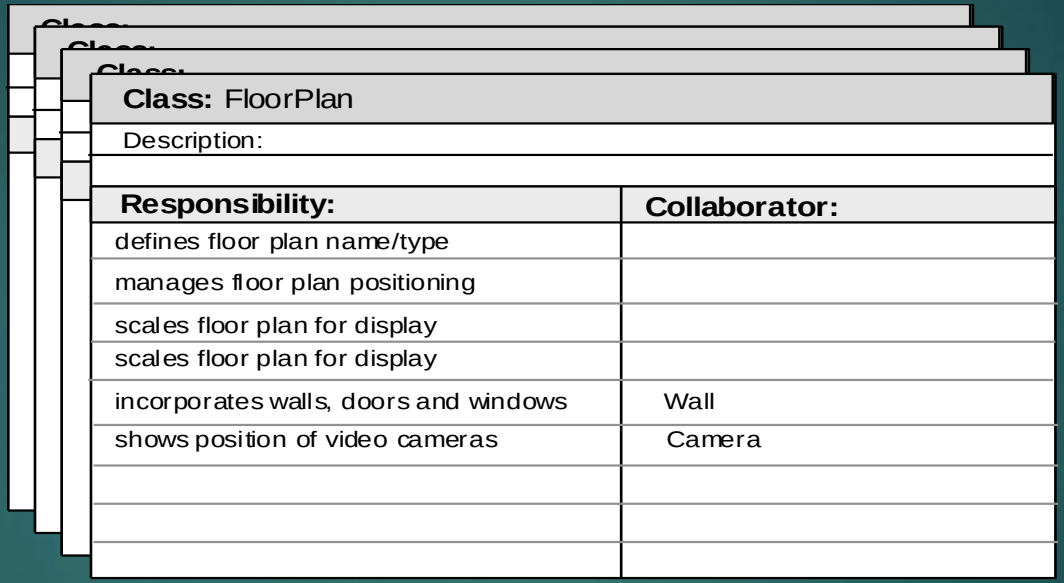
Class Diagram



CRC Modeling

- Analysis classes have "responsibilities"
  - *Responsibilities* are the attributes and operations encapsulated by the class
- Analysis classes collaborate with one another
  - *Collaborators* are those classes that are required to provide a class with the information needed to complete a responsibility.
  - In general, a collaboration implies either a request for information or a request for some action.

CRC Modeling



Class: FloorPlan

Description:

Responsibility:	**Collaborator:**
defines floor plan name/type	
manages floor plan positioning	
scales floor plan for display	
scales floor plan for display	
incorporates walls, doors and windows	Wall
shows position of video cameras	Camera

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.

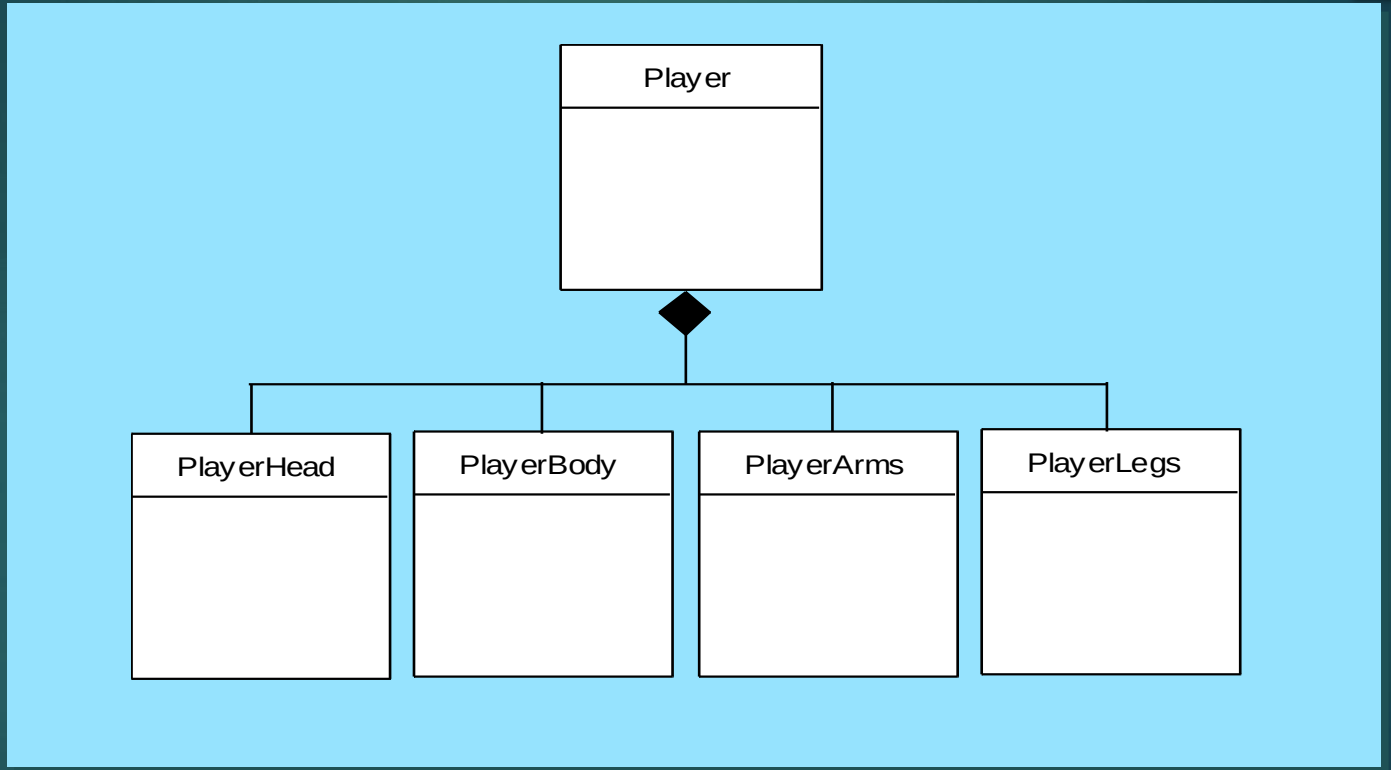
Responsibilities

- System intelligence should be distributed across classes to best address the needs of the problem
- Each responsibility should be stated as generally as possible
- Information and the behavior related to it should reside within the same class
- Information about one thing should be localized with a single class, not distributed across multiple classes.
- Responsibilities should be shared among related classes, when appropriate.

Collaborations

- Classes fulfill their responsibilities in one of two ways:
  - A class can use its own operations to manipulate its own attributes, thereby fulfilling a particular responsibility, or
  - a class can collaborate with other classes.
- Collaborations identify relationships between classes
- Collaborations are identified by determining whether a class can fulfill each responsibility itself
- three different generic relationships between classes [WIR90]:
  - the *is-part-of* relationship
  - the *has-knowledge-of* relationship
  - the *depends-upon* relationship

Composite Aggregate Class



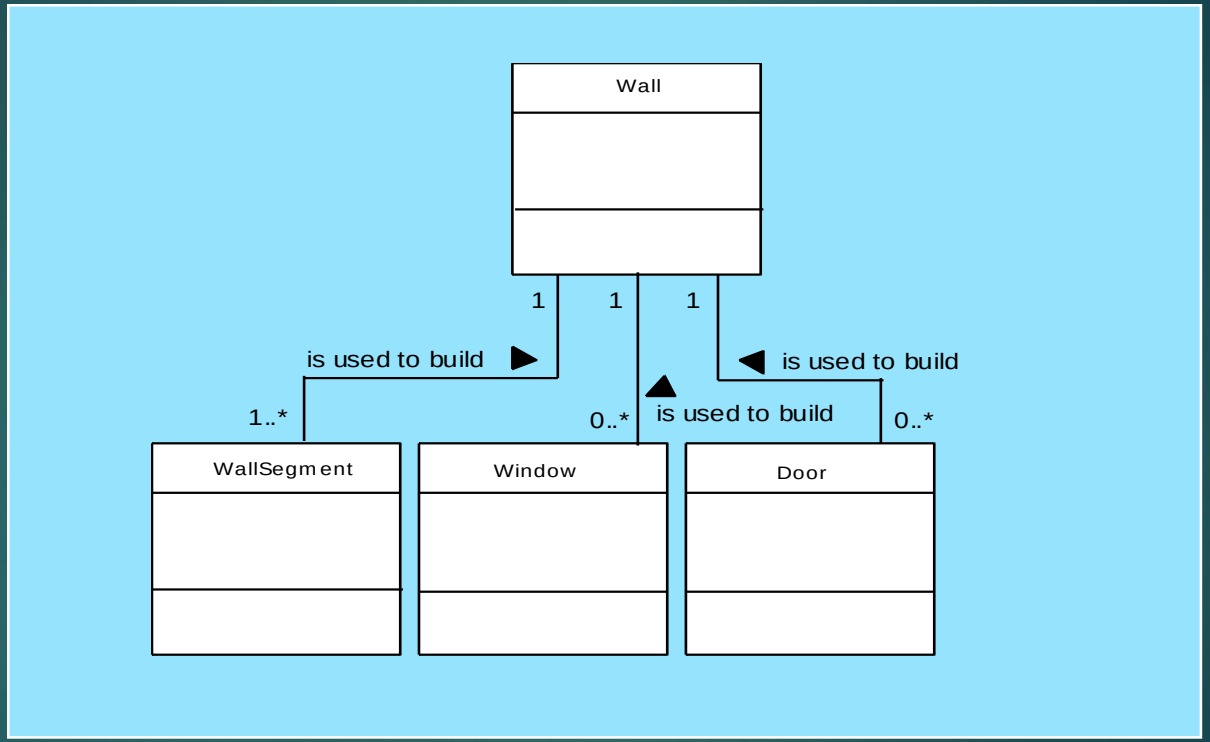
Reviewing the CRC Model

- All participants in the review (of the CRC model) are given a subset of the CRC model index cards.
  - Cards that collaborate should be separated (i.e., no reviewer should have two cards that collaborate).
- All use-case scenarios (and corresponding use-case diagrams) should be organized into categories.
- The review leader reads the use-case deliberately.
  - As the review leader comes to a named object, she passes a token to the person holding the corresponding class index card.
- When the token is passed, the holder of the class card is asked to describe the responsibilities noted on the card.
  - The group determines whether one (or more) of the responsibilities satisfies the use-case requirement.
- If the responsibilities and collaborations noted on the index cards cannot accommodate the use-case, modifications are made to the cards.
  - This may include the definition of new classes (and corresponding CRC index cards) or the specification of new or revised responsibilities or collaborations on existing cards.

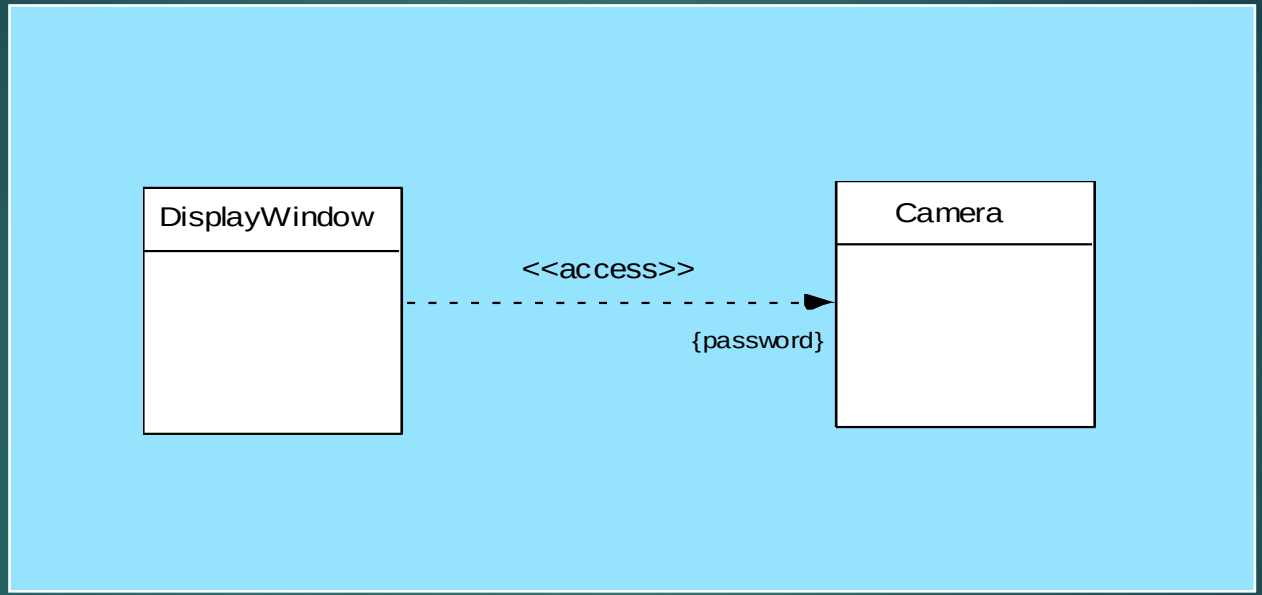
Associations and Dependencies

- Two analysis classes are often related to one another in some fashion
  - In UML these relationships are called *associations*
  - Associations can be refined by indicating *multiplicity* (the term *cardinality* is used in data modeling
- In many instances, a client-server relationship exists between two analysis classes.
  - In such cases, a client-class depends on the server-class in some way and a *dependency relationship* is established

Multiplicity



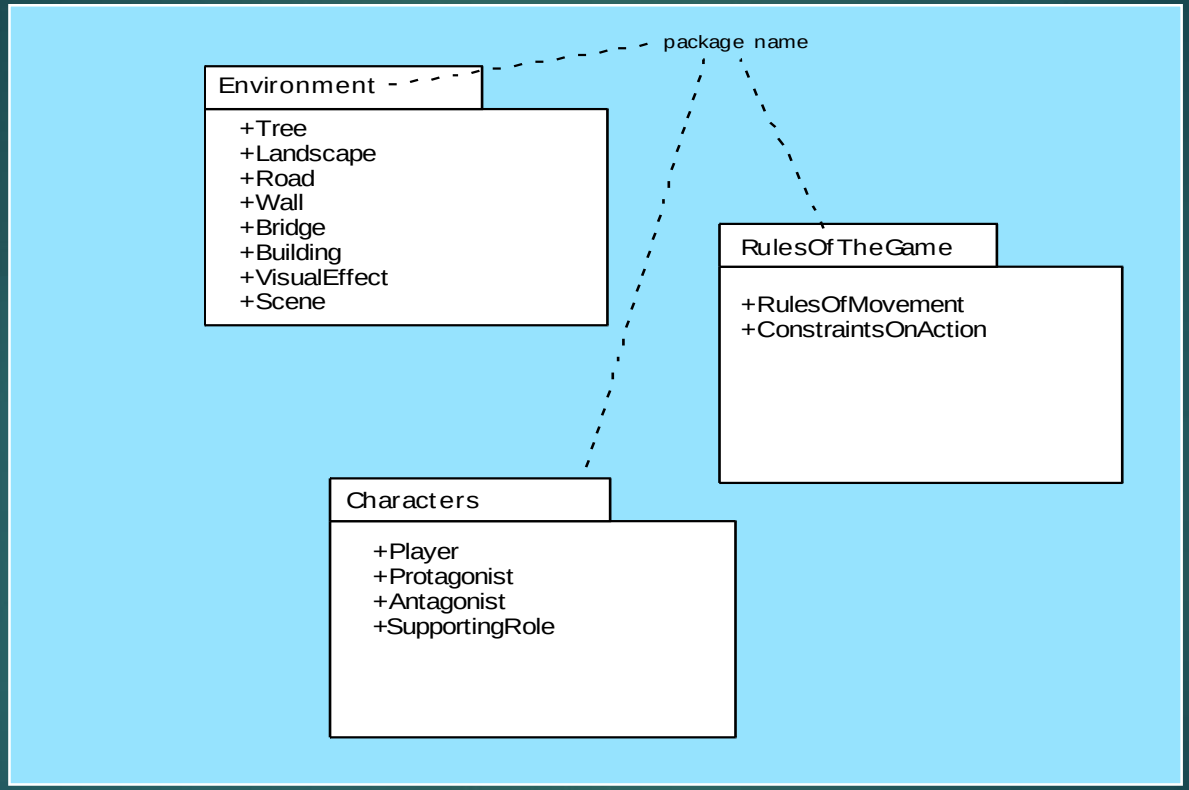
Dependencies

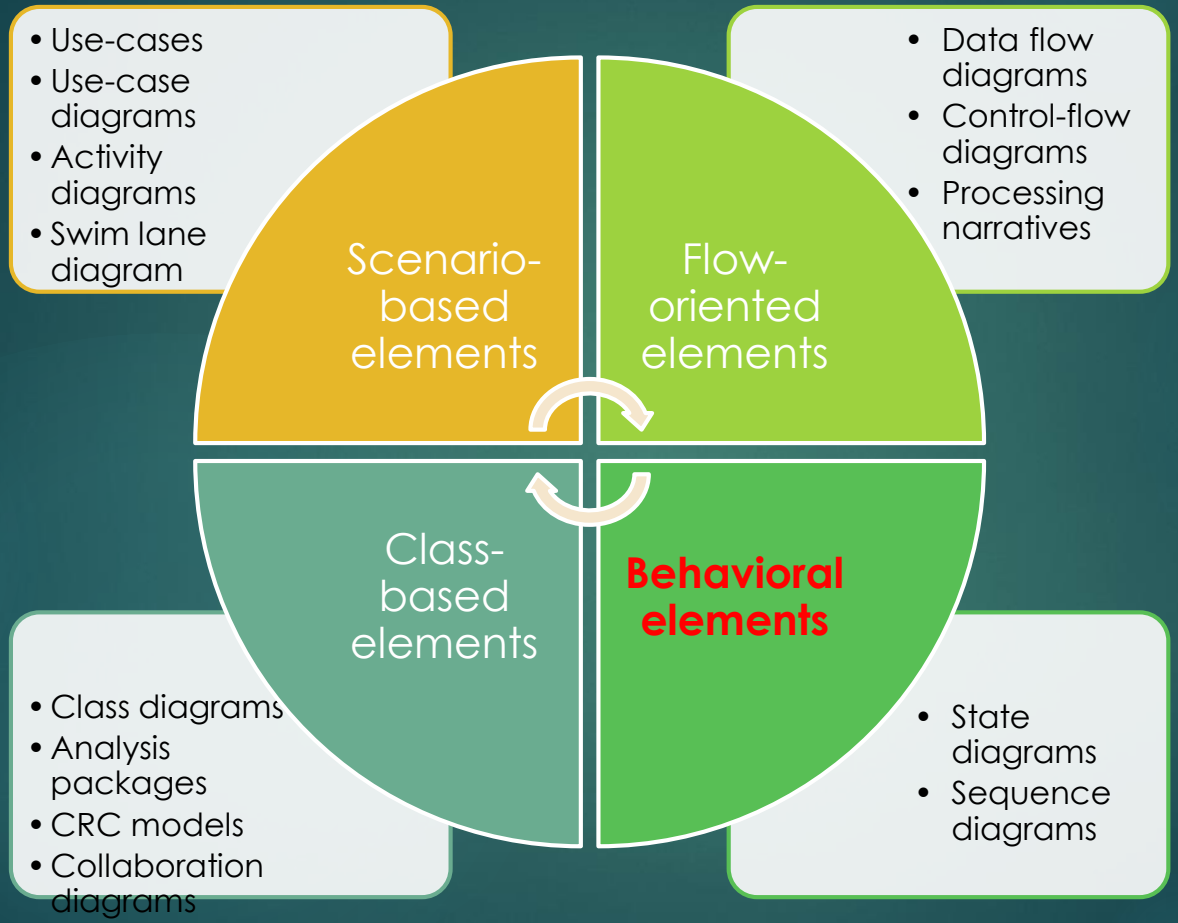


Analysis Packages

- Various elements of the analysis model (e.g., use-cases, analysis classes) are categorized in a manner that packages them as a grouping
- The plus sign preceding the analysis class name in each package indicates that the classes have public visibility and are therefore accessible from other packages.
- Other symbols can precede an element within a package. A minus sign indicates that an element is hidden from all other packages and a # symbol indicates that an element is accessible only to packages contained within a given package.

Analysis Packages




• Use-cases
• Use-case diagrams
• Activity diagrams
• Swim lane diagram
Scenario-based elements
Flow-oriented elements
• Data flow diagrams
• Control-flow diagrams
• Processing narratives
Class-based elements
Behavioral elements
• Class diagrams
• Analysis packages
• CRC models
• Collaboration diagrams
• State diagrams
• Sequence diagrams

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.

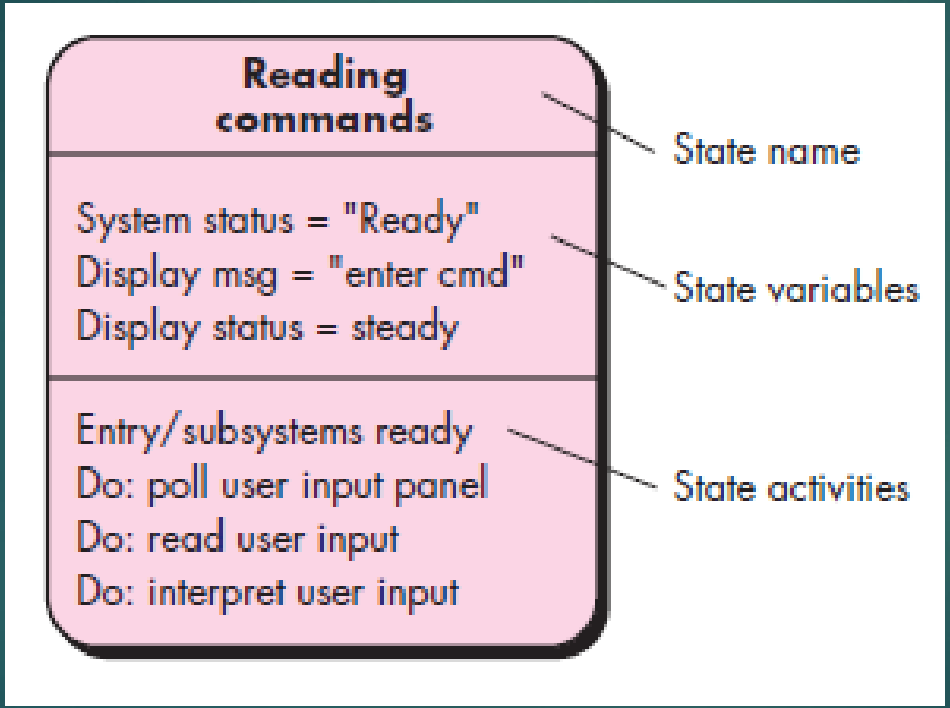
State Representations

- 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.

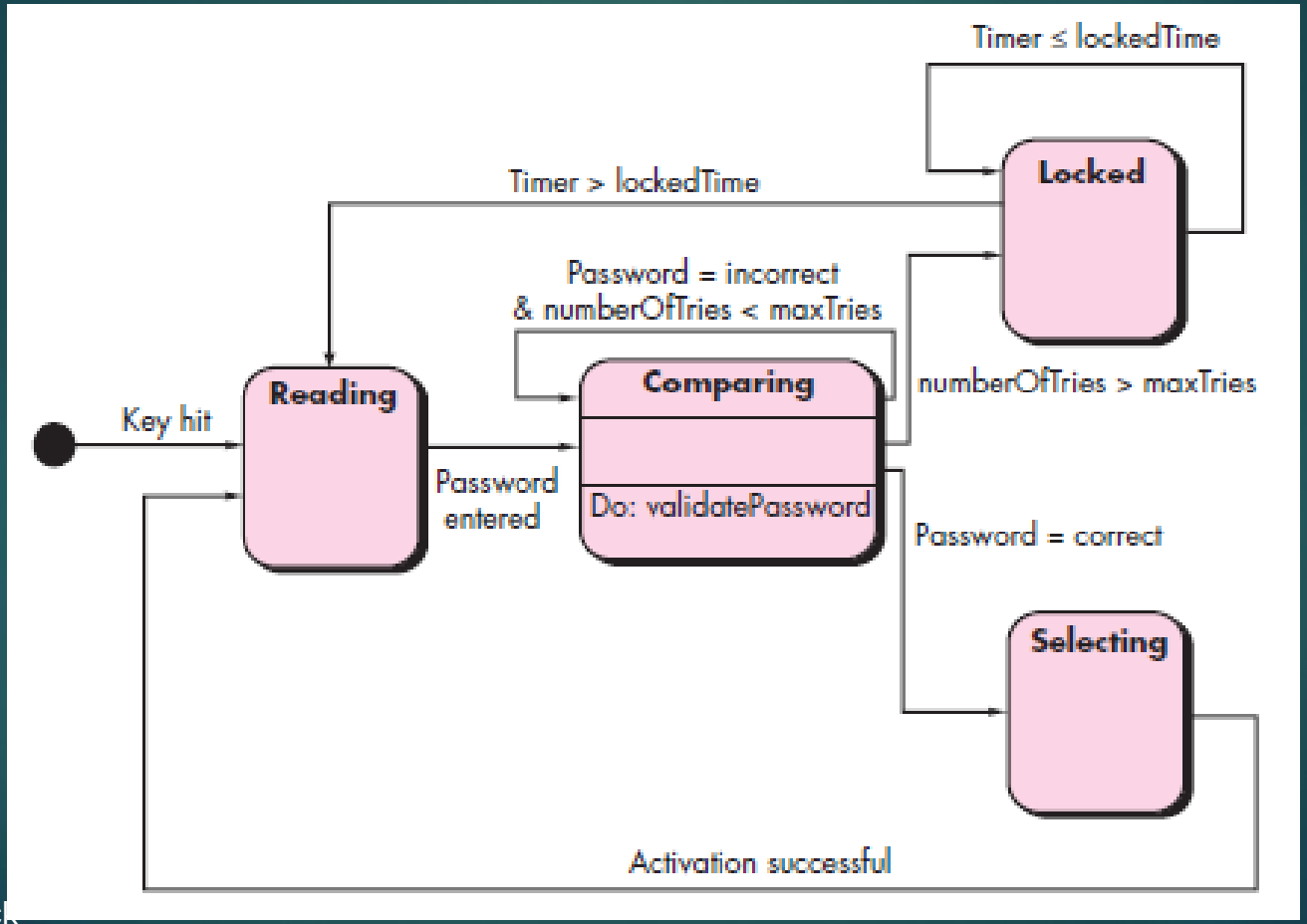
The States of a System

- state—a set of observable circumstances 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

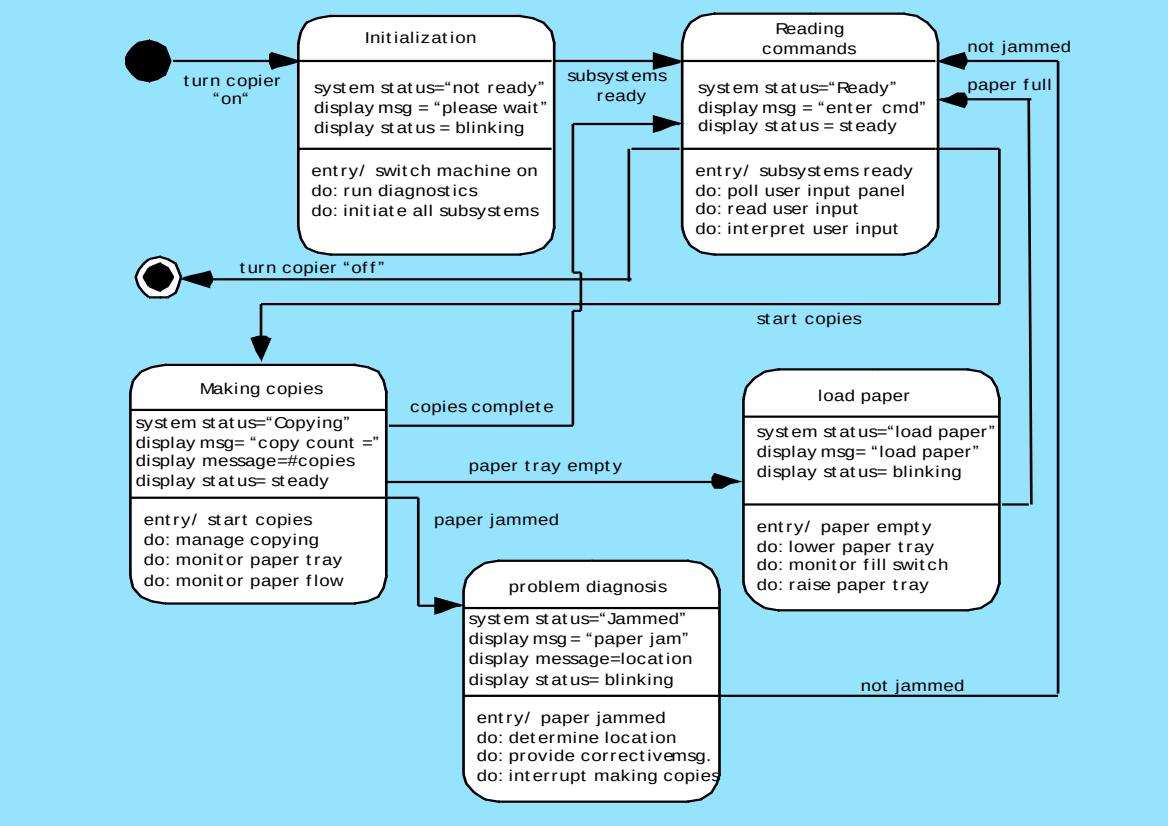
State Diagram



State Diagram for the ControlPanel Class



State Diagram



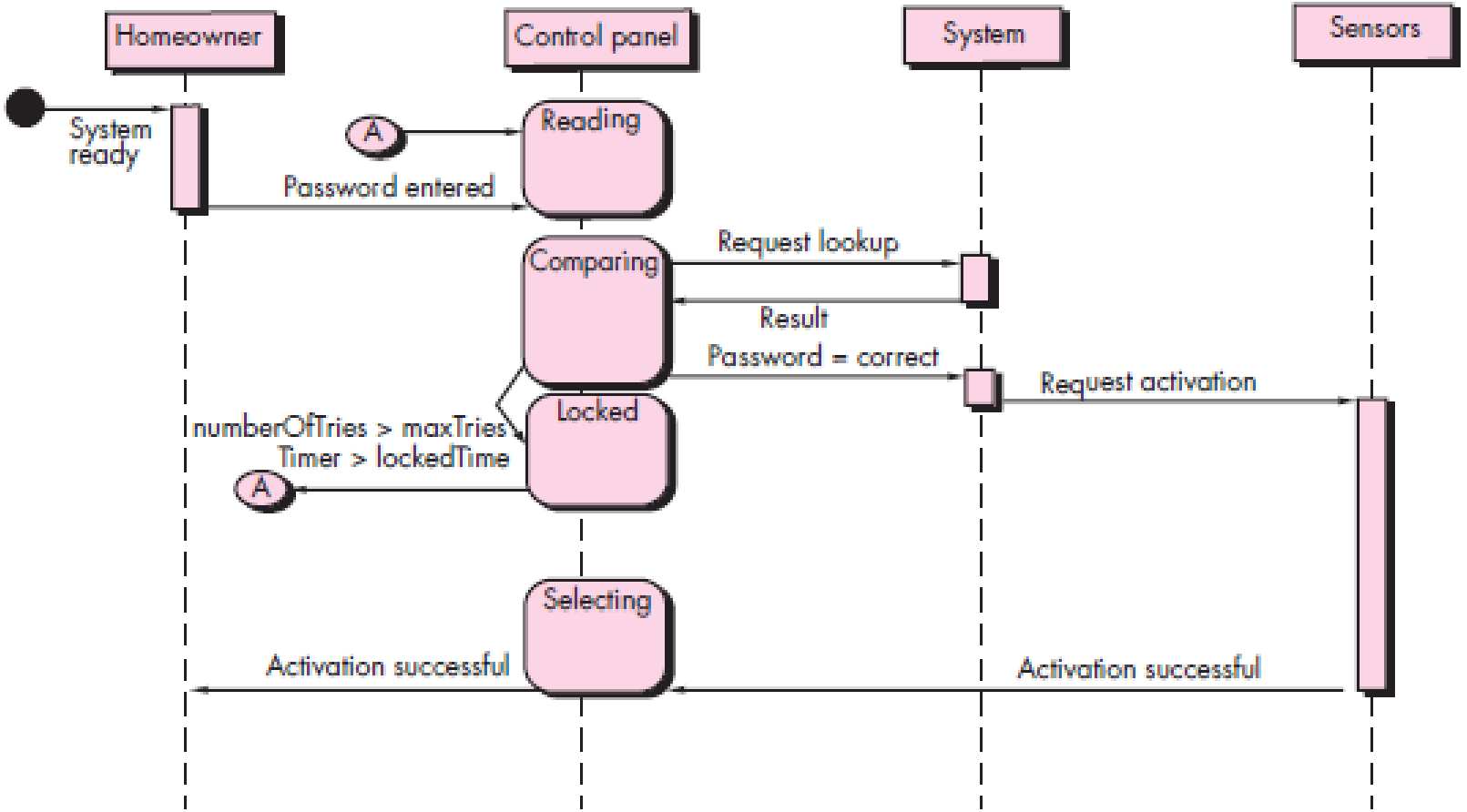
UML state diagram for a office Xerox machine

Behavioral Modeling

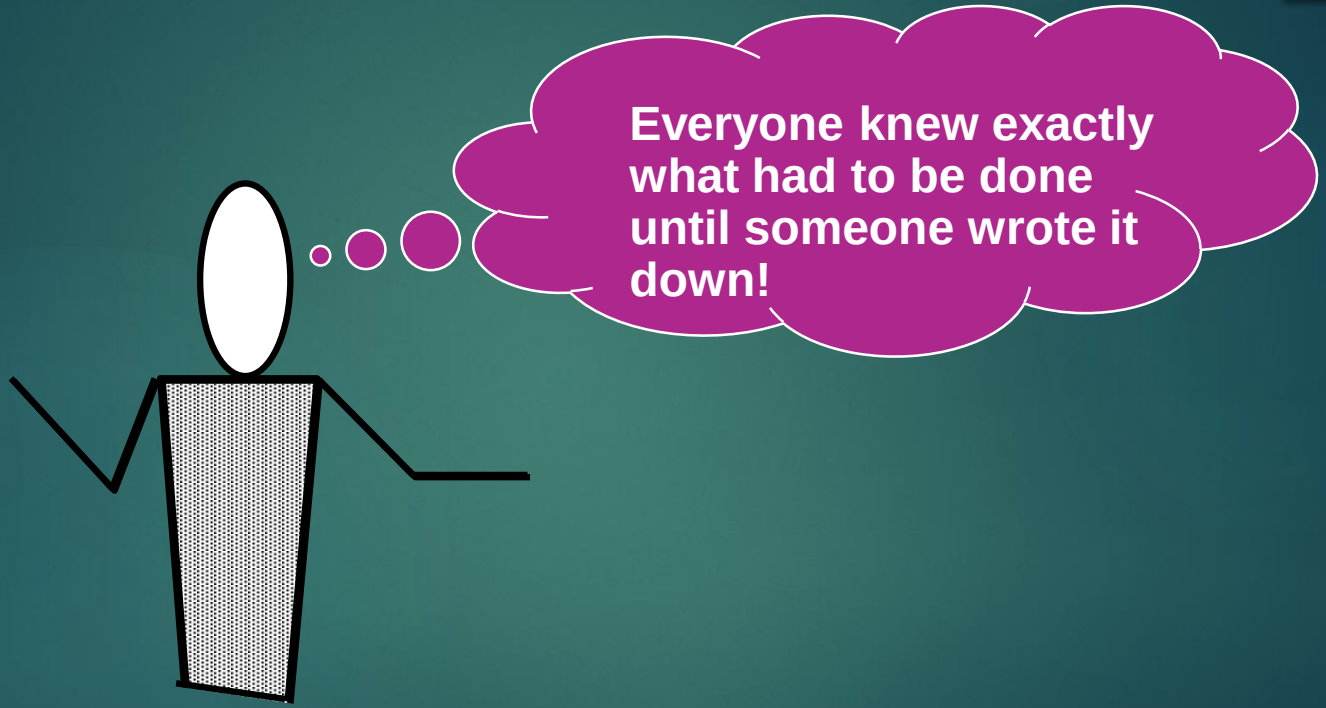
- make a list of the different states of a system (How does the system behave?)
- indicate how the system makes a transition from one state to another (How does the system change state?)
  - indicate event
  - indicate action
- draw a state diagram or a sequence diagram

Sequence Diagram

FIGURE 7.7 Sequence diagram (partial) for the *SafeHome* security function



Writing the Software Specification



Specification Guidelines

- use a layered format that provides increasing detail as the "layers" deepen
- use consistent graphical notation and apply textual terms consistently (stay away from aliases)
- be sure to define all acronyms
- be sure to include a table of contents; ideally, include an index and/or a glossary
- write in a simple, unambiguous style (see "editing suggestions" on the following pages)
- always put yourself in the reader's position, "Would I be able to understand this if I wasn't intimately familiar with the system?"

Specification Guidelines

Be on the lookout for persuasive connectors, ask why?
keys: *certainly, therefore, clearly, obviously, it follows that ...*

Watch out for vague terms
keys: *some, sometimes, often, usually,ordinarily, most, mostly ...*

When lists are given, but not completed, be sure all items are understood
keys: *etc., and so forth, and so on, such as*

Be sure stated ranges don't contain unstated assumptions
e.g., *Valid codes range from 10 to 100.* Integer? Real? Hex?

Beware of vague verbs such as *handled, rejected, processed, ...*

Beware "passive voice" statements
e.g., *The parameters are initialized.* By what?

Beware "dangling" pronouns
e.g., *The I/O module communicated with the data validation module and its contol flag is set.* Whose control flag?

Specification Guidelines

When a term is explicitly defined in one place, try substituting the definition forother occurrences of the term

When a structure is described in words, draw a picture

When a structure is described with a picture, try to redraw the picture to emphasize different elements of the structure

When symbolic equations are used, try expressing their meaning in words

When a calculation is specified, work at least two examples

Look for statements that imply certainty, then ask for proof keys; always, every, all, none, never

Search behind certainty statements—be sure restrictions or limitations are realistic

Estimation

- Estimation of resources, cost, and schedule for a software engineering effort requires
  - experience
  - access to good historical information (metrics
  - the courage to commit to quantitative predictions when qualitative information is all that exists
- Estimation carries inherent risk and this risk leads to uncertainty

Write it Down!

To Understand Scope …

- Understand the customers needs
- understand the business context
- understand the project boundaries
- understand the customer's motivation
- understand the likely paths for change
- understand that …

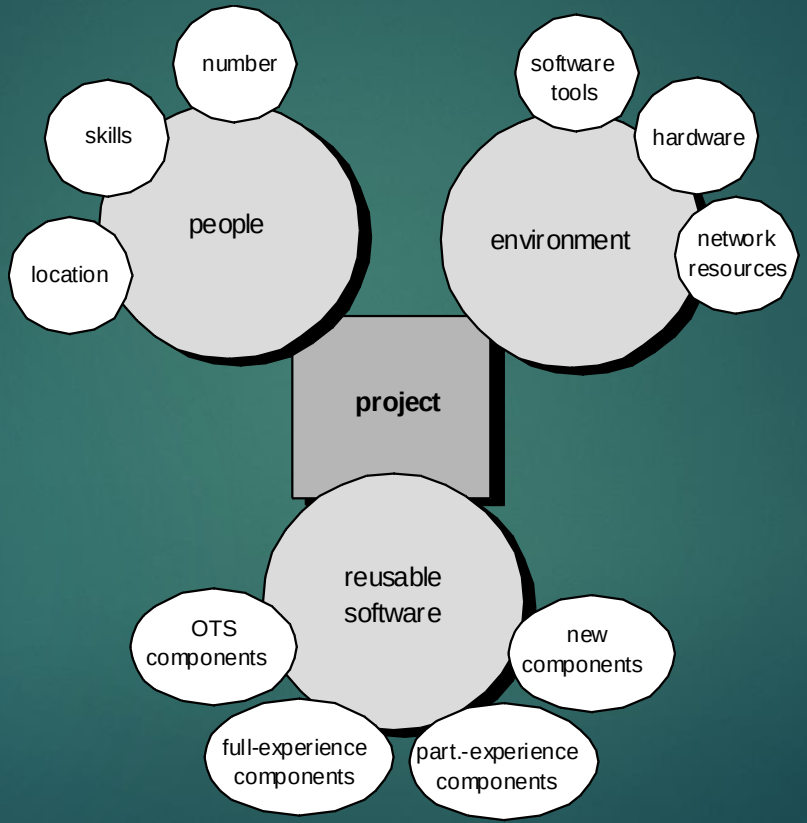
Even when you understand, nothing is guaranteed!

What is Scope?

- *Software scope* describes
  - the functions and features that are to be delivered to end-users
  - the data that are input and output
  - the "content" that is presented to users as a consequence of using the software
  - the performance, constraints, interfaces, and reliability that *bound* the system.

- Scope is defined using one of two techniques:
  - A narrative description of software scope is developed after communication with all stakeholders.
  - A set of use-cases is developed by end-users.

Resources


number
skills
people
location
software tools
hardware
environment
network resources
project
reusable software
OTS components
new components
full-experience components
part.-experience components

Software Project Estimation

- Project scope must be understood
- Elaboration (decomposition) is necessary
- Historical metrics are very helpful
- At least two different techniques should be used
- Uncertainty is inherent in the process
- To achieve reliable cost and effort estimates
  - Delay estimation until late in the project
  - Base estimates on similar projects completed earlier
  - Use simple decomposition techniques to generate the project cost and effort estimates
  - Use one or more empirical models for s/w cost and effort estimation

Estimation Techniques

- Past (similar) project experience
- Conventional estimation techniques
  - task breakdown and effort estimates
  - size (e.g., FP) estimates
- Empirical models
- Automated tools

Estimation Accuracy

- Predicated on …
  - the degree to which the planner has properly estimated the size of the product to be built
  - the ability to translate the size estimate into human effort, calendar time, and dollars (a function of the availability of reliable software metrics from past projects)
  - the degree to which the project plan reflects the abilities of the software team
  - the stability of product requirements and the environment that supports the software engineering effort.

Conventional Methods: LOC/FP Approach

- compute LOC/FP using estimates of information domain values
- use historical data to build estimates for the project

Typical Size-Oriented Metrics

- errors per KLOC (thousand lines of code)
- defects per KLOC
- $ per LOC
- pages of documentation per KLOC
- errors per person-month
- Errors per review hour
- LOC per person-month
- $ per page of documentation

Calculating LOC

There are two major types of SLOC measures:
Physical SLOC (LOC): count of lines in the text of the program's source code including comment lines. Blank lines are also included unless the lines of code in a section consists of more than 25% blank lines. In this case blank lines in excess of 25% are not counted toward lines of code.

Logical SLOC (LLOC). Logical LOC attempts to measure the number of "statements", but their specific definitions are tied to specific computer languages (one simple logical LOC measure for C-like programming languages is the number of statement-terminating semicolons).

Physical LOC	Logical LOC
easier to create tools that measure physical SLOC	Not easy
definitions are easier to explain	
sensitive to logically irrelevant formatting and style conventions	less sensitive to formatting and style conventions

Consider this snippet of C code as an example of the ambiguity encountered when determining SLOC:

```
for (i = 0; i < 100; i += 1) printf("hello"); /* How many lines of code is this? */
```

In this example we have:
•1 Physical Lines of Code (LOC)
•2 Logical Line of Code (LLOC) (for statement and printf statement)
•1 comment line

Depending on the programmer and/or coding standards, the above "line of code" could be written on many separate lines:

```
for (i = 0; i < 100; i += 1)
{
printf("hello");
} /* Now how many lines of code is this? */
```

In this example we have:
•4 Physical Lines of Code (LOC): is placing braces work to be estimated?
•2 Logical Line of Code (LLOC): what about all the work writing non-statement lines?
•1 comment line: tools must account for all code and comments regardless of comment placement.
Even the "logical" and "physical" SLOC values can have a large number of varying definitions. Robert

Size Oriented Metrics

- Not universally accepted as the best measure of software process
- Proponents claim
  - LOC is an arti fact that can be easily counted,
  - many estimation models use LOC or KLOC as key input
  - Large body of literature and data predicated on LOC is available
- Opponents argue
  - LOC are programming language dependent
  - They penalize well designed but shorter programs
  - Cannot accommodate non-procedural languages
  - Use in estimation requires level of detail that is difficult to achieve

Effort Calculation using LOC Approach

Function	Estimated LOC
user interface and control facilities (UICF)	2,300
two-dimensional geometric analysis (2DGA)	5,300
three-dimensional geometric analysis (3DGA)	6,800
database management (DBM)	3,350
computer graphics display facilities (CGDF)	4,950
peripheral control (PC)	2,100
design analysis modules (DAM)	8,400
estimated lines of code	33,200

LOC = 33200

Prod = 620 LOC/pm

Cost = $13 / LOC

Effort = LOC/ Prod = 33200/620 = 54 pm

Total estimated cost = LOC * Cost / LOC

= 33200 * 13

= $431,600

Why Opt for FP?

- Programming language independent
- Uses readily countable characteristics that are determined early in the software process
- Does not “penalize” inventive (short) implementations that use fewer LOC than other more clumsy versions
- Makes it easier to measure the impact of reusable components

Function-Oriented Metrics

- Use a measure of the functionality delivered by the software as a normalization value.
- FP is based on characteristics of the software's information domain and complexity
- For each project
  - errors per FP (thousand lines of code)
  - defects per FP
  - $ per FP
  - pages of documentation per FP
  - FP per person-month

Function-Oriented Metrics

- Like LOC, FP is also controversial
- Proponents claim
  - FP is language independent, ideal for applications using conventional and non-procedural languages
  - It is based on data that are more likely to be known early in the evolution of a project
  - So more attractive than LOC
- Opponents argue
  - Computation is subjective rather than objective, requires expertise
  - Counts of information domain can be difficult to collect
  - FP has no direct physical meaning – it's just a number

Comparing LOC and FP

Programming Language	LOC per Function point avg.	median	low	high
Ada	154	-	104	205
Assembler	337	315	91	694
C	162	109	33	704
C++	66	53	29	178
COBOL	77	77	14	400
Java	63	53	77	-
JavaScript	58	63	42	75
Perl	60	-	-	-
PL/1	78	67	22	263
Powerbuilder	32	31	11	105
SAS	40	41	33	49
Smalltalk	26	19	10	55
SQL	40	37	7	110
Visual Basic	47	42	16	158

Representative values developed by QSM

Function-Point Metrics

- The *function point metric* (FP), first proposed by Albrecht [ALB79], can be used effectively as a means for measuring the functionality delivered by a system.
- Function points are derived using an empirical relationship based on countable (direct) measures of software's information domain and assessments of software complexity
- Information domain values are defined in the following manner:
  - number of external inputs (EIs)
  - number of external outputs (EOs)
  - number of external inquiries (EQs)
  - number of internal logical files (ILFs)
  - Number of external interface files (EIFs)

Function Point Estimation

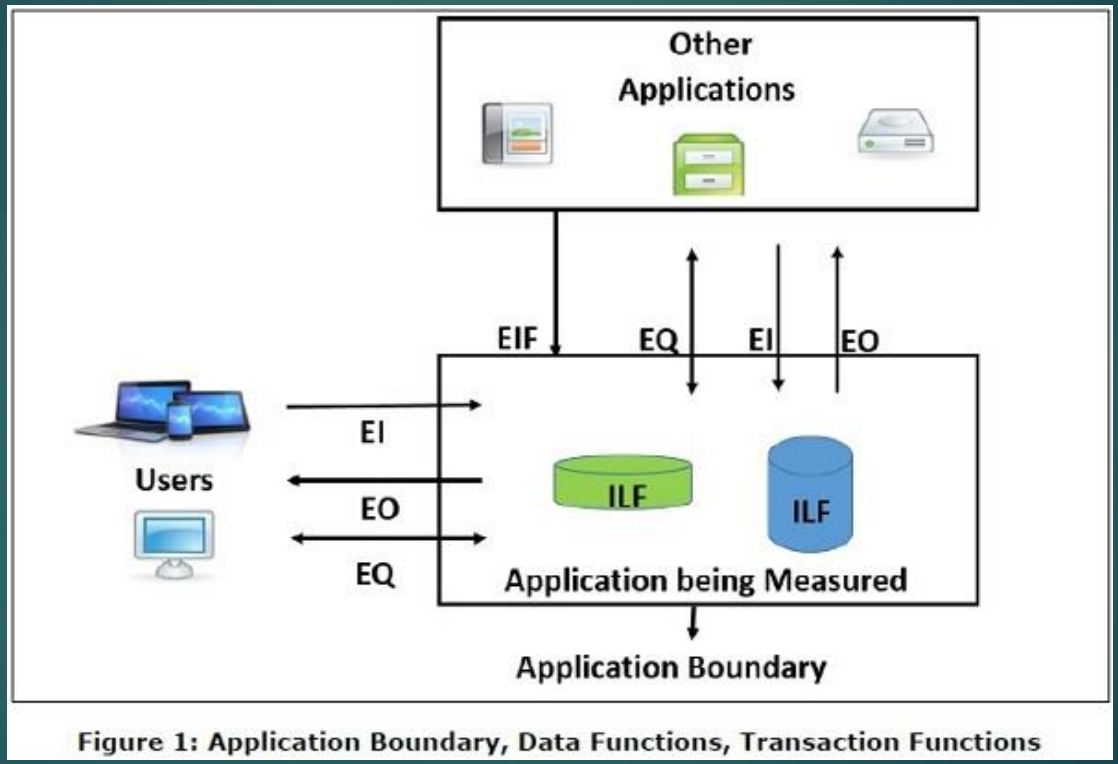


Figure 1: Application Boundary, Data Functions, Transaction Functions

- **External Inputs**
  - External Input (EI) is a transaction function in which Data goes "into" the application from outside the boundary to inside. This data is coming external to the application.
  - Data may come from a data input screen or another application.
  - An EI is how an application gets information.
  - Data can be either control information or business information.
  - Data may be used to maintain one or more Internal Logical Files.
  - If the data is control information, it does not have to update an Internal Logical File.
- **External Outputs**
  - External Output (EO) is a transaction function in which data comes "out" of the system. Additionally, an EO may update an ILF. The data creates reports or output files sent to other applications.
- **External Inquiries**
  - An online input that results in the generation of some immediate s/w response in form of an online output
  - External Inquiry (EQ) is a transaction function with both input and output components that result in data retrieval.

- **Internal Logical Files**
  - Internal Logical File (ILF) is a user identifiable group of logically related data or control information that resides entirely within the application boundary. The primary intent of an ILF is to hold data maintained through one or more elementary processes of the application being counted. An ILF has the inherent meaning that it is internally maintained, it has some logical structure and it is stored in a file. (Refer Figure 1)
- **External Interface Files**
  - External Interface File (EIF) is a user identifiable group of logically related data or control information that is used by the application for reference purposes only. The data resides entirely outside the application boundary and is maintained in an ILF by another application. An EIF has the inherent meaning that it is externally maintained, an interface has to be developed to get the data from the file.

Function Points

Information Domain Value	Count	Weighting factor simple	average	complex	
External Inputs (EIs)		3	4	6	=
External Outputs (EOs)		4	5	7	=
External Inquiries (EQs)		3	4	6	=
Internal Logical Files (ILFs)		7	10	15	=
External Interface Files (EIFs)		5	7	10	=
Count total	→				

The estimated number of FP is derived:

$$FP_{estimated} = \text{count-total} \times [0.65 + 0.01 \times \Sigma (F_i)]$$

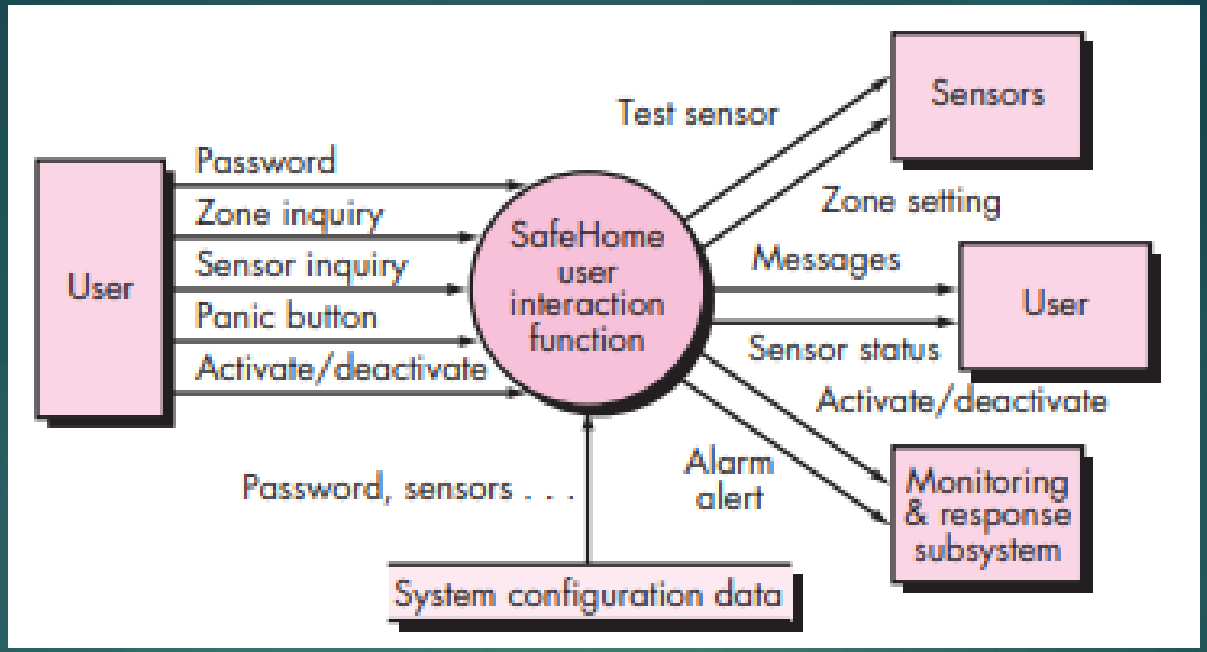
$$FP_{estimated} =$$

Where Fi (I = 1 to 14) are value adjustment factors (VAF)

Each of these questions is answered using a scale ranging from 0 (not applicable) to 5 (absolutely essential)

The Fi (i = 1 to 14) are value adjustment factors (VAF) based on responses to the following questions:

1. Does the system require reliable backup and recovery?
2. Are specialized data communications required to transfer information to or from the application?
3. Are there distributed processing functions?
4. Is performance critical?
5. Will the system run in an existing, heavily utilized operational environment?
6. Does the system require online data entry?
7. Does the online data entry require the input transaction to be built over multiple screens or operations?
8. Are the ILFs updated online?
9. Are the inputs, outputs, files, or inquiries complex?
10. Is the internal processing complex?
11. Is the code designed to be reusable?
12. Are conversion and installation included in the design?
13. Is the system designed for multiple installations in different organizations?
14. Is the application designed to facilitate change and ease of use by the user



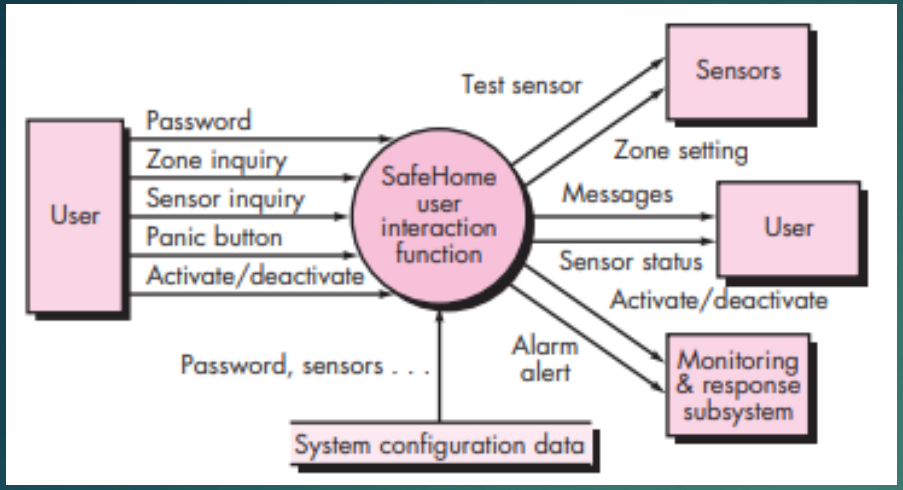
External inputs — password, panic button, and activate/deactivate

External inquiries — zone inquiry and sensor inquiry.

Internal Logical Files - system configuration file.

External outputs - messages and sensor status

External Interface Files - test sensor, zone setting, activate/deactivate, and alarm alert



Information Domain Value	Count		Weighting factor: Simple	Average	Complex		
External Inputs (EIs)	3	×	3	4	6	=	9
External Outputs (EOs)	2	×	4	5	7	=	8
External Inquiries (EQs)	2	×	3	4	6	=	6
Internal Logical Files (ILFs)	1	×	7	10	15	=	7
External Interface Files (EIFs)	4	×	5	7	10	=	20
Count total							50

FP = 50 x [0.65 x (0.01 x 46)] = 56

Example: FP Approach

Information Domain Value	opt.	likely	pess.	est. count	weight	FP-count
number of inputs	20	24	30	24	4	97
number of outputs	12	15	22	16	5	78
number of inquiries	16	22	28	22	5	88
number of files	4	4	5	4	10	42
number of external interfaces	2	2	3	2	7	15
count-total						321

FP = 375

Prod = 6.5 FP/pm

Cost = $1230 / FP

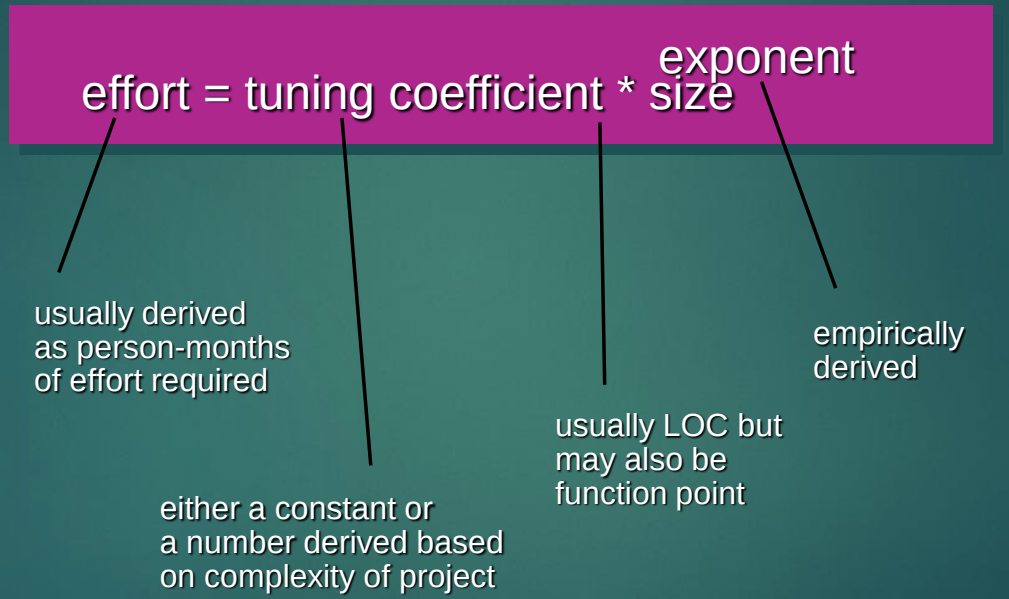
Effort = FP/ Prod = 375/6.5 = 58 pm

Total estimated cost = FP * Cost / FP =

375 * 1230= $461,250

Empirical Estimation Models

General form:



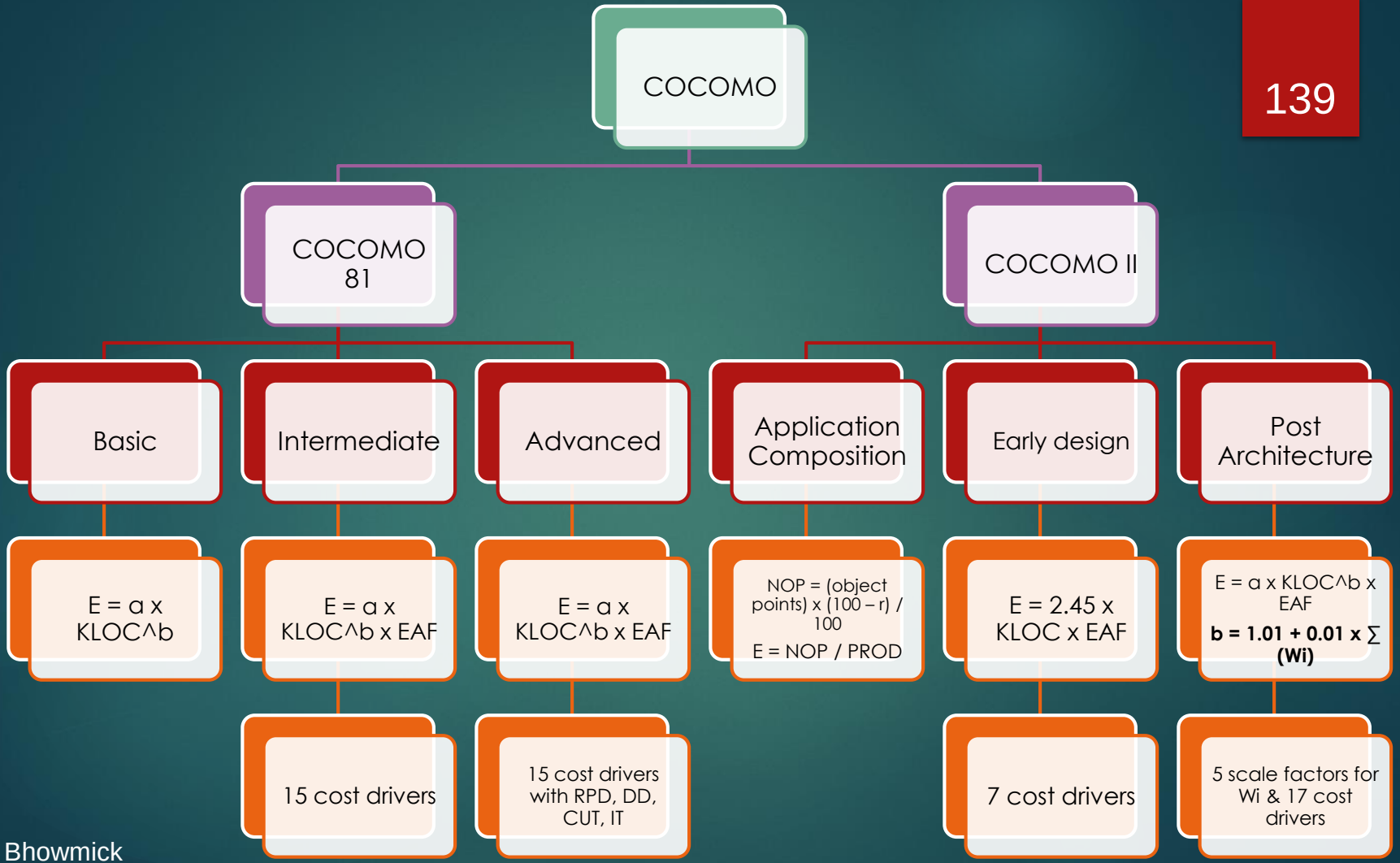
$E = A + B \times (e_v)^C$

LOC-oriented

- $E = 5.2 \times (KLOC)^{0.91}$ Walston-Felix model
- $E = 5.5 + 0.73 \times (KLOC)^{1.16}$ Bailey-Basili model
- $E = 3.2 \times (KLOC)^{1.05}$ Boehm simple model
- $E = 5.288 \times (KLOC)^{1.047}$ Doty model for KLOC > 9

FP oriented

- $E = -91.4 + 0.355\ FP$ Albrecht and Gaffney model
- $E = -37 + 0.96\ FP$ Kemerer model
- $E = -12.88 + 0.405\ FP$ small project regression model


COCOMO
COCOMO 81
COCOMO II
Basic
Intermediate
Advanced
Application Composition
Early design
Post Architecture
E = a x KLOC^b
E = a x KLOC^b x EAF
E = a x KLOC^b x EAF
NOP = (object points) x (100 – r) / 100
E = NOP / PROD
E = 2.45 x KLOC x EAF
E = a x KLOC^b x EAF
b = 1.01 + 0.01 x ∑ (Wi)
15 cost drivers
15 cost drivers with RPD, DD, CUT, IT
7 cost drivers
5 scale factors for Wi & 17 cost drivers

COCOMO-81

- Three modes:
  - Organic
  - Embedded
  - Semi-detached
- Three models:
- Basic - It is applied early in a project. The Basic COCOMO model computes effort as function of program size.
- Intermediate - It is applied after requirements are specified. The Intermediate COCOMO model computes effort as a function of program size and a set of cost drivers.
- Advanced - It is applied after the design is complete. The Advanced COCOMO model computes effort as a function of program size and a set of cost drivers weighted according to each phase of the software lifecycle.

COCOMO-II

COCOMO II is actually a hierarchy of estimation models that address the following areas:

- *Application composition model.* Used during the early stages of software engineering, when prototyping of user interfaces, consideration of software and system interaction, assessment of performance, and evaluation of technology maturity are paramount.
- *Early design stage model.* Used once requirements have been stabilized and basic software architecture has been established.
- *Post-architecture-stage model.* Used during the construction of the software.

COCOMO-II Application composition model

- COCOMO II requires Sizing information.
- 3 different sizing options are available
  - Object points
  - Function points
  - Lines of source code
- COCOMO II uses object points – indirect measure of number of screens, reports and components required
- Each is classified into 3 complexity levels
  - Simple, Medium and Difficult
  - Complexity – function of number and source of the client and server data tables required to generate

COCOMO-II Application composition model

Object point complexity levels for screens and reports

Number of views contained	Number and source of data tables		
	Total <4	Total <8	Total 8+
<3	simple	simple	Medium
3-7	simple	medium	difficult
8+	medium	difficult	difficult

Object Type	Complexity Weight		
	Simple	Medium	Difficult
Screen	1	2	3
Report	2	5	8
3GL component			10

Object points = Σ (object type * complexity)

New Object Points – component based development or reuse is to be applied.

NOP = (object points) x [(100 - %reuse)/100]

Productivity rate for object points for different levels of developer experience and development environment maturity

Developer's experience/capability	Very Low	Low	Nominal	High	Very High
Environment maturity/capability	Very Low	Low	Nominal	High	Very High
PROD	4	7	13	25	50

Prod = ---------- NOP/person-month

Estimated Effort = NOP/PROD

Early Design model

- The Early Design model is used to evaluate alternative software/system architectures and concepts of operation. An unadjusted function point count (UFC) is used for sizing. This value is converted to LOC using following tables

Language	Level	Min	Mode	Max
Machine language	0.10	-	640	-
Assembly	1.00	237	320	416
C	2.50	60	128	170
RPGII	5.50	40	58	85
C++	6.00	40	55	140
Visual C++	9.50	-	34	-
PowerBuilder	20.00	-	16	-
Excel	57.00	-	5.5	-

Information Domain Value	Count	Weighting factor				
			simple	average	complex	
External Inputs (EIs)		3	3	4	6	=
External Outputs (EOs)		3	4	5	7	=
External Inquiries (EQs)		3	3	4	6	=
Internal Logical Files (ILFs)		3	7	10	15	=
External Interface Files (EIFs)		3	5	7	10	=
Count total						

- The Early Design model equation is:

  E = a x KLOC x EAF

- where *a* is a constant, provisionally set to 2.45.
- The effort adjustment factor (EAF) is calculated as in the original COCOMO model using the 7 cost drivers shown in the table.

Cost Driver	Description	Counterpart Combined Post-Architecture Cost Driver
RCPX	Product reliability and complexity	RELY, DATA, CPLX, DOCU
RUSE	Required reuse	RUSE
PDIF	Platform difficulty	TIME, STOR, PVOL
PERS	Personnel capability	ACAP, PCAP, PCON
PREX	Personnel experience	AEXP, PEXP, LTEX
FCIL	Facilities	TOOL, SITE
SCED	Schedule	SCED

Post Architecture Model

- is used during the actual development and maintenance of a product. Function points or LOC can be used for sizing, with modifiers for reuse and software breakage. The model includes a set of 17 cost drivers and a set of 5 factors determining the projects scaling component. The 5 factors replace the development modes (organic, semidetached, and embedded) of the original COCOMO model.
- The Post-Architecture model equation is:

  E = a x KLOC^b x EAF

- where *a* is set to 2.55 and *b* is calculated as:

  b = 1.01 + 0.01 x ∑ (Wi)

- where *W* is the set of 5 scale factors shown in table below

W(i)	Very Low	Low	Nominal	High	Very High	Extra High
Precedence	4.05	3.24	2.42	1.62	0.81	0.00
Development/ Flexibility	6.07	4.86	3.64	2.43	1.21	0.00
Architecture / Risk Resolution	4.22	3.38	2.53	1.69	0.84	0.00
Team Cohesion	4.94	3.95	2.97	1.98	0.99	0.00
Process Maturity	4.54	3.64	2.73	1.82	0.91	0.00

The EAF is calculated using the 17 cost drivers shown in table.

Cost Driver	Description	Rating					
		Very Low	Low	Nominal	High	Very High	Extra High
Product							
RELY	Required software reliability	0.75	0.88	1.00	1.15	1.39	-
DATA	Database size	-	0.93	1.00	1.09	1.19	-
CPLX	Product complexity	0.70	0.88	1.00	1.15	1.30	1.66
RUSE	Required reusability		0.91	1.00	1.14	1.29	1.49
DOCU	Documentation		0.95	1.00	1.06	1.13	
Platform							
TIME	Execution time constraint	-	-	1.00	1.11	1.31	1.67
STOR	Main storage constraint	-	-	1.00	1.06	1.21	1.57
PVOL	Platform volatility	-	0.87	1.00	1.15	1.30	-
Personnel							
ACAP	Analyst capability	1.50	1.22	1.00	0.83	0.67	-
PCAP	Programmer capability	1.37	1.16	1.00	0.87	0.74	-
PCON	Personnel continuity	1.24	1.10	1.00	0.92	0.84	-
AEXP	Applications experience	1.22	1.10	1.00	0.89	0.81	-
PEXP	Platform experience	1.25	1.12	1.00	0.88	0.81	-
LTEX	Language and tool experience	1.22	1.10	1.00	0.91	0.84	
Project							
TOOL	Software Tools	1.24	1.12	1.00	0.86	0.72	-
SITE	Multisite development	1.25	1.10	1.00	0.92	0.84	0.78
SCED	Development Schedule	1.29	1.10	1.00	1.00	1.00	-

Estimation for OO Projects

1. Develop estimates using effort decomposition, FP analysis, and any other method that is applicable for conventional applications.
2. Using object-oriented analysis modeling, develop use-cases and determine a count.
3. From the analysis model, determine the number of key classes.
4. Categorize the type of interface for the application and develop a multiplier for support classes:

Interface type	**Multiplier**
No GUI	2.0
Text-based user interface	2.25
GUI	2.5
Complex GUI	3.0

Estimation for OO Projects

5. Multiply the number of key classes (step 3) by the multiplier to obtain an estimate for the number of support classes.

   No. of support classes = No. of key classes * multiplier

6. Multiply the total number of classes (key + support) by the average number of work - units per class. Lorenz and Kidd suggest 15 to 20 person-days per class.

   Assuming 18 person-days per class;

   E = (no. of key + no. of support) * 18

7. Cross check the class-based estimate by multiplying the average number of work-units per use-case

Estimation for Agile Projects

- Each user scenario (a mini-use-case) is considered separately for estimation purposes.
- The scenario is decomposed into the set of software engineering tasks that will be required to develop it.
- Each task is estimated separately. Note: estimation can be based on historical data, an empirical model, or "experience."
  - Alternatively, the 'volume' of the scenario can be estimated in LOC, FP or some other volume-oriented measure (e.g., use-case count).
- Estimates for each task are summed to create an estimate for the scenario.
  - Alternatively, the volume estimate for the scenario is translated into effort using historical data.
- The effort estimates for all scenarios that are to be implemented for a given software increment are summed to develop the effort estimate for the increment.

Estimation for Web Engineering Projects

- Information domain values for adapting function point for WebApp
  - Inputs – input screen or forms
  - Outputs – static or dynamic web page
  - Tables – logical table in database + XML to store data
  - Interfaces – out-of-the-system boundaries logical files (e.g unique record formats)
  - Queries – externally published or use a message oriented interface.
- FP are a reasonable indicator of volume for a WebApp
- Alternatively, one can measure
  - Page count, function count
  - Page complexity, linking complexity, graphic complexity
  - Media duration
  - Code length, reused code length