Introduction to UML

Modeling

- Describing a system at a high level of abstraction
  - A model of the system (drawings)
  - Used for requirements specifications

- It is necessary to model software systems?

Why do we model?

- Provide structure for problem solving
- Experiment to explore multiple solutions
- Furnish abstractions to manage complexity
- Reduce time-to-market for business problem solutions
- Decrease development costs
- Manage the risk of mistakes

What is UML?

- UML stands for “**Unified Modeling Language**”
- It is an industry-standard graphical language for specifying, visualizing, constructing, and documenting the artifacts of software systems
- The UML uses mostly graphical notations to express the OO analysis and design of software projects.
- Simplifies the complex process of software design

What is UML?

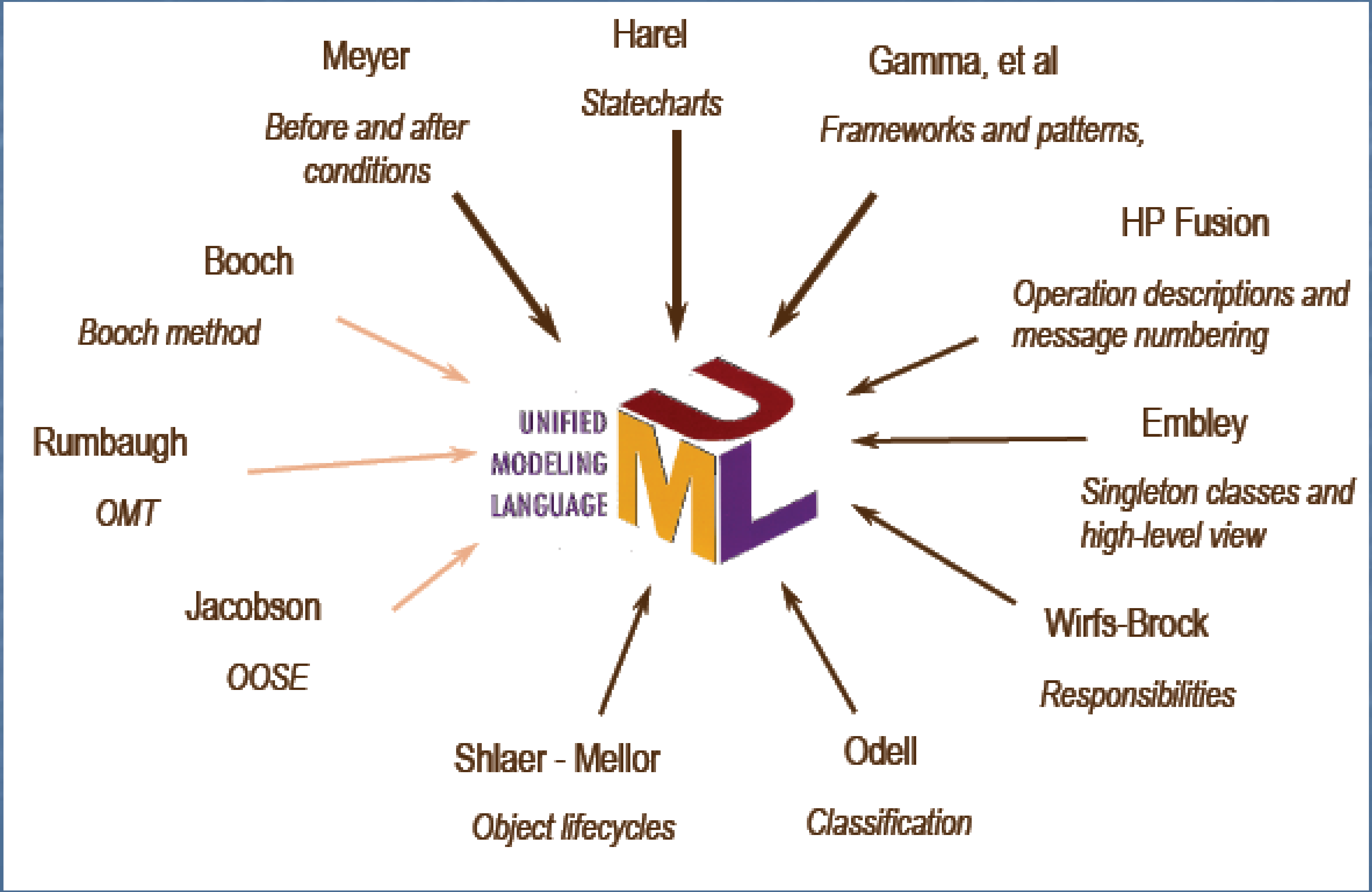
- The UML is a graphical language for
  - specifying
  - visualizing
  - constructing
  - documenting

  the artifacts of software systems
- Added to the list of OMG (object management group) adopted technologies in November 1997 as UML 1.1 (Current Version is 2.5 -- http://www.uml.org/)
- To end the OO method wars
  - Lack of standardization
- Has become the **formal and de facto industry standard (accepted by all)**

Why UML for Modeling

- Use graphical notation to communicate more clearly than natural language (inaccurate) and code (too detailed).
- Help acquire an overall view of a system.
- **UML is *not* dependent on any programming language or technology.**

UML is … unified



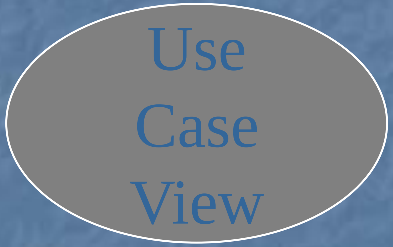
Parts of UML

- Views - shows different aspects of the system that are modeled, links the modeling language to the method/process chosen for development

- Diagrams - graphs that describe the contents in a view

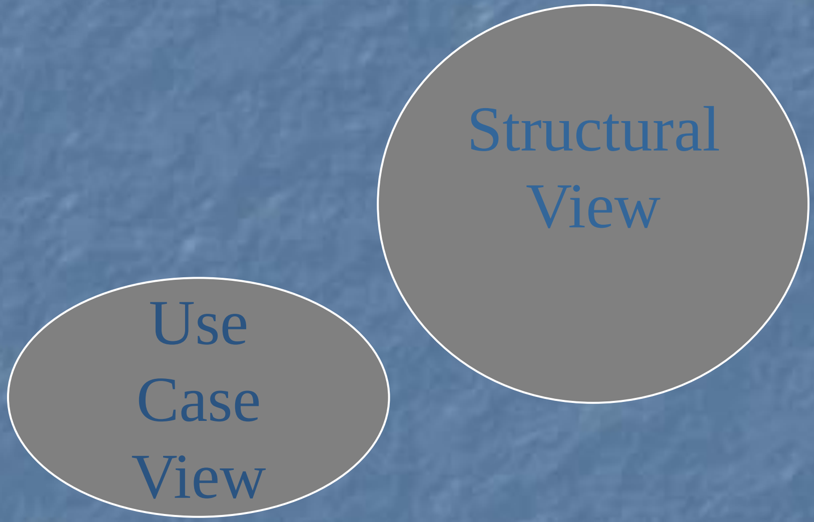
- Model elements - concepts used in a diagram

Views in UML



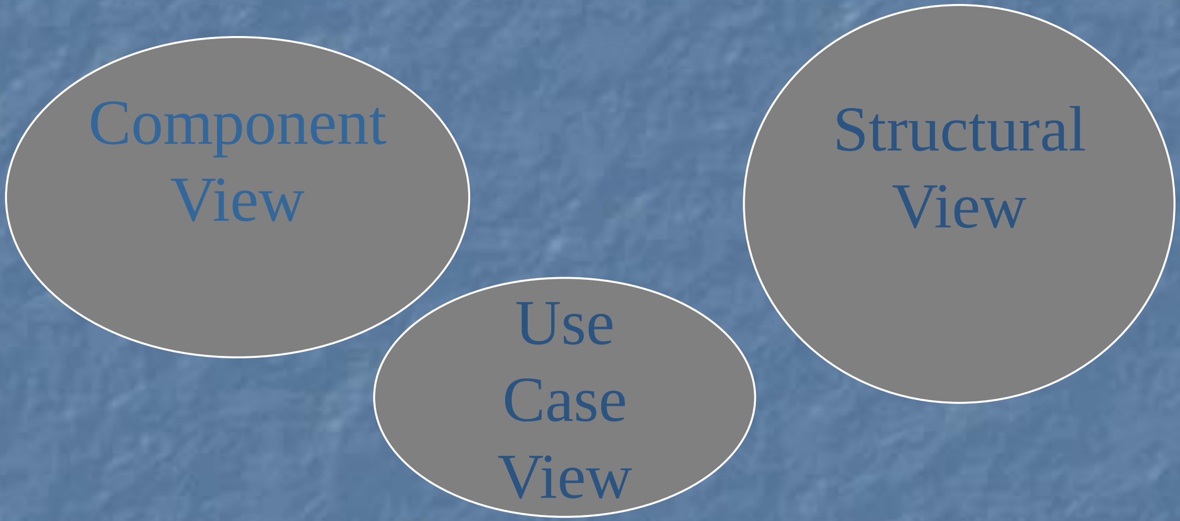
- Use-case view : *A view showing the functionality of the system as perceived by the external actors*

Views in UML



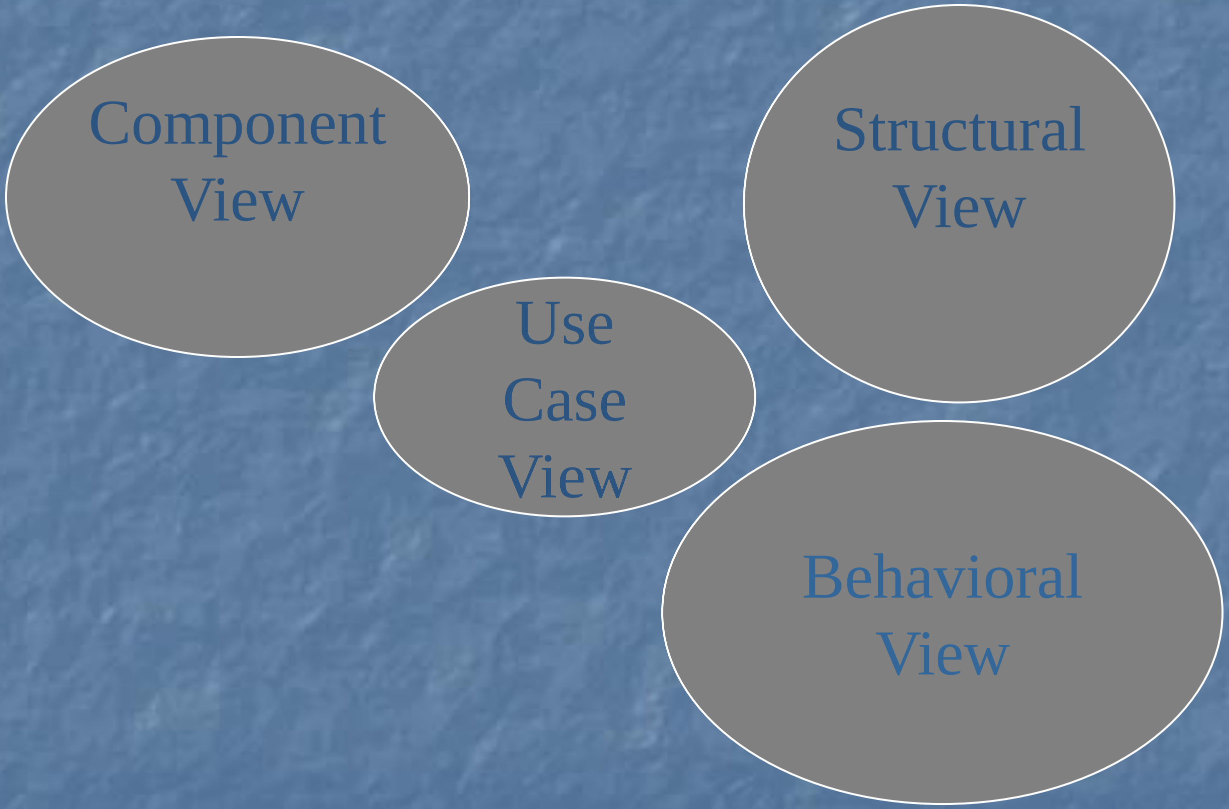
- Structural view: *A view showing how the basic structure of the system is designed.*

Views in UML



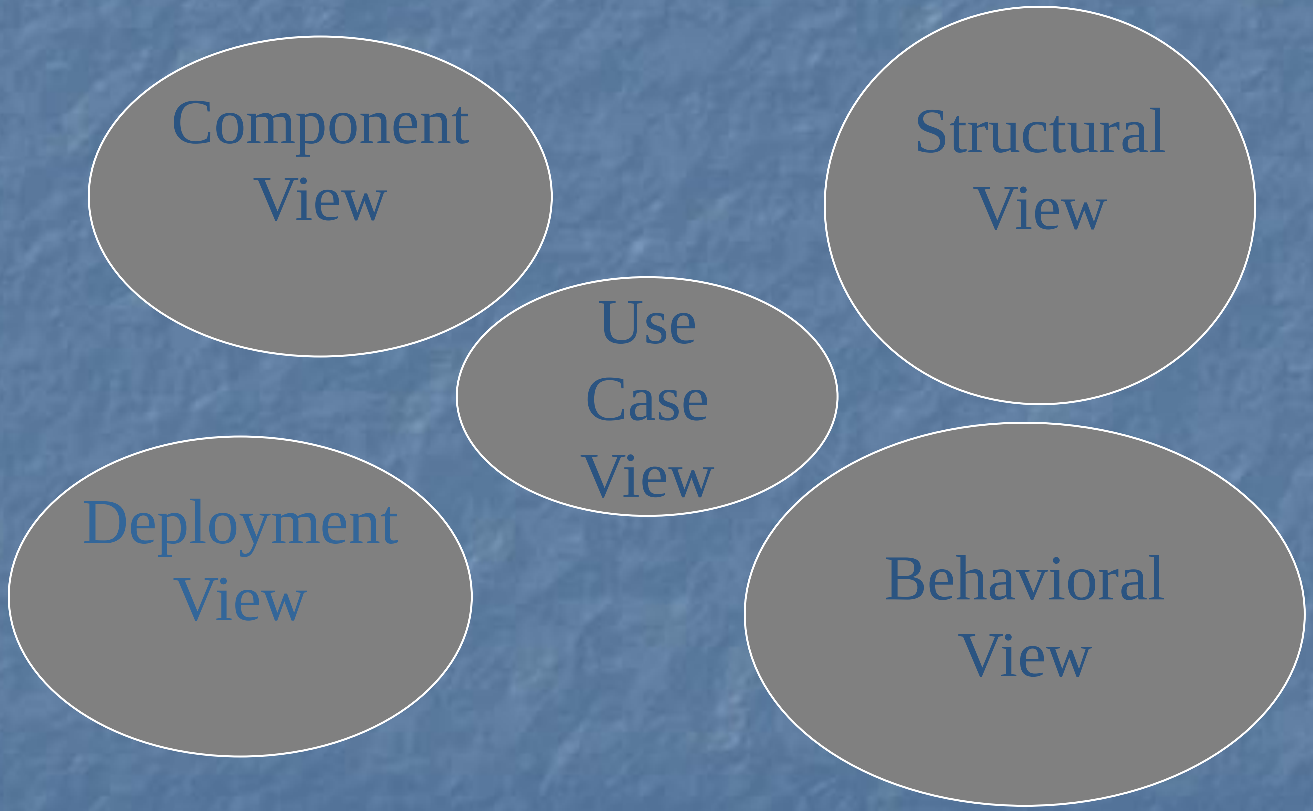
- Component view: *A view showing the organization of the core components*

Views in UML



- Behavioral view: *A view showing the overall behavior of the system*

Views in UML

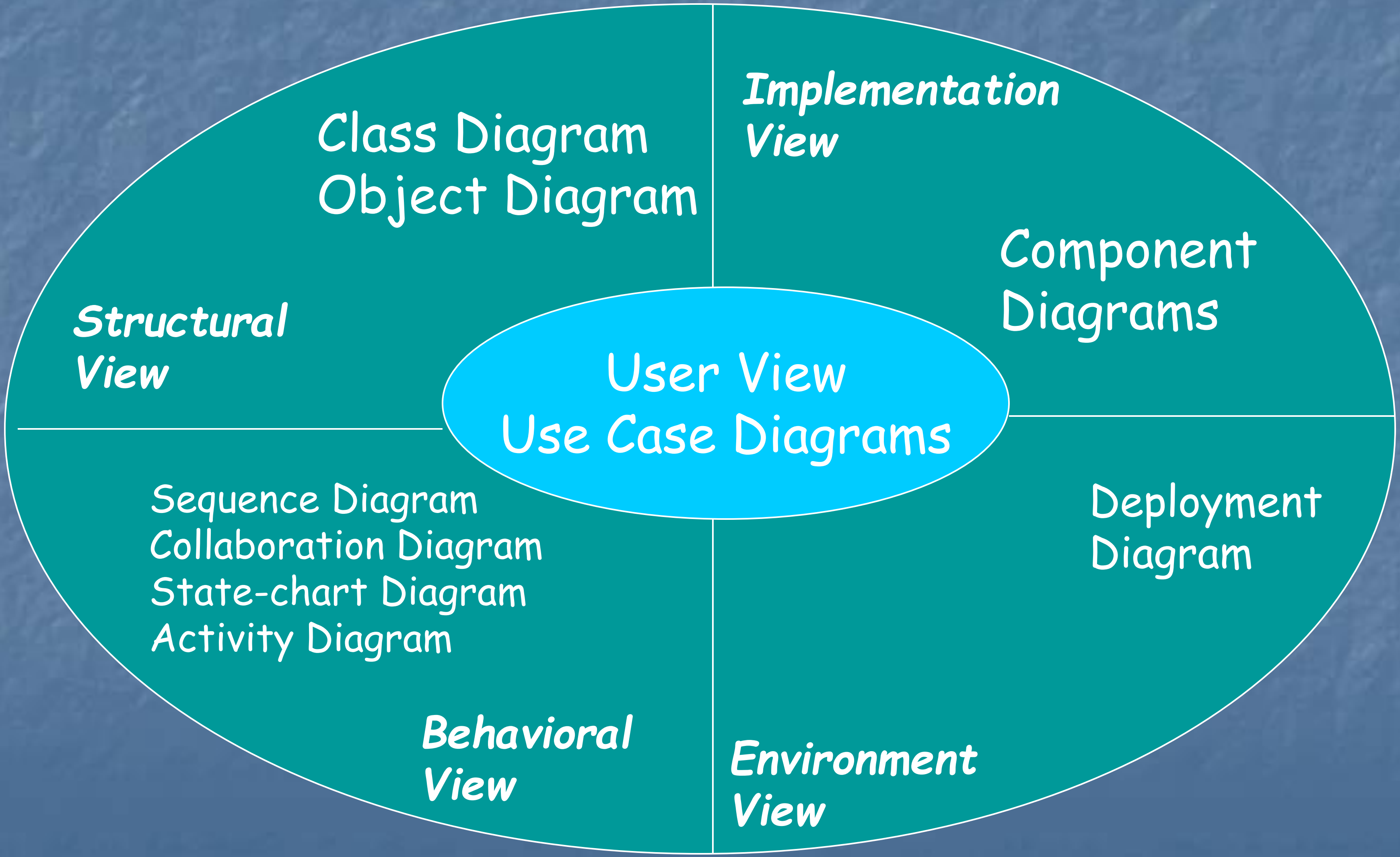


- Deployment view: *A view showing the deployment of the system in terms of the physical architecture*

Introduction to UML…

- Model elements (artifacts):
  - Class
  - Object
  - State
  - Use case
  - Interface
  - Association
  - Link
  - Package ….

Model and Views of UML



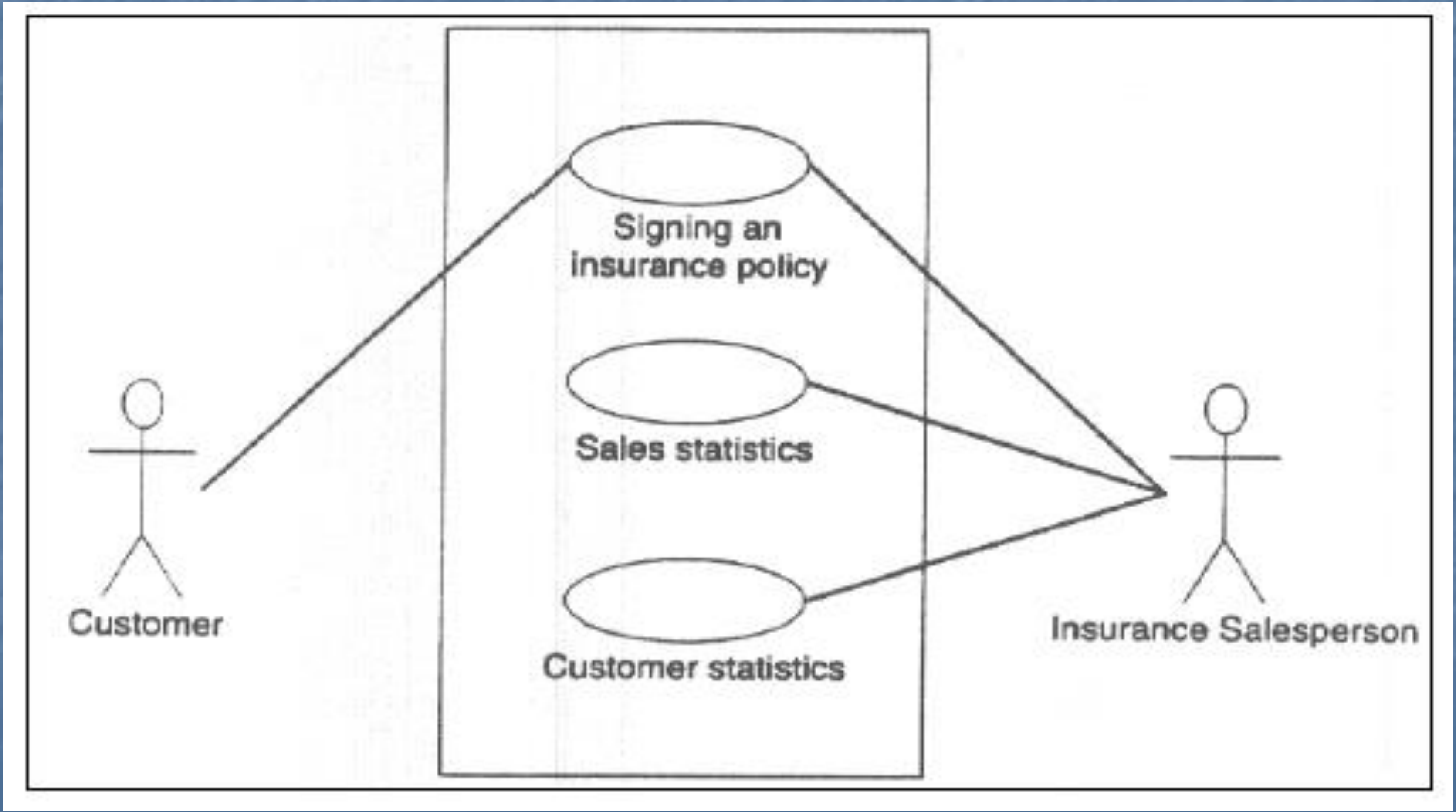
UML diagrams

- Use Case diagram: External interaction with actors
- Class/Object Diagram : captures static structural aspects, objects and relationships.
- State-chart Diagram: Dynamic state behavior
- Sequence diagram: models object interaction over time
- Collaboration diagram: models component interaction and structural dependencies

More UML diagrams

- Activity diagram : models object workflow of activities
- Deployment diagram : models physical architecture
- Component diagram : models software architecture

Use-Case Diagram (e.g. insurance system)



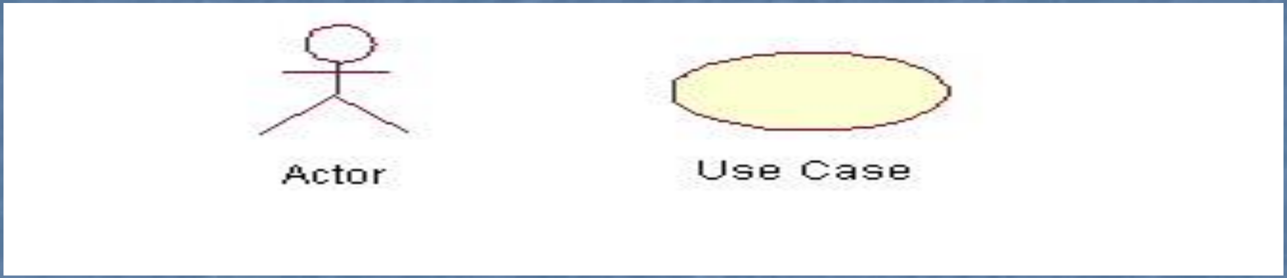
Use Case Diagram

- Used for describing a set of user **scenarios**
- Mainly used for capturing user requirements
- Work like a **contract** between end user and software developers

Use Case Diagram (core components)

Actors: A role that a user plays with respect to the system, including human users and other systems. e.g. physical objects (e.g. robot); an external system that needs some information from the current system.

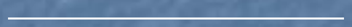
Use case: A set of scenarios that describing an interaction between a user and a system, including alternatives.



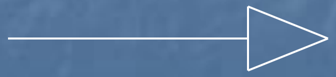
System boundary: rectangle diagram representing the boundary between the actors and the system.

Use Case Diagram (core relationship)

Association: communication between an actor and a use case; Represented by a solid line.



Generalization: relationship between one general use case and a special use case (used for defining special alternatives)
Represented by a line with a triangular arrow head toward the parent use case.



Use Case Diagram (core relationship)

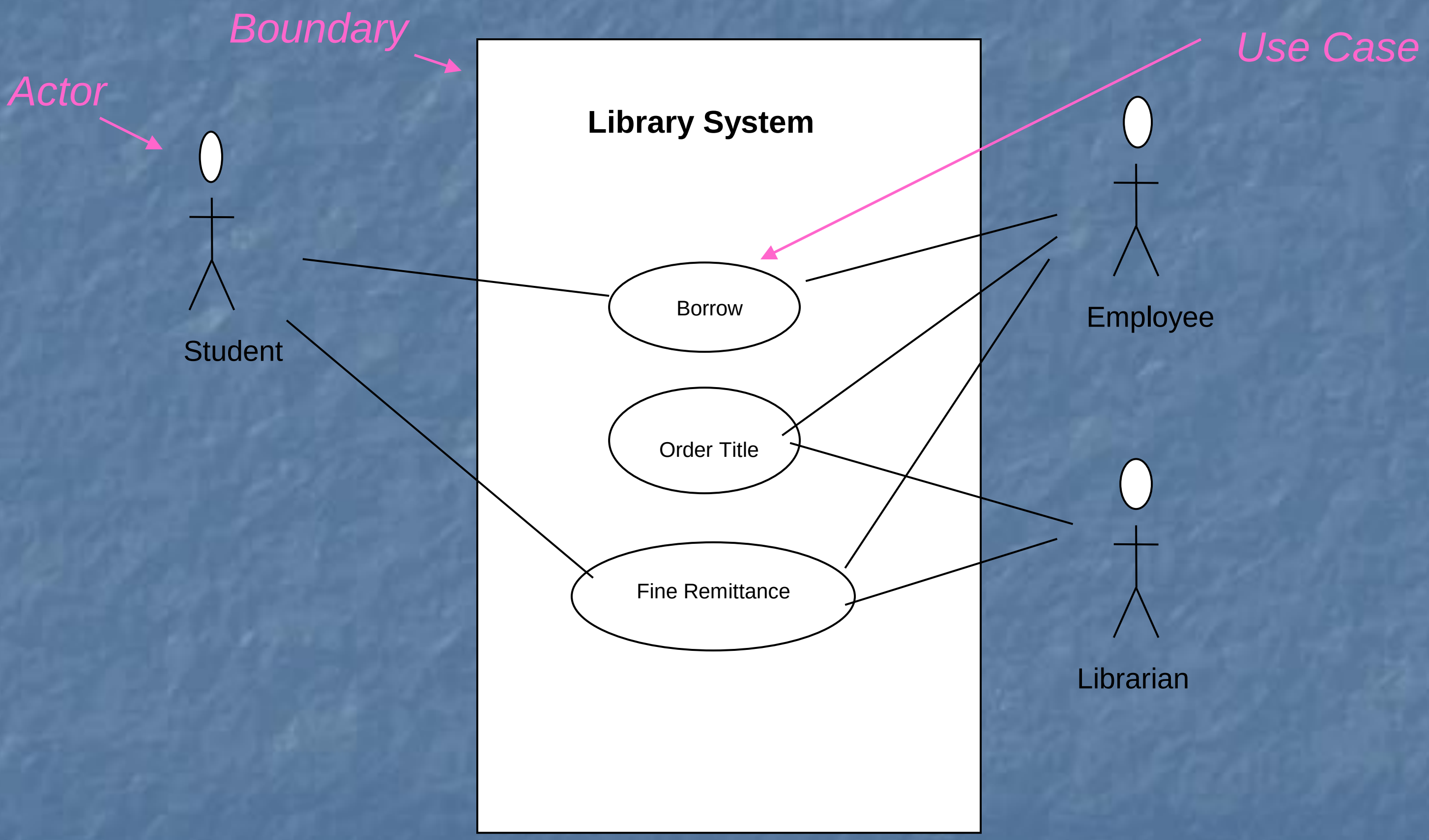
Include: a dotted line labeled <<include>> beginning at base use case and ending with an arrows pointing to the include use case. The include relationship occurs when a chunk of behavior is similar across more than one use case. Use "include" in stead of copying the description of that behavior.

$$\begin{tikzcd} {} \arrow[rr, dashed, "\text{<<include>>}"] & & {} \end{tikzcd}$$

Extend: a dotted line labeled <<extend>> with an arrow toward the base case. The extending use case may add behavior to the base use case if required. The base class declares "extension points" and is **optional**.

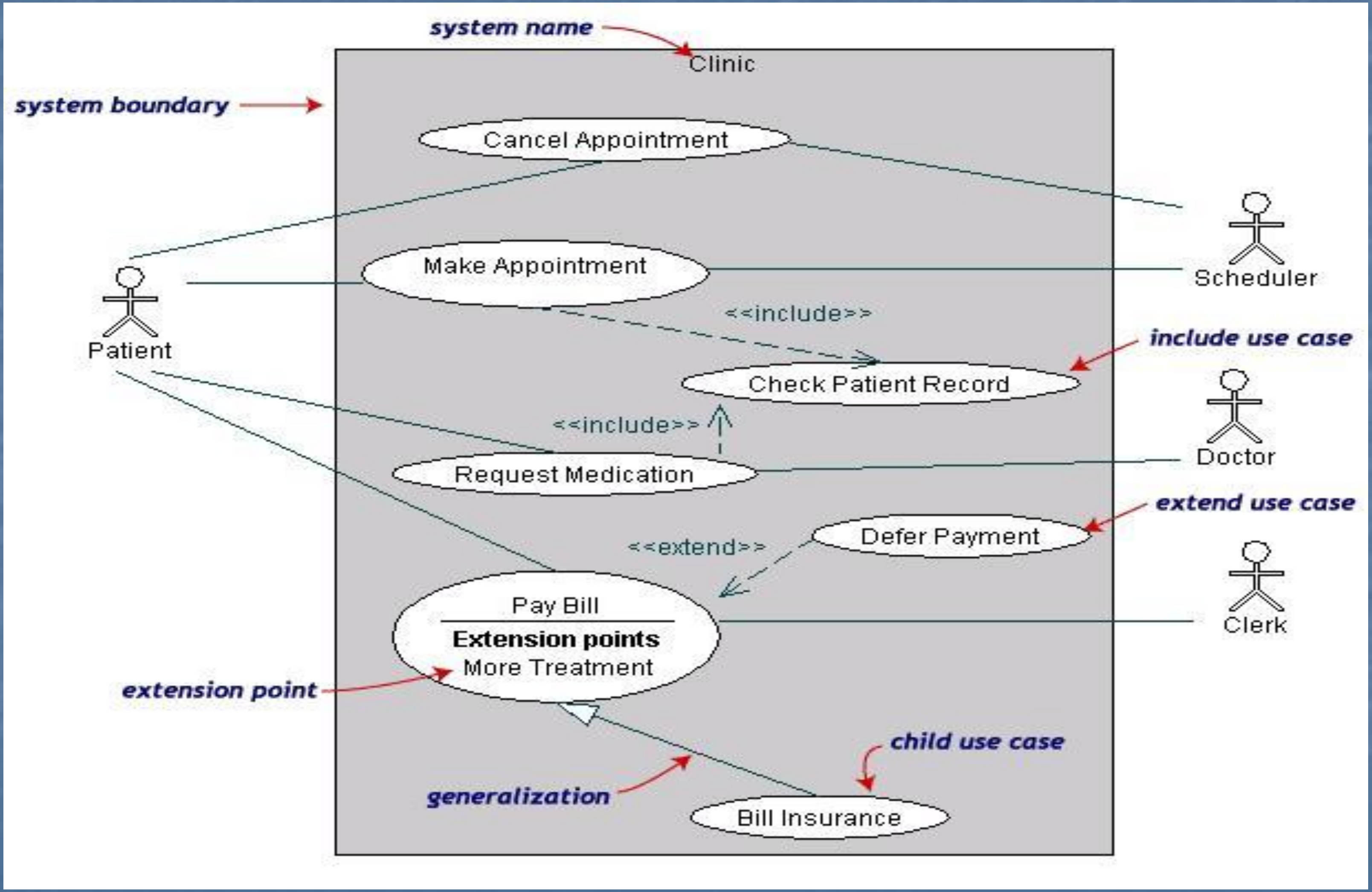
$$\begin{tikzcd} {} \arrow[rr, dashed, "\text{<<extend>>}"] & & {} \end{tikzcd}$$

Use Case Diagrams



- A generalized description of how a system will be used.
- Provides an overview of the intended functionality of the system

Use Case Diagrams(cont.)



(clinic mgt system)

Use Case Diagrams (cont.)

•**Pay Bill** is a parent use case and **Bill Insurance** is the child use case. (generalization)

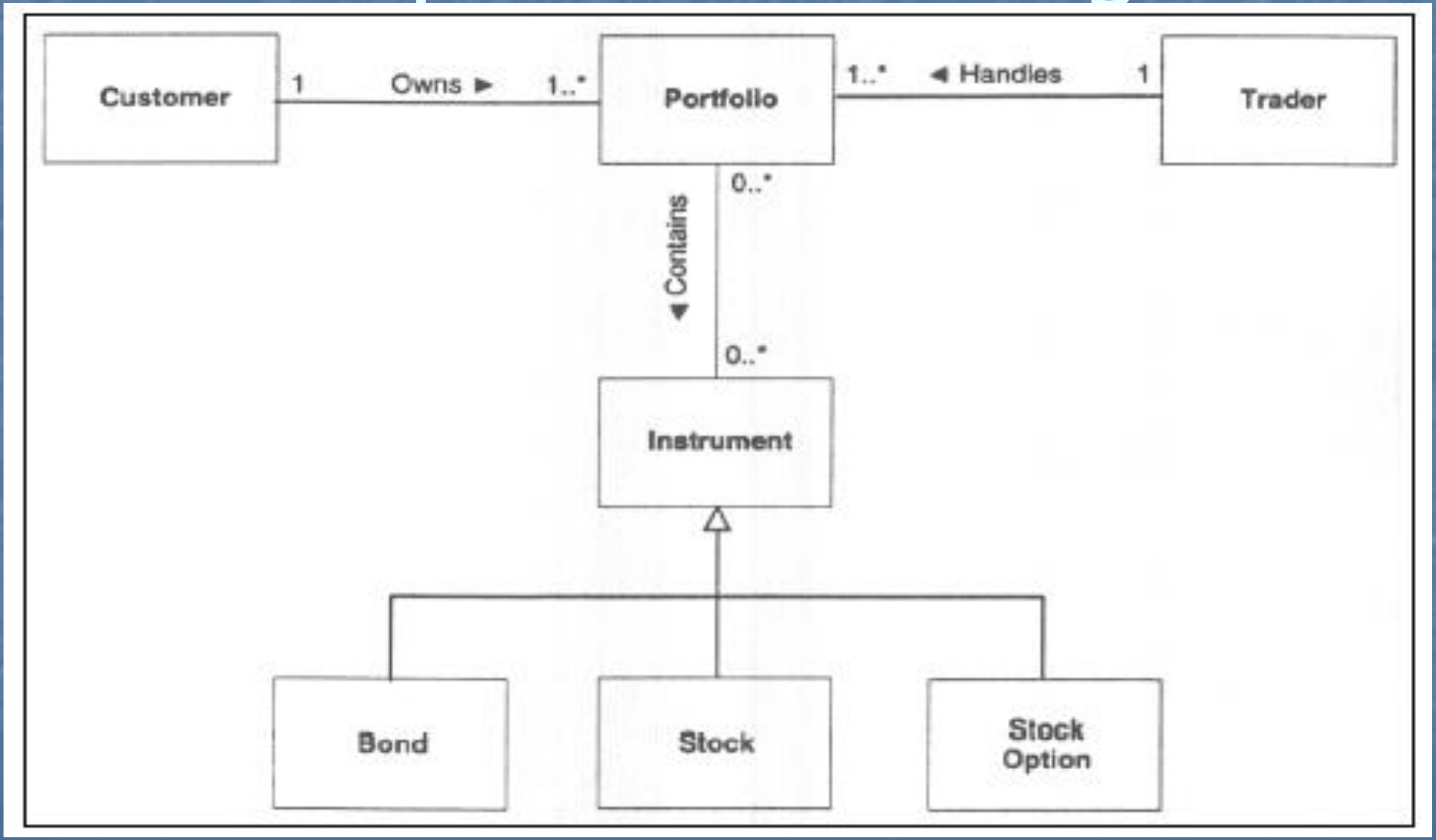
•Both **Make Appointment** and **Request Medication** include **Check Patient Record** as a subtask (include)

•The **extension point** is written inside the base case **Pay bill**; the extending class **Defer payment** adds the behavior of this extension point. (extend)

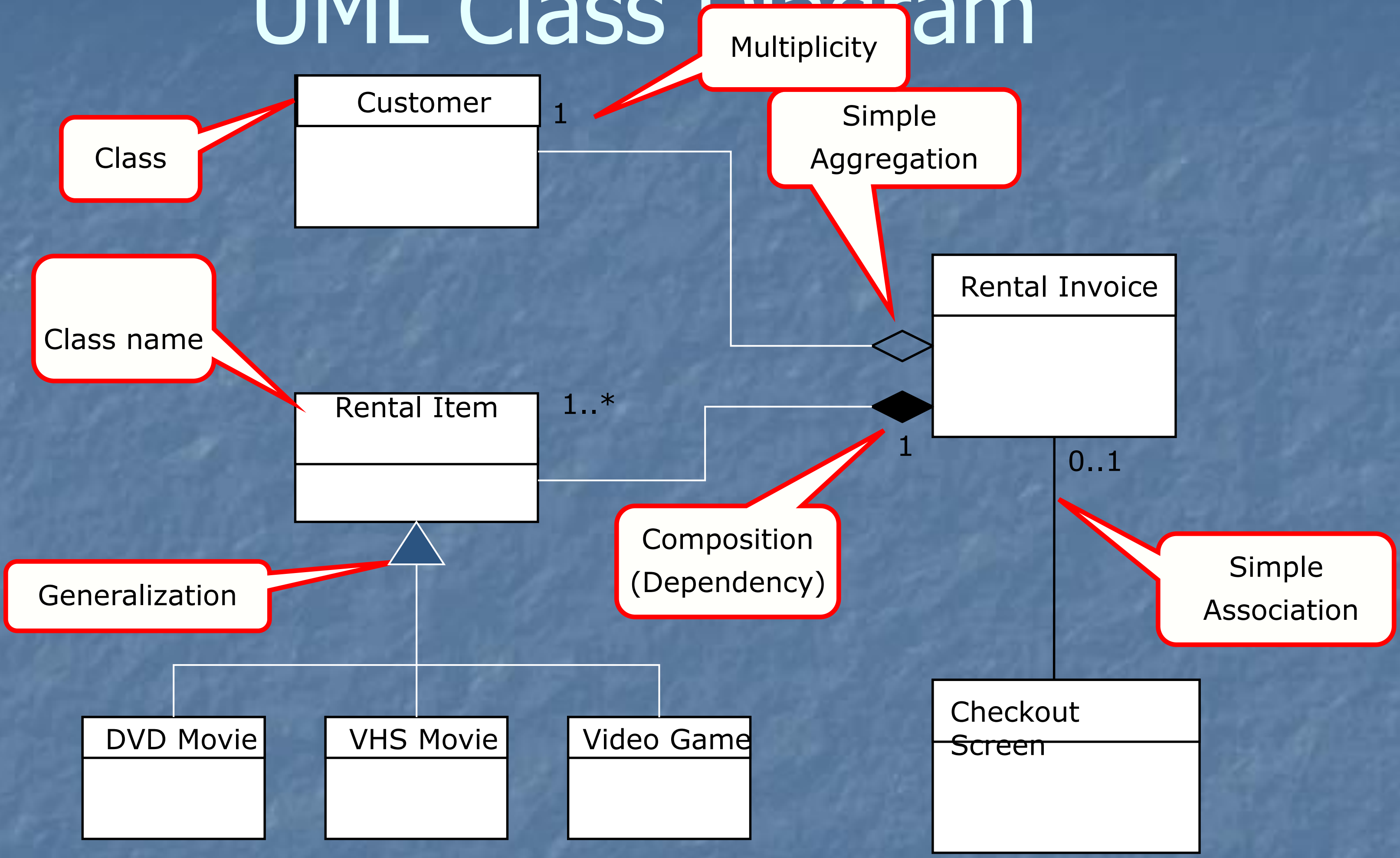
Class diagram

- Used for describing structure and behavior in the use cases
- Provide a conceptual model of the system in terms of entities and their relationships
- Used for requirement capture, end-user interaction
- Detailed class diagrams are used for developers

Conceptual Class Diagram



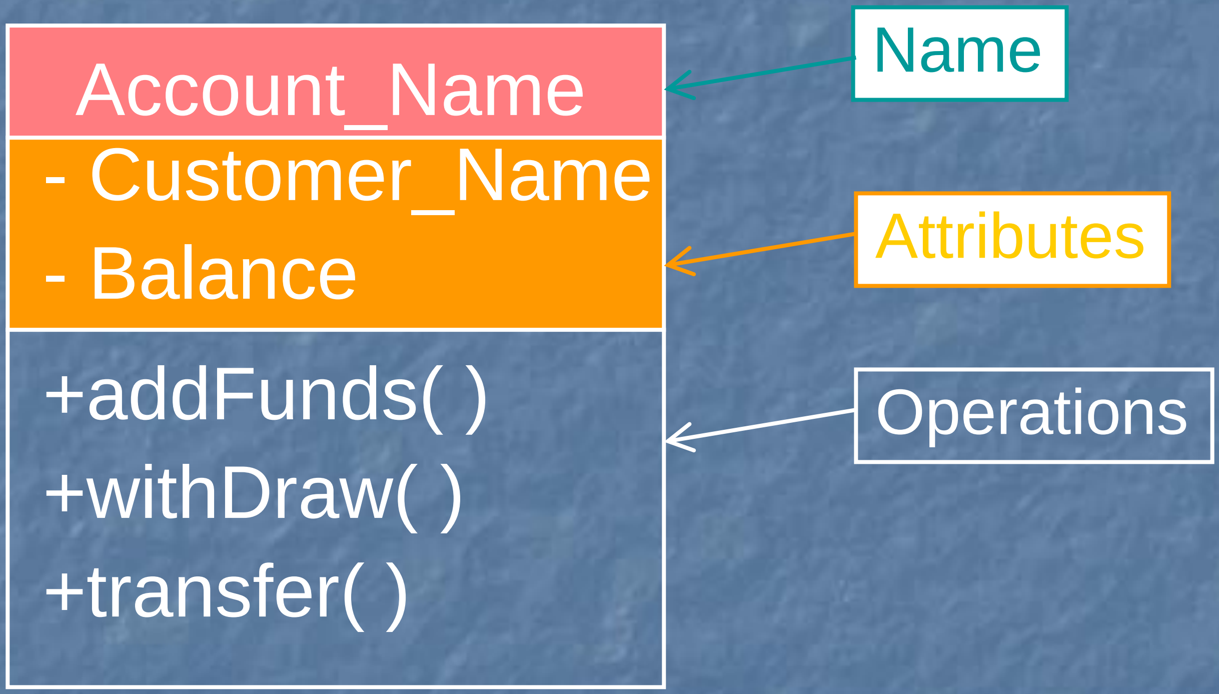
UML Class Diagram



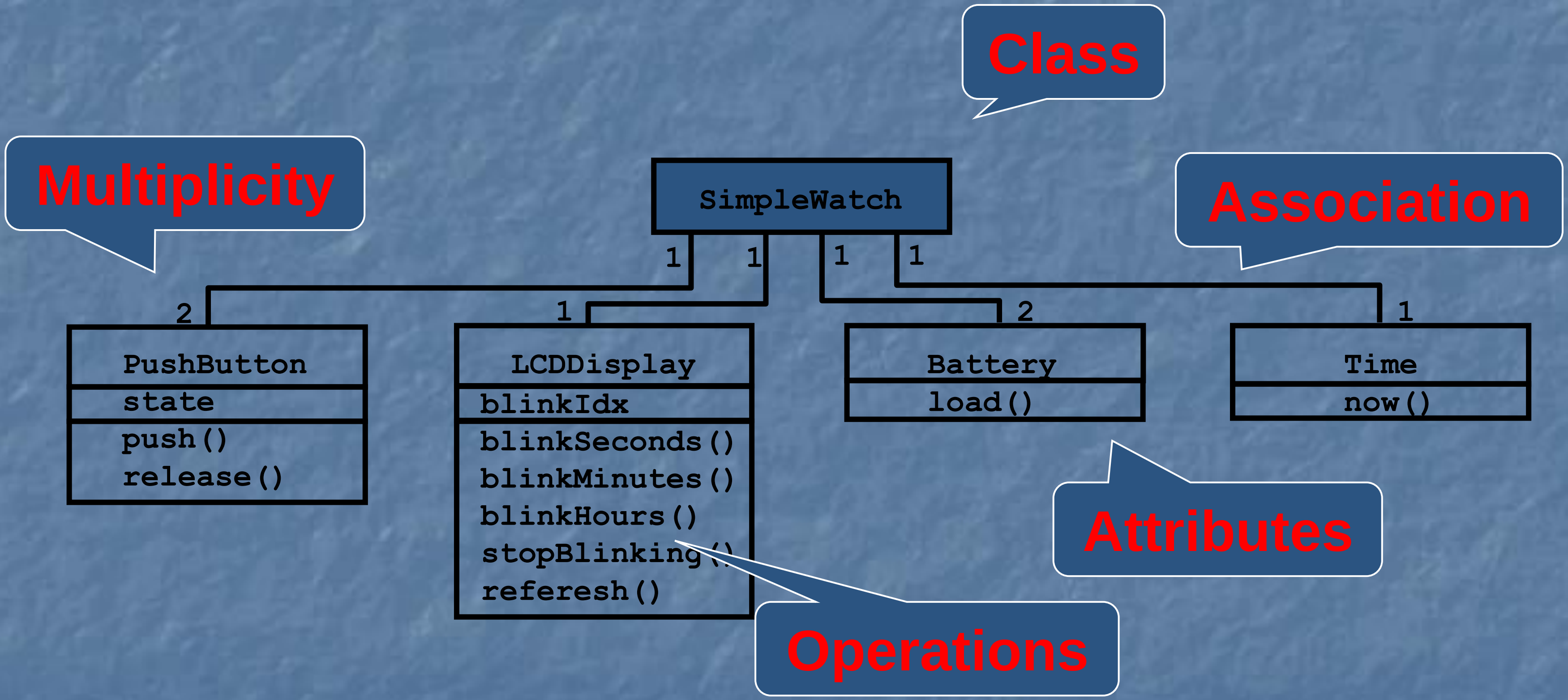
Class representation

- Each class is represented by a rectangle subdivided into three compartments
  - Name
  - Attributes
  - Operations
- Modifiers are used to indicate visibility of attributes and operations.
  - '+' is used to denote ***Public*** visibility (everyone)
  - '#' is used to denote ***Protected*** visibility (friends and derived)
  - '-' is used to denote ***Private*** visibility (no one)
- By default, attributes are hidden and operations are visible.

An example of Class



UML First Pass: Class Diagrams

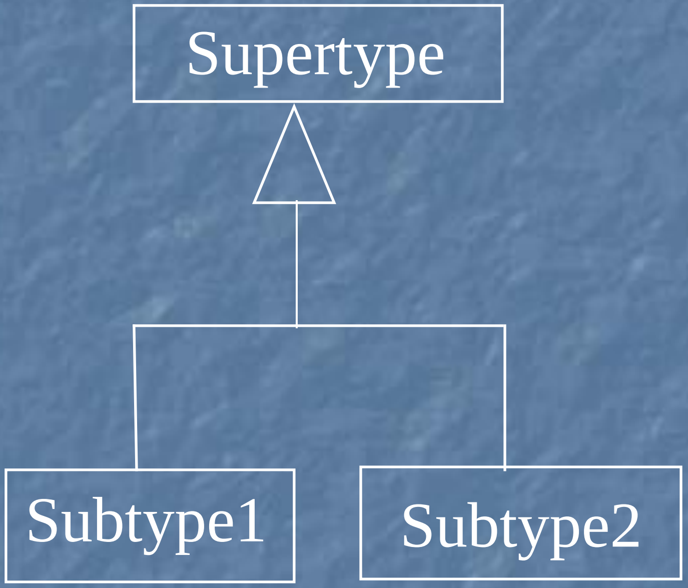


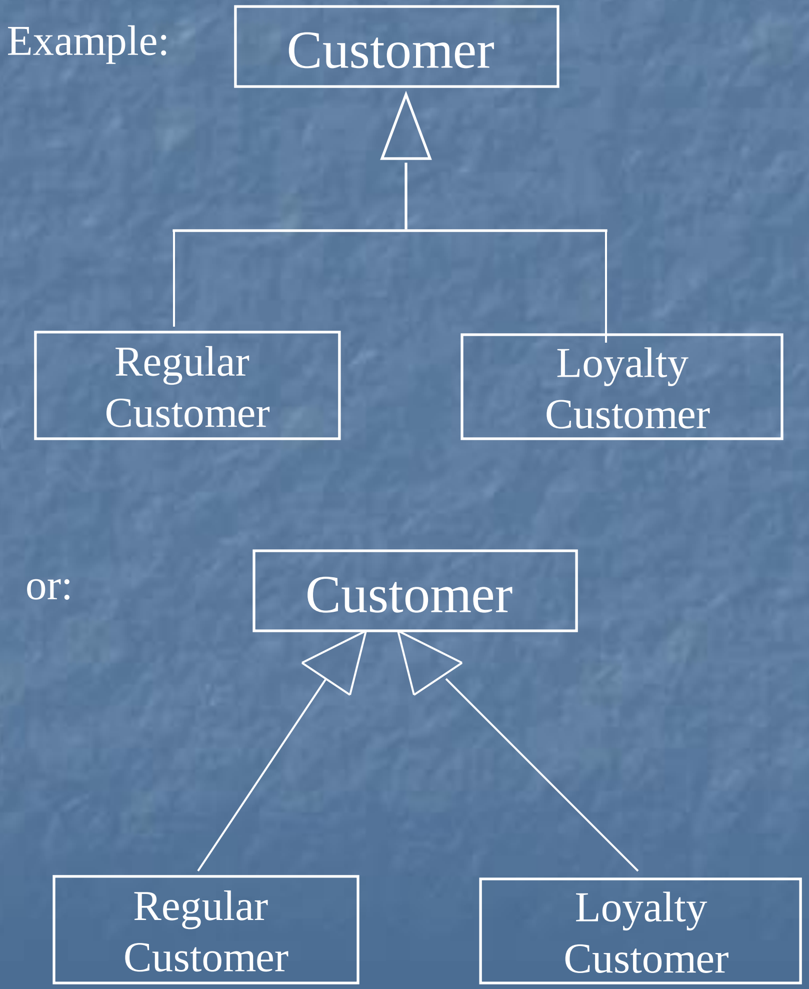
Class diagrams represent the structure of the system

OO Relationships

- There are two kinds of Relationships
  - Generalization (parent-child relationship)
  - Association ()
- Associations can be further classified as
  - Aggregation
  - Composition

OO Relationships: **Generalization**





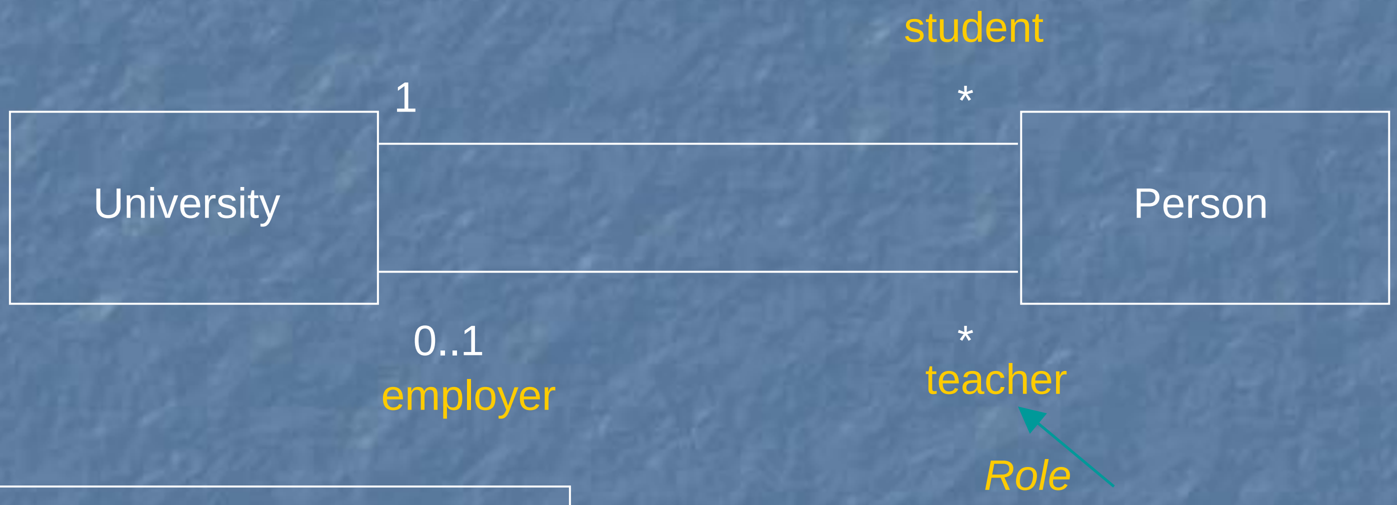
- Generalization expresses a parent/child relationship among related classes.

- Used for abstracting details in several layers

OO Relationships: **Association**

- Represent relationship between instances of classes
  - Student enrolls in a course
  - Courses have students
  - Courses have exams
  - Etc.
- Association has two ends
  - Role names (e.g. enrolls)
  - Multiplicity (e.g. One course can have many students)
  - Navigability (unidirectional, bidirectional)

Association: Multiplicity and Roles



Multiplicity

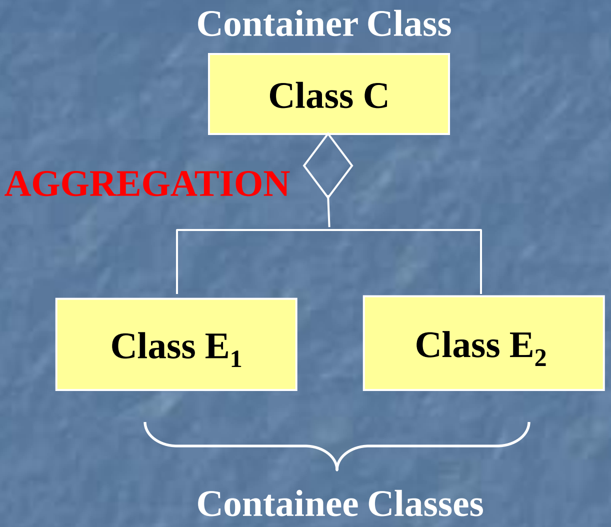
Symbol	Meaning
1	One and only one
0..1	Zero or one
M..N	From M to N (natural language)
*	From zero to any positive integer
0..*	From zero to any positive integer
1..*	From one to any positive integer

Role

"A given university groups many people; some act as students, others as teachers. A given student belongs to a single university; a given teacher may or may not be working for the university at a particular time."

- **Association** is a relationship where all objects have their own lifecycle and there is no owner.
- An example is of Teacher and Student. Multiple students can associate with single teacher and single student can associate with multiple teachers, but there is no ownership between the objects and both have their own lifecycle. Both can create and delete independently

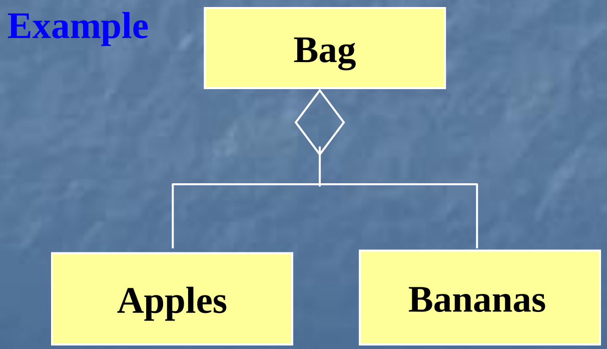
OO Relationships: **Aggregation**



Aggregation: expresses a relationship among instances of related classes. It is a specific kind of Container-Containee
relationship.

"is-part-of" "has –a"

It expresses a relationship where an instance of the Container-class has the responsibility to hold and maintain instances of each Containee-class that have been created outside the auspices of the Container-class.

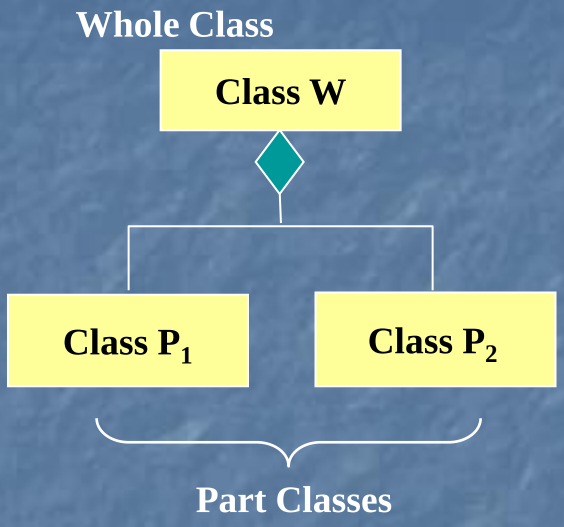


Aggregation should be used to express a more informal relationship than composition expresses. That is, it is an appropriate relationship where the Container and its Containees can be manipulated independently.

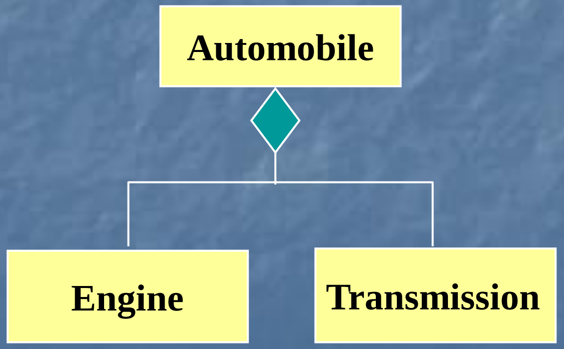
Aggregation is appropriate when Container and Containees have no special access privileges to each other.

- *Aggregation* can occur when a class is a collection or container of other classes, but the contained classes do not have a strong *lifecycle dependency* on the container.
- **The contents of the container are not automatically destroyed when the container is destroyed.**

OO Relationships: **Composition**



Example



Composition: expresses a relationship among instances of related classes. It is a specific kind of Whole-Part relationship.

It expresses a relationship where an instance of the Whole-class has the responsibility to create and initialize instances of each Part-class.

It may also be used to express a relationship where instances of the Part-classes have privileged access or visibility to certain attributes and/or behaviors defined by the Whole-class.

Composition should also be used to express relationship where instances of the Whole-class have exclusive access to and control of instances of the Part-classes.

Composition should be used to express a relationship where the behavior of Part instances is undefined without being related to an instance of the Whole. And, conversely, the behavior of the Whole is ill-defined or incomplete if one or more of the Part instances are undefined.

- **Composition** is a specialised form of Aggregation and we can call this as a “death” relationship.
- It is a **strong** type of Aggregation. Child object does not have its lifecycle and if parent object is deleted, all child objects will also be deleted.
- An example of relationship between **House and Rooms**. House can contain multiple rooms - there is no independent life of room and any room can not belong to two different houses. If we delete the house - room will automatically be deleted.
- Another example relationship between **Questions and Options**. Single questions can have multiple options and If we delete questions, options will automatically be deleted.

Aggregation vs. Composition

•**Composition** is really a strong form of **aggregation** (**strong bonding**)

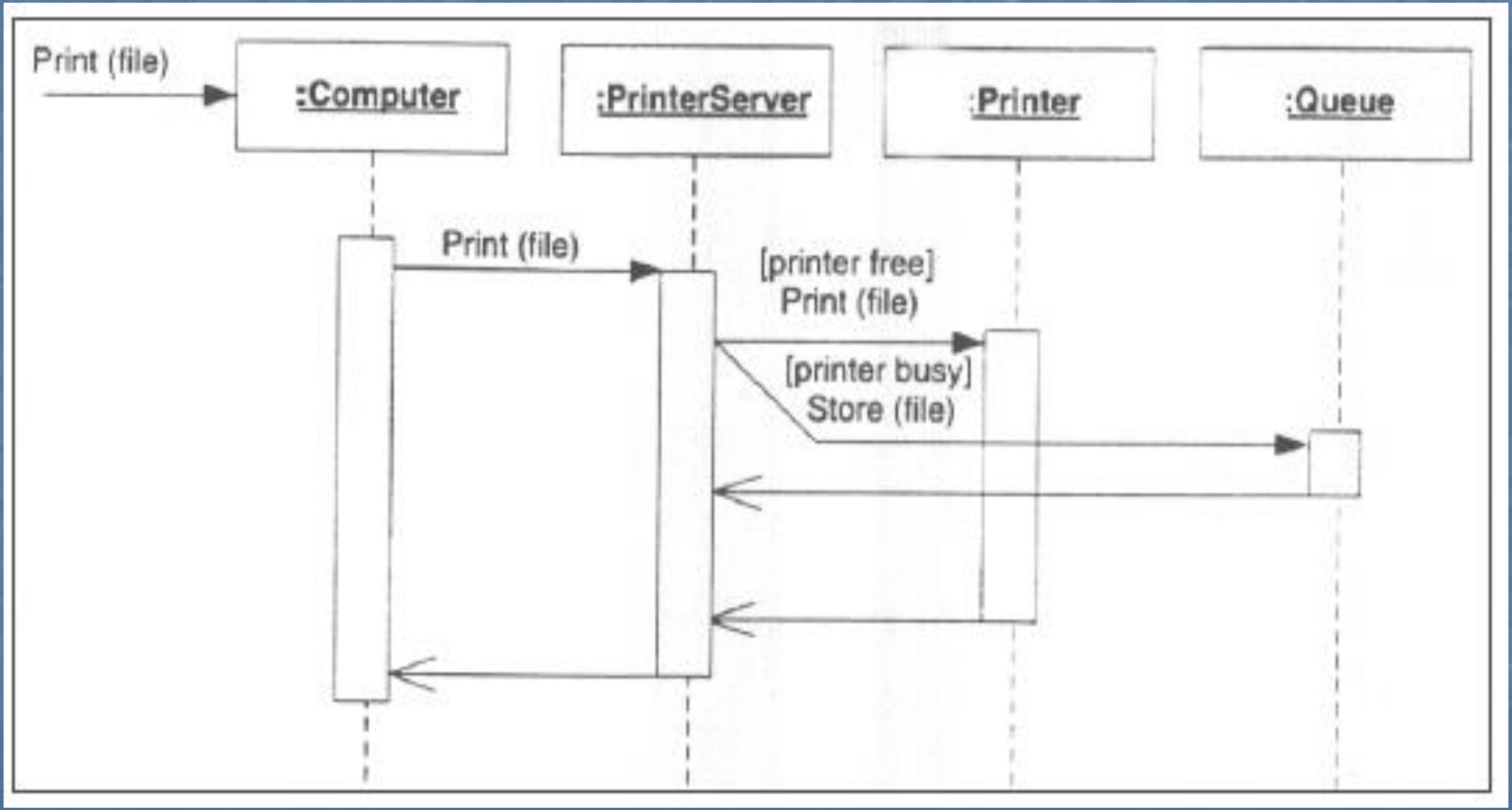
- •components have only one owner
- •components **cannot exist independent** of their owner
- •components live or die with their owner

e.g. Each car has an engine that can not be shared with other cars.

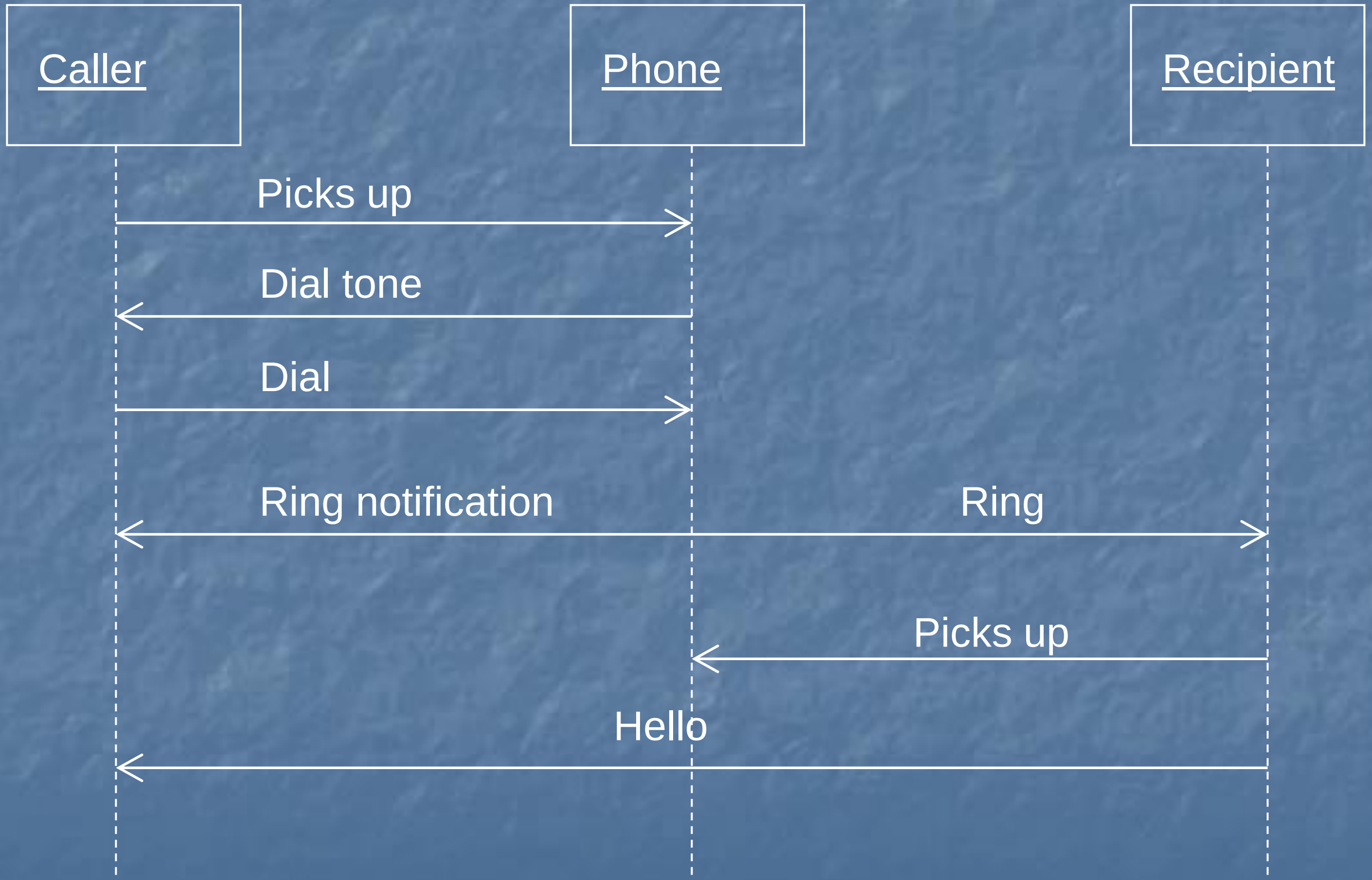
•**Aggregations** may form "**part of**" or ‘uses’ the aggregate, but may not be essential to it. They may also exist independent of the aggregate.

e.g. Apples may exist independent of the bag. (**loose bonding**)

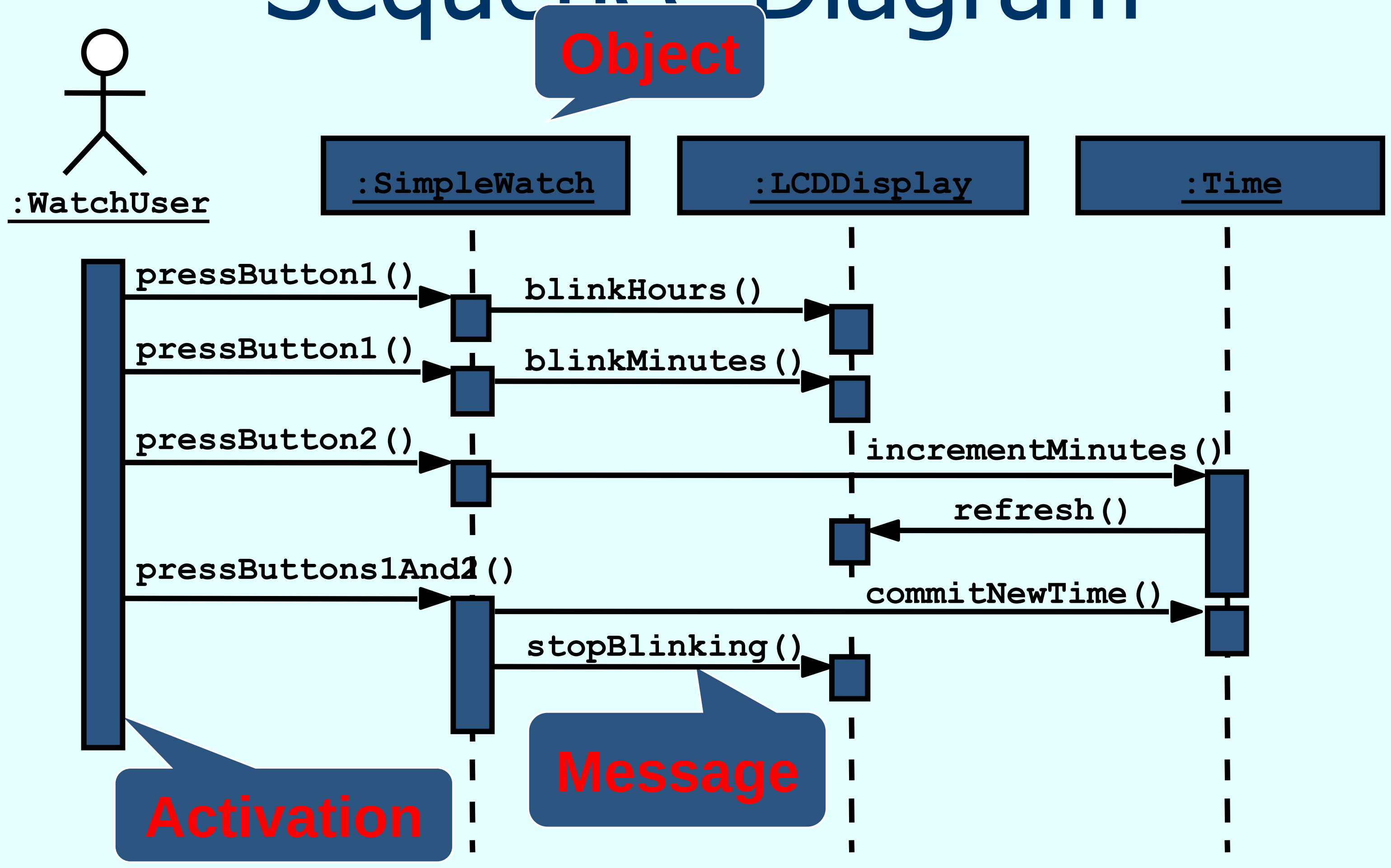
Sequence Diagram



Sequence Diagram (make a phone call)



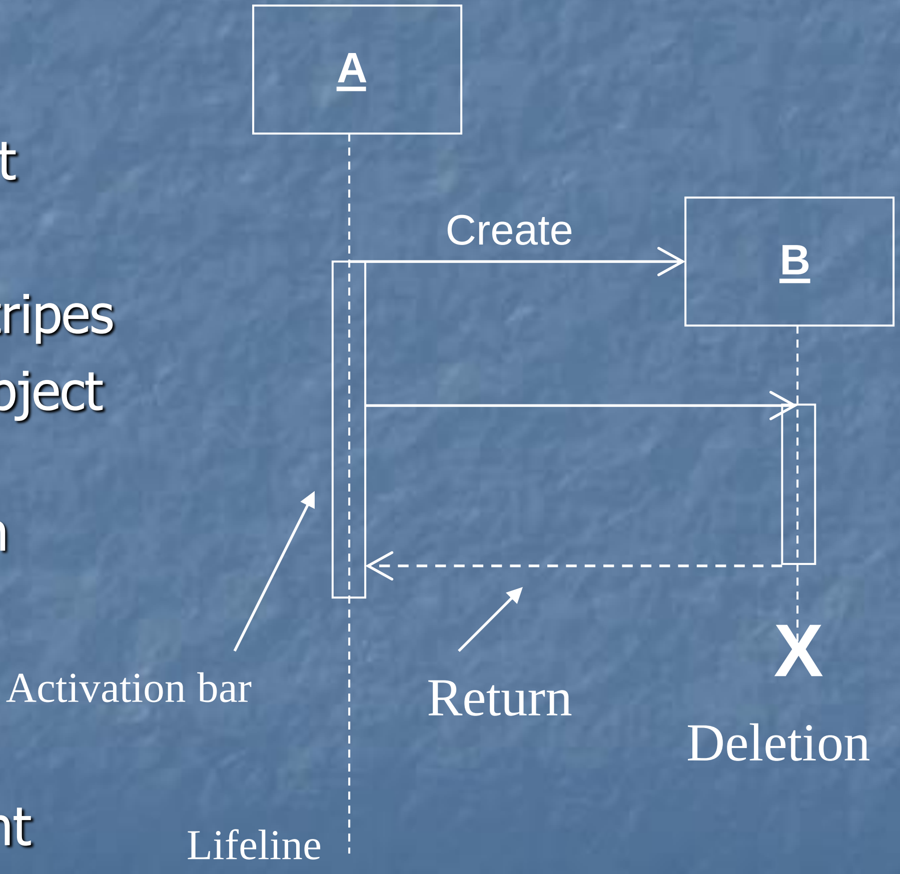
Sequence Diagram



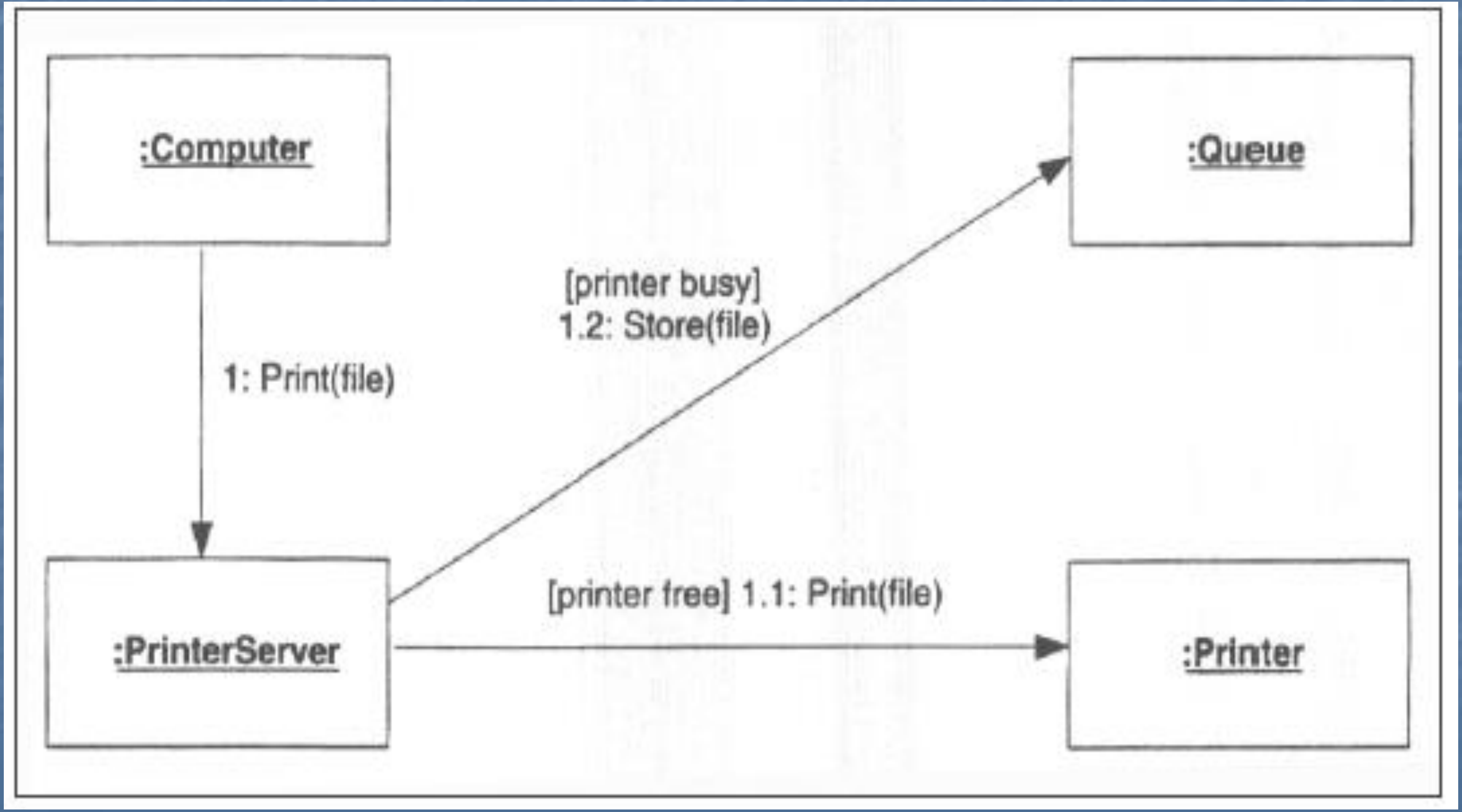
Sequence diagrams represent the behavior as interactions

Sequence Diagrams – Object Life Spans

- Creation
  - Create message
  - Object life starts at that point
- Activation
  - Symbolized by rectangular stripes
  - Place on the lifeline where object is activated.
  - Rectangle also denotes when object is deactivated.
- Deletion
  - Placing an 'X' on lifeline
  - Object's life ends at that point

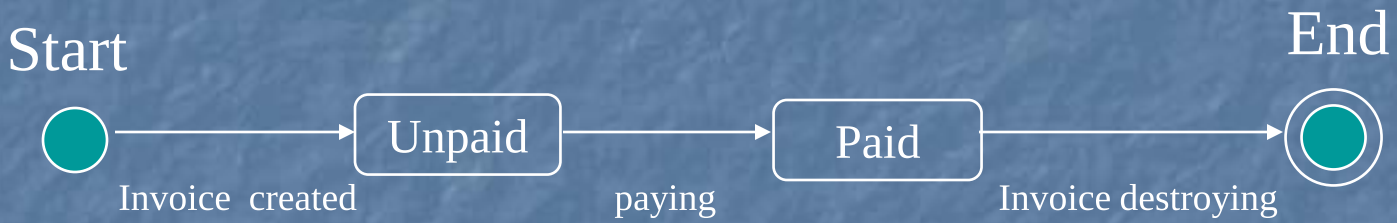


Collaboration Diagram

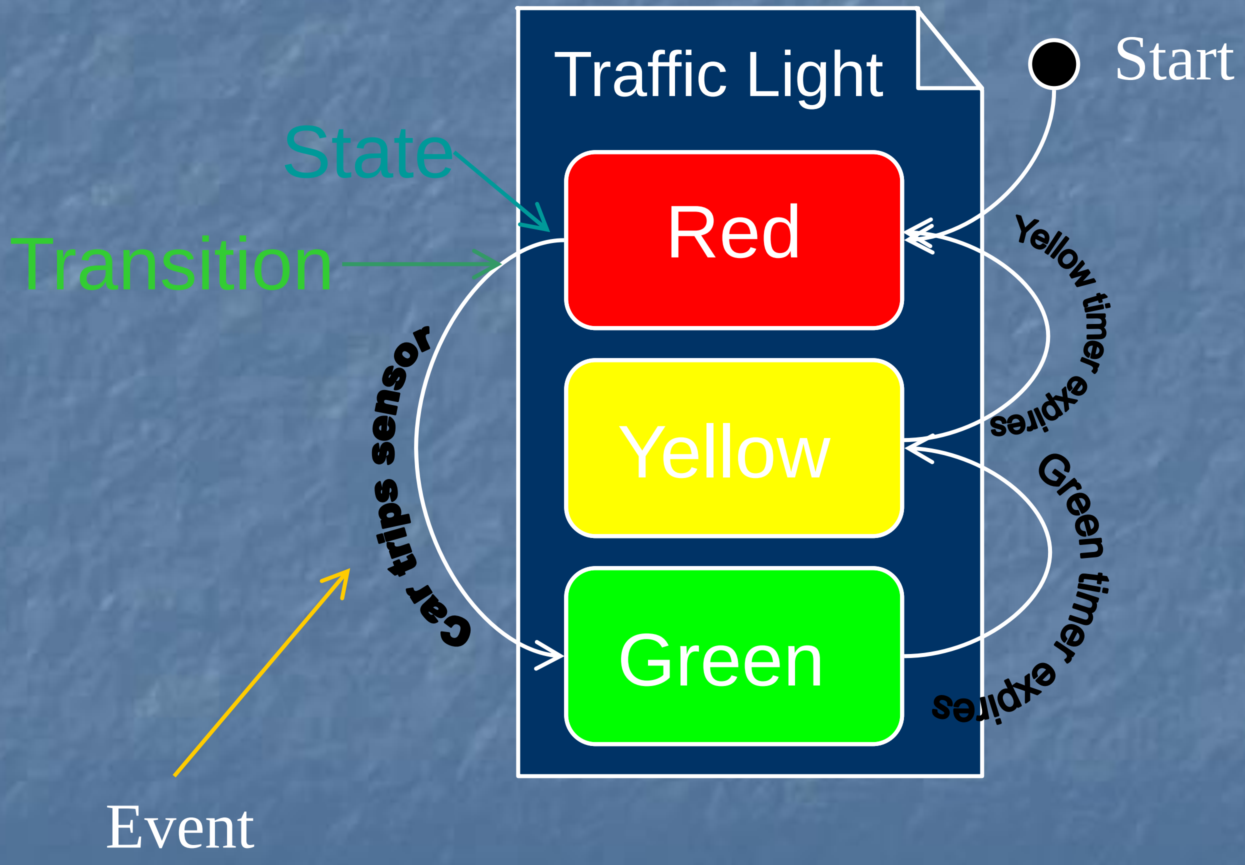


State Diagrams
(Billing Example)

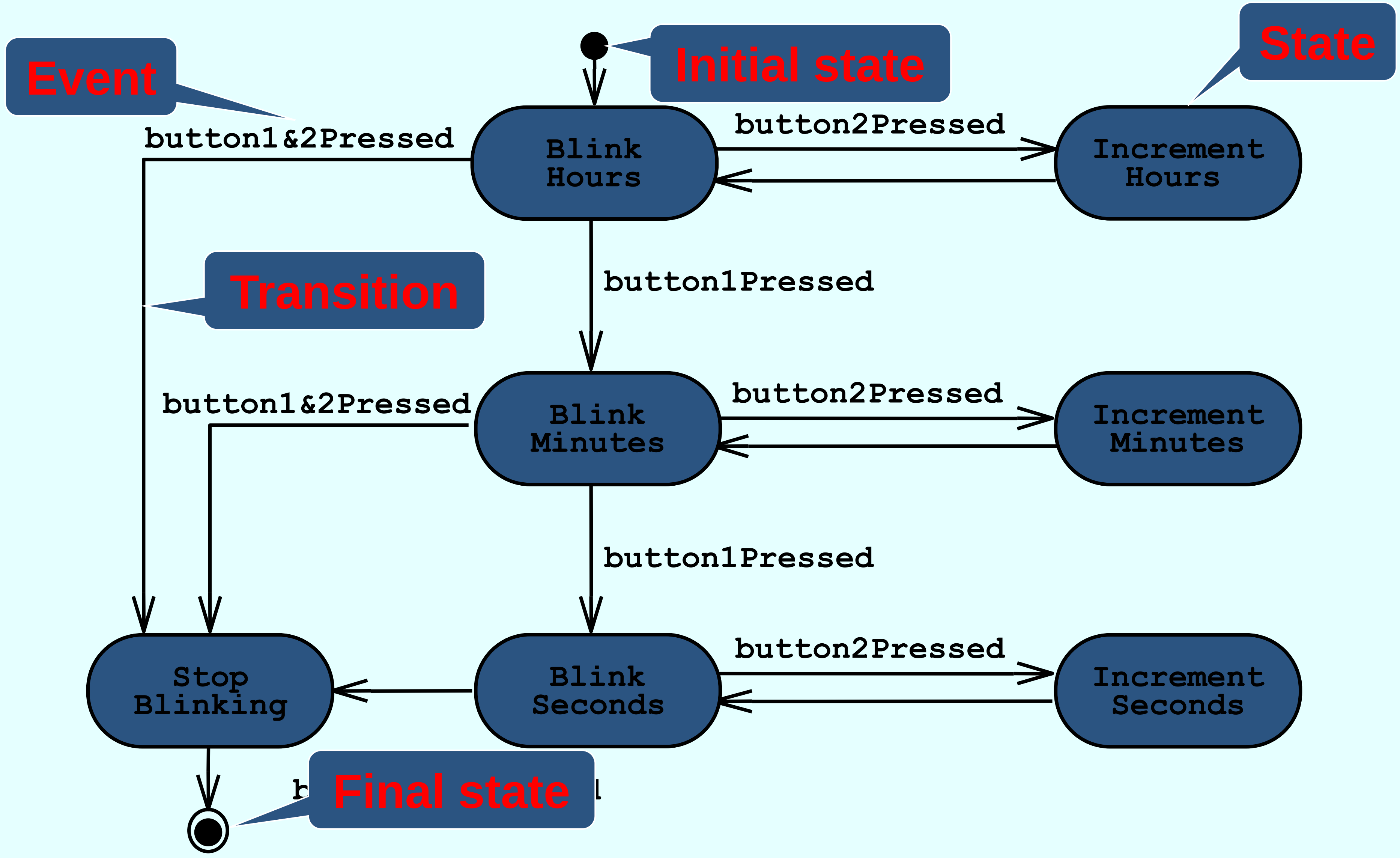
State Diagrams show the sequences of states an object goes through during its life cycle in response to stimuli, together with its responses and actions; an abstraction of all possible behaviors.



State Diagrams
(Traffic light example)



Statechart Diagrams (digital watch)



Event
Initial state
State
Transition
Final state
button1&2Pressed
Blink Hours
button2Pressed
Increment Hours
button1Pressed
button1&2Pressed
Blink Minutes
button2Pressed
Increment Minutes
button1Pressed
Stop Blinking
Blink Seconds
button2Pressed
Increment Seconds