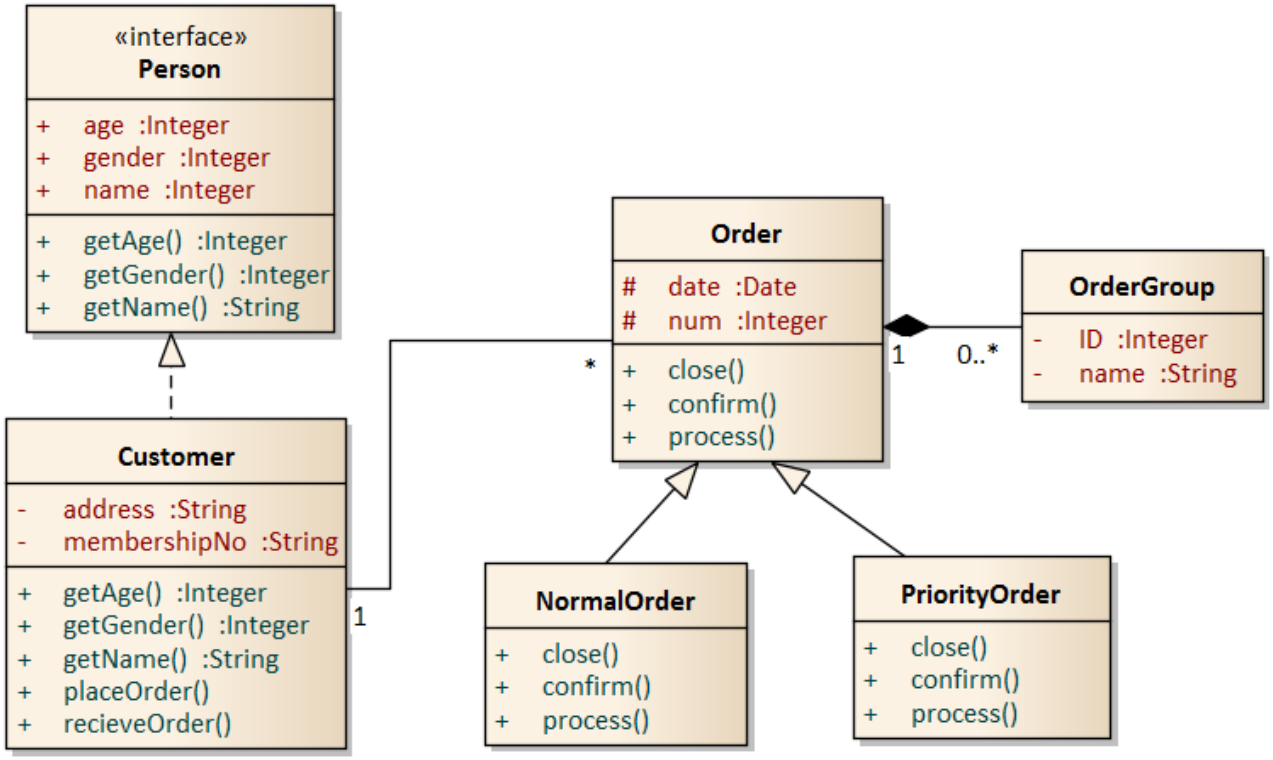


Object-Oriented Analysis and Design

UML – Design

Agenda

- Object Oriented Design
- Class Diagram
- Interaction Diagrams
  - Sequence Diagrams
  - Collaboration Diagrams
- State Transition (Chart) Diagrams
- Activity Diagrams
- Component Diagrams
- Deployment Diagrams

Object-Oriented Design

- A method for decomposing software architectures based on the objects every system or subsystem manipulates

- Object-oriented design concepts include:
  - Decomposition/Composition
  - Abstraction
    - Modularity
    - Information Hiding
  - Separating Policy and Mechanism
  - Subset Identification and Program Families
  - Reusability

- Main purpose of these design concepts is to manage software system complexity by improving software quality factors

Goals of the OO Design Phase

- Decompose System into Modules
  - i.e., identify the software architecture via “clustering"
  - In general, clusters should maximize cohesion and minimize coupling

- Determine Relations Between Modules
  - Identify and specify module dependencies
    - e.g., inheritance, composition, uses, etc.
  - Determine the form of intermodule communication, e.g.,
    - global variables
    - parameterized function calls
    - shared memory
    - message passing

Goals of the OO Design Phase

- Specify Module Interfaces
  - Interfaces should be well-defined
    - facilitate independent module testing
    - improve group communication
- Describe Module Functionality
  - Informally
    - e.g., comments or documentation
  - Formally
    - e.g., via module interface specification languages

Class Diagram

- The **structural view** represents the static aspect of a system. The structural view comprises class and object diagrams. Also know as Static Modelling.

- **Class diagrams** depict various classes and their associations. Associations are depicted as bi-directional connections between classes.

- **Object diagrams** depict various objects (instances of classes) and their links with each other. Links are pathways of communication between objects.

Class Diagram

A complete class diagram should include ….:

- Class Name
  - Name of the class
  - Example: GraduateStudent

- State
  - a set of attributes with their types and
  - Example:

    + age:int
    - salary:float
    - totalDue: double
    - enrollment: Date <<final>>

Class Diagram

- In the above example, the age attribute is a public attribute of type int
- Similarly, the enrollment attribute is a private attribute of a class type and cannot be modified (final).

- Behaviour

A set of methods with their parameters and return type

Example:

+ DisplayEmployee():void
+ CompareSalary(:Employee):double
+ CalculateSalary():void <<Abstract>>
+ CalculateTotal(:Employee[]):double

Class Diagram

- DisplayEmployee() is a public method that does not take any parameters nor returns any value.
- CompareSalary() is a public method that takes the reference to an employee object and return the difference in salaries.
- CalculateSalary() is a public method that takes no parameters and returns a salary amount. This method is abstract in that its implementation is deferred to its subclass.
- CalculateTotal() is a public method that takes an array of employee objects and return the total salary

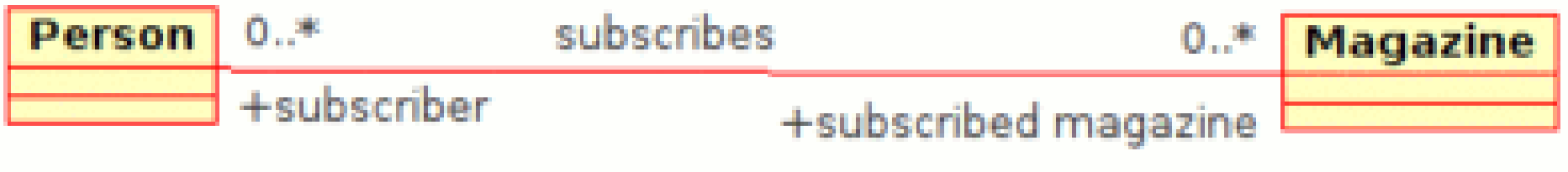
Class Diagram

Relationships

- Line - association
- Diamond – aggregation (weak form)
- Solid Diamond – aggregation (strong form)
- Arrow – Inheritance

Class Diagram

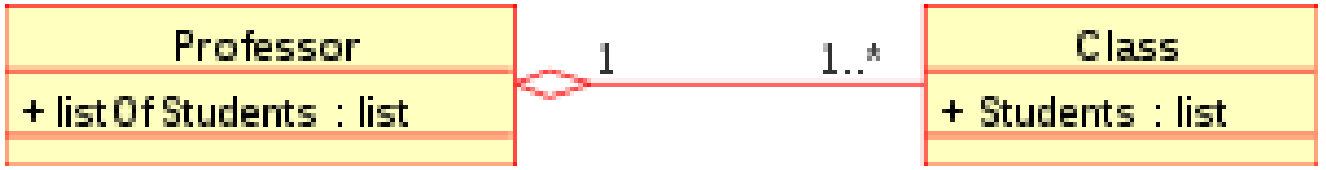
- **Association**
- An *association* represents a family of links. Binary associations (with two ends) are normally represented as a line. An association can be named, and the ends of an association can be adorned with role names, ownership indicators, multiplicity, visibility, and other properties.



- Class diagram example of association between two classes

Class Diagram

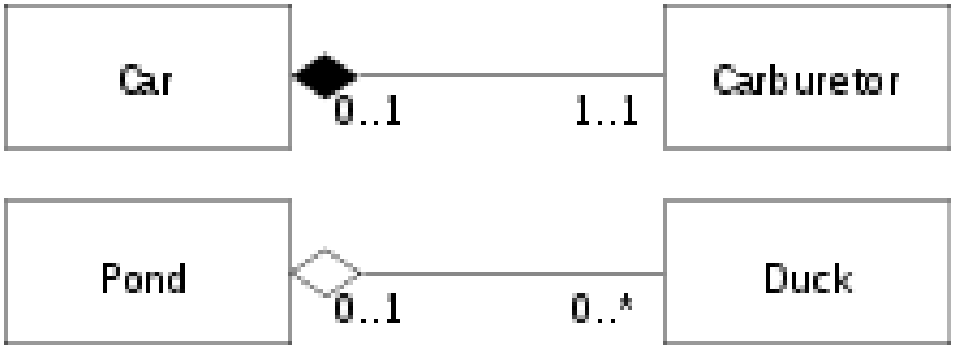
- **Aggregation**
- *Aggregation* is a variant of the "has a" association relationship; aggregation is more specific than association. It is an association that represents a part-whole or part-of relationship.



- Class diagram showing Aggregation between two classes

Class Diagram

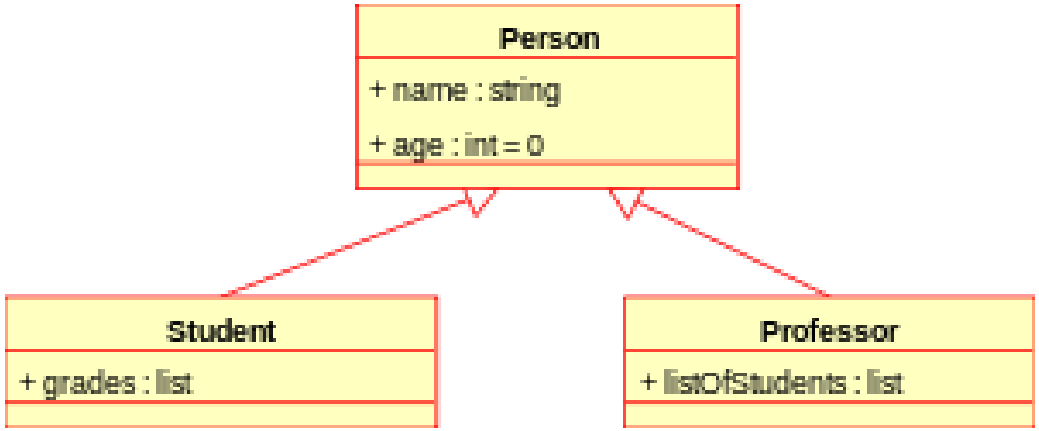
- **Composition**
- *Composition* is a stronger variant of the "has a" association relationship; composition is more specific than aggregation.
- *Composition* usually has a strong *life cycle dependency* between instances of the container class and instances of the contained class(es): if the container is destroyed, normally every instance that it contains is destroyed as well.



- Class diagram showing Composition between two classes at top and Aggregation between two classes at bottom

Class Diagram

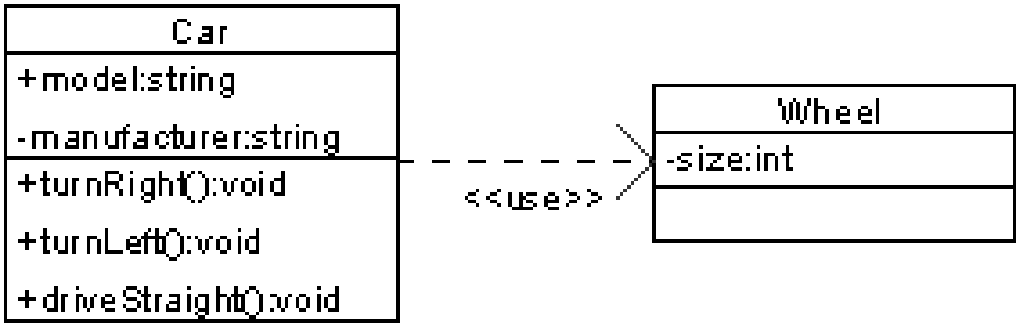
- **Generalization**
- The Generalization relationship ("is a") indicates that one of the two related classes (the *subclass*) is considered to be a specialized form of the other (the *super type*) and superclass is considered as '***Generalization***' of subclass.



- Class diagram showing generalization between one superclass and two subclasses

Class Diagram

- **Dependency**
- Dependency is a weaker form of bond which indicates that one class depends on another because it uses it at some point in time. One class depends on another if the independent class is a parameter variable or local variable of a method of the dependent class.



- Class diagram showing dependency between "Car" class and "Wheel" class

Class Diagram

- **Multiplicity**
- Multiplicity denotes the number of objects from one class that relate with a single object in an associated class, and is represented using multiplicity indicators.
- Multiplicity indicators are mostly used with associations and aggregations.
- The various multiplicity indicators are:

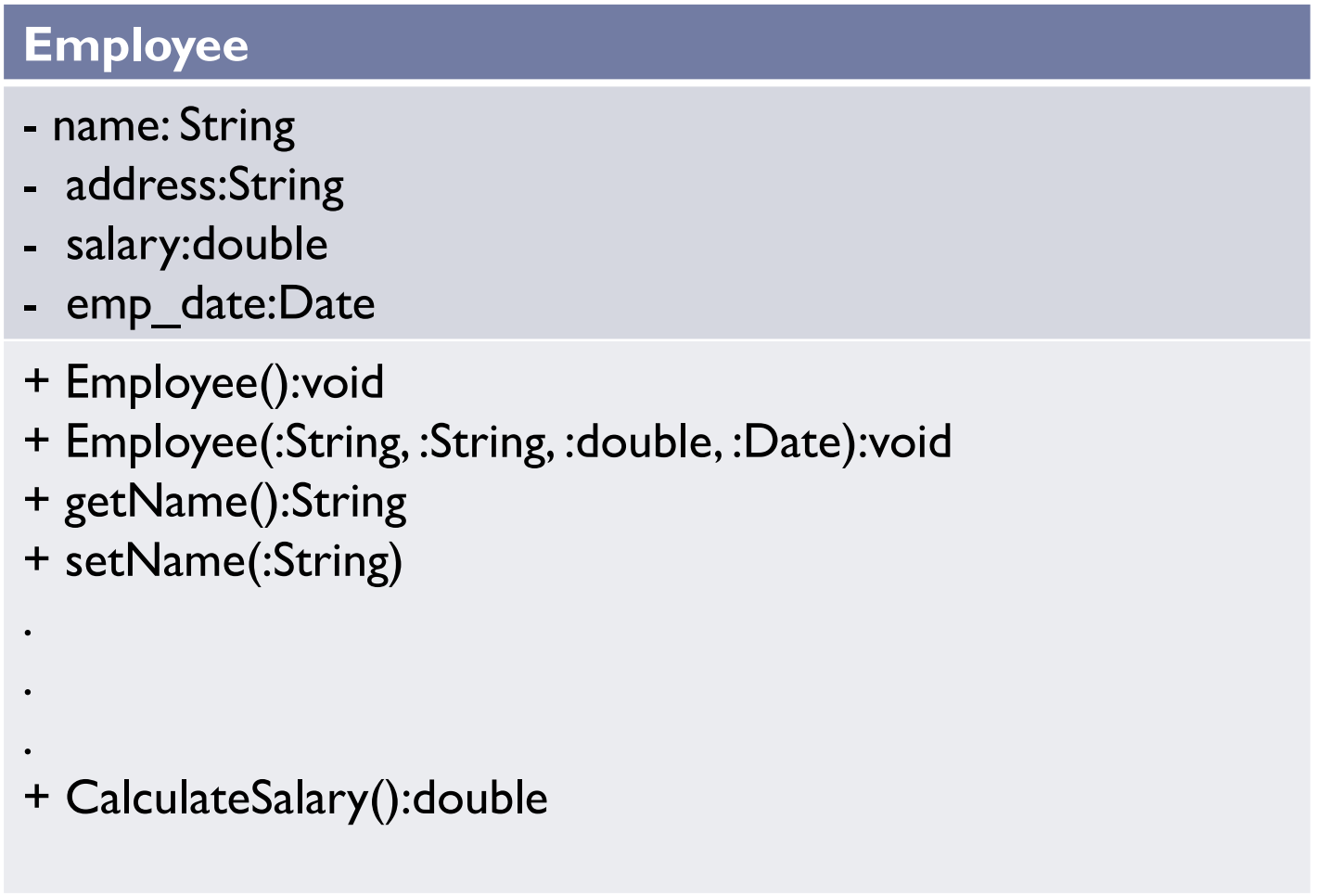
Representation	Multiplicity
1	Represents exactly one
0..*	Represents zero or more
1..*	Represents one or more
0..1	Represents zero or one
1..5	Represents specified range

Class Diagram

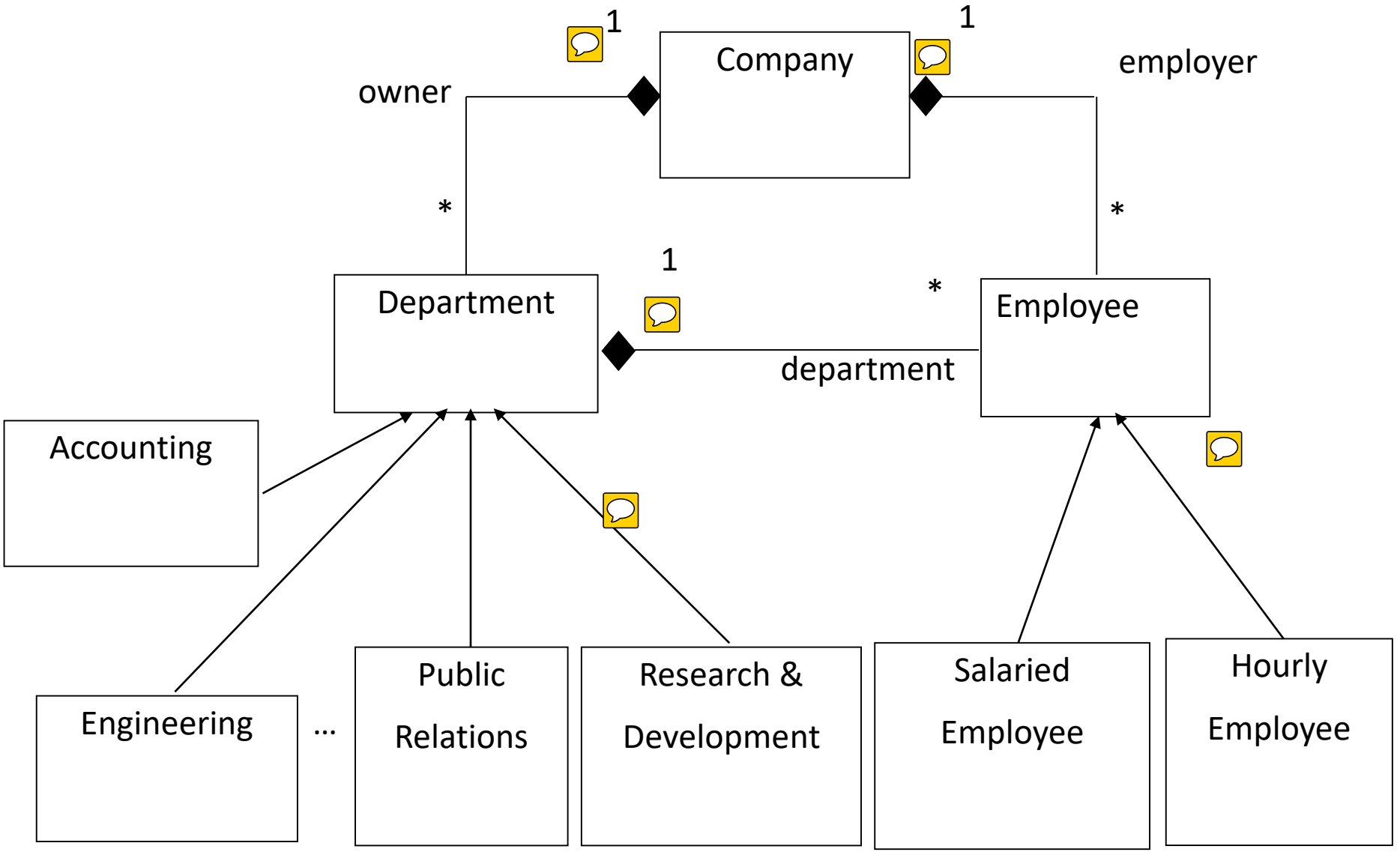
- **Visibility**
- To specify the visibility of a class member (i.e. any attribute or method) these are notations that must be placed before the member's name:

+	Public
-	Private
#	Protected
/	Derived (can be combined with one of the others)
~	Package

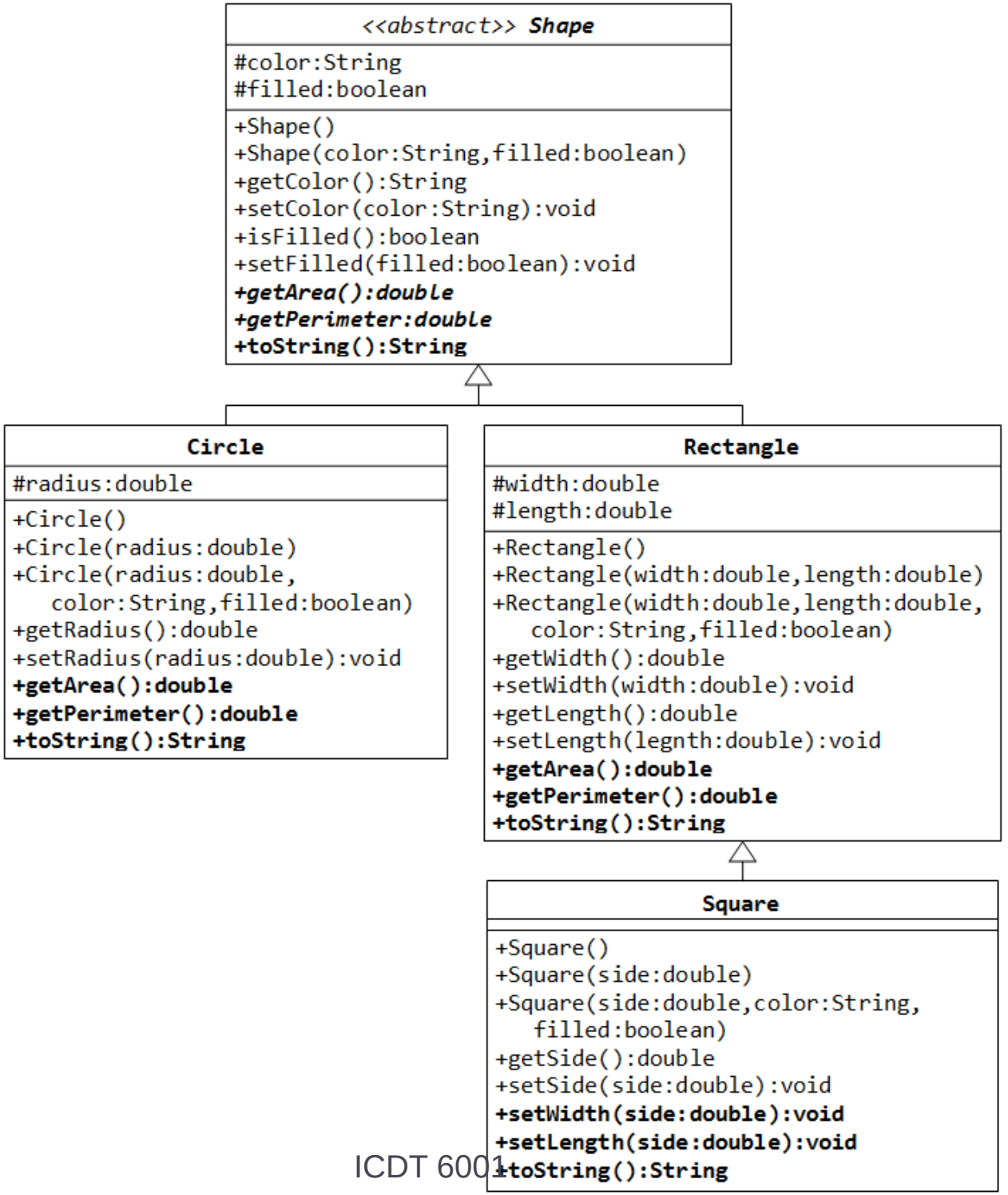
Class Diagram



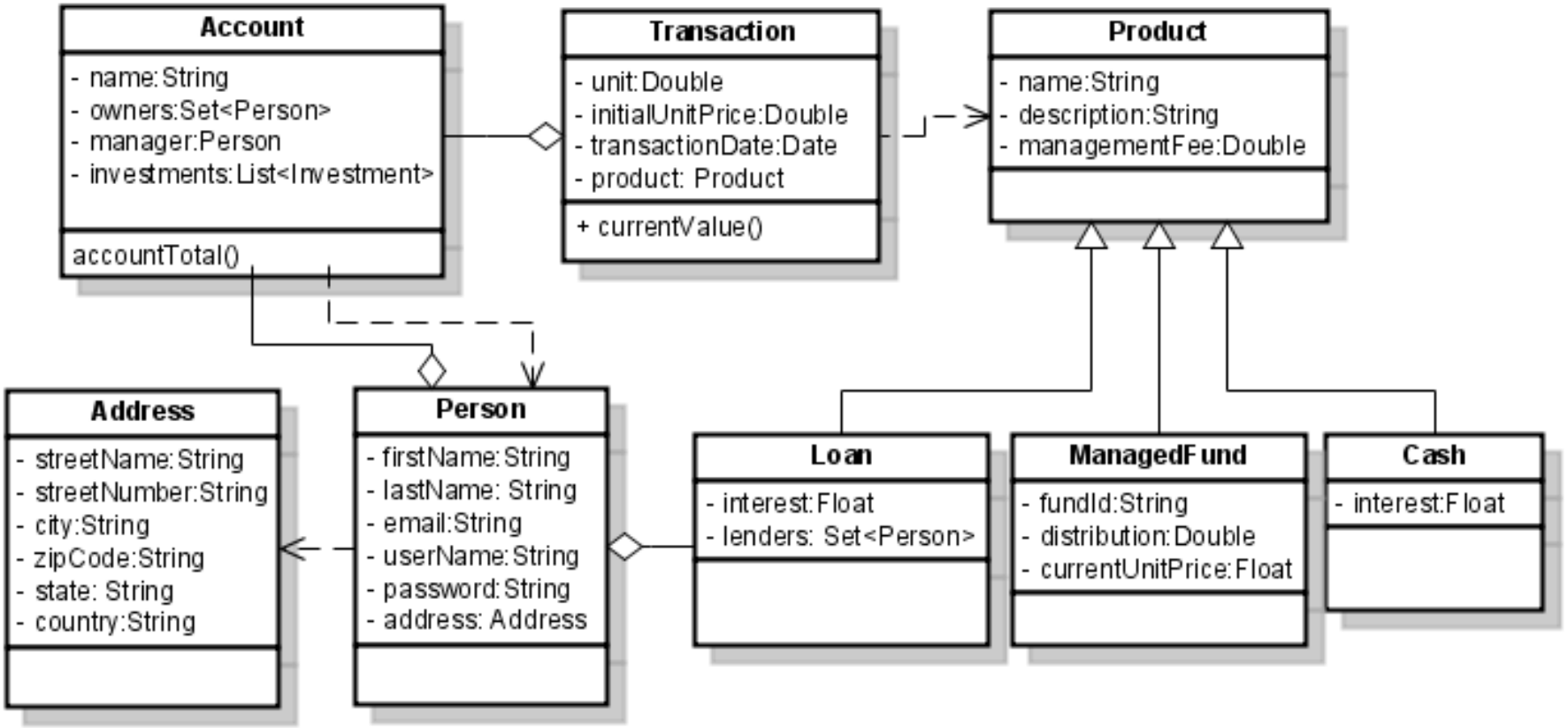
Class Diagram

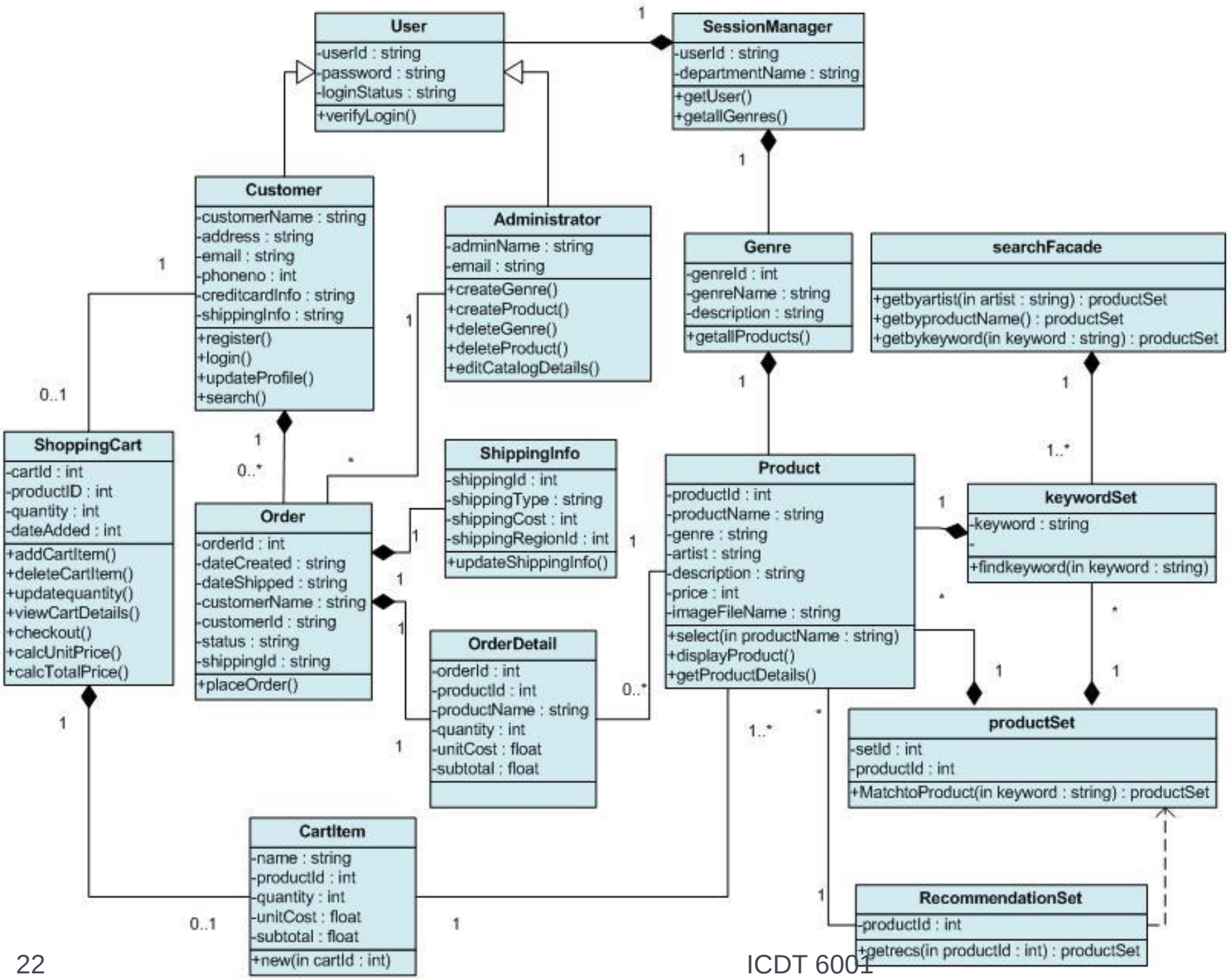


Class Diagram



Class Diagram




User
-userId : string
-password : string
-loginStatus : string
+verifyLogin()
SessionManager
-userId : string
-departmentName : string
+getUser()
+getallGenres()
Customer
-customerName : string
-address : string
-email : string
-phoneno : int
-creditcardInfo : string
-shippingInfo : string
+register()
+login()
+updateProfile()
+search()
Administrator
-adminName : string
-email : string
+createGenre()
+createProduct()
+deleteGenre()
+deleteProduct()
+editCatalogDetails()
Genre
-genreId : int
-genreName : string
-description : string
+getallProducts()
searchFacade
+getbyartist(in artist : string) : productSet
+getbyproductName() : productSet
+getbykeyword(in keyword : string) : productSet
ShoppingCart
-cartId : int
-productID : int
-quantity : int
-dateAdded : int
+addCartItem()
+deleteCartItem()
+updatequantity()
+viewCartDetails()
+checkout()
+calcUnitPrice()
+calcTotalPrice()
Order
-orderId : int
-dateCreated : string
-dateShipped : string
-customerName : string
-customerId : string
-status : string
-shippingId : string
+placeOrder()
ShippingInfo
-shippingId : int
-shippingType : string
-shippingCost : int
-shippingRegionId : int
+updateShippingInfo()
Product
-productId : int
-productName : string
-genre : string
-artist : string
-description : string
-price : int
-imageFileName : string
+select(in productName : string)
+displayProduct()
+getProductDetails()
keywordSet
-keyword : string
+findkeyword(in keyword : string)
OrderDetail
-orderId : int
-productId : int
-productName : string
-quantity : int
-unitCost : float
-subtotal : float
productSet
-setId : int
-productId : int
+MatchtoProduct(in keyword : string) : productSet
CartItem
-name : string
-productId : int
-quantity : int
-unitCost : float
-subtotal : float
+new(in cartId : int)
RecommendationSet
-productId : int
+getrecs(in productId : int) : productSet
1
0..1
0..*
*
1..*

Interaction Diagrams

- **Dynamic Modelling (Behavioural View)**
- Interaction diagrams are models that describe how a group of objects collaborate in some behavior - typically a single use-case – by exchanging messages to achieve a user goal. Recall a message is an object's invocation of an operation/service of some other object.

- Interaction diagrams fall under the dynamic view as they show the pattern of interaction that take place among objects in terms of service invocations. Many of the service invocations becomes methods of interacting objects, thus making our classes more functionally complete.

- There are two kinds of interaction models:
  - sequence diagrams
  - collaboration diagrams.

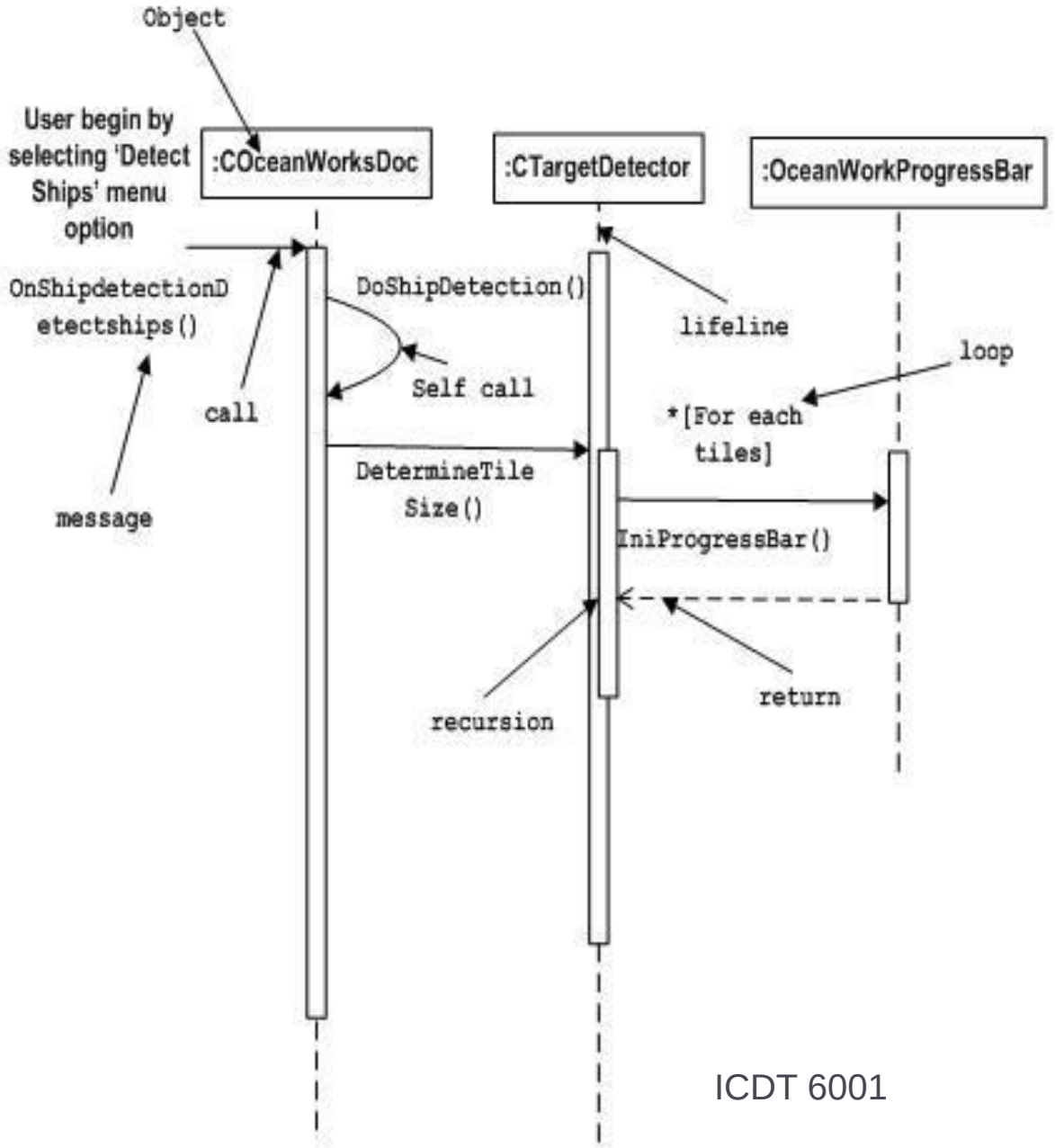
  We can use either kind for our models, NOT BOTH.

Sequence Diagram

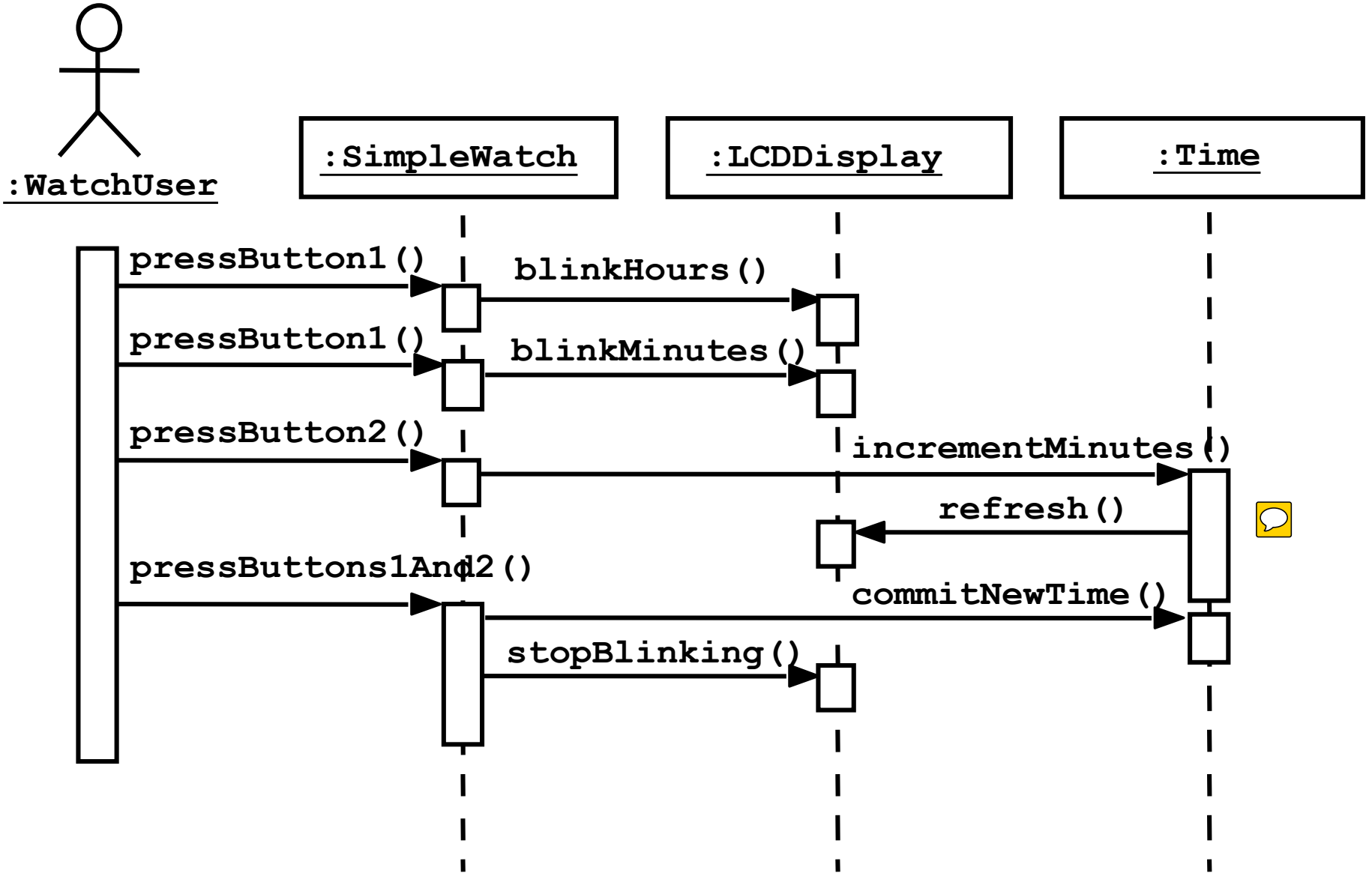
- A sequence diagram has two dimensions: the vertical dimension represents the passage of time, the horizontal dimension represents different objects involved in the interaction.

Notation	Description
Square with Text	Interacting object; text tells name of object
Actor Icon	Actor interacting with objects
Vertical line	Object's lifetime
Solid Arrow	Invocation of operation/service
Vertical Thin Rectangle	Activation period; time during which an object is active
Dotted Arrow	Return message after operation is complete
Star	Operation invoked iteratively

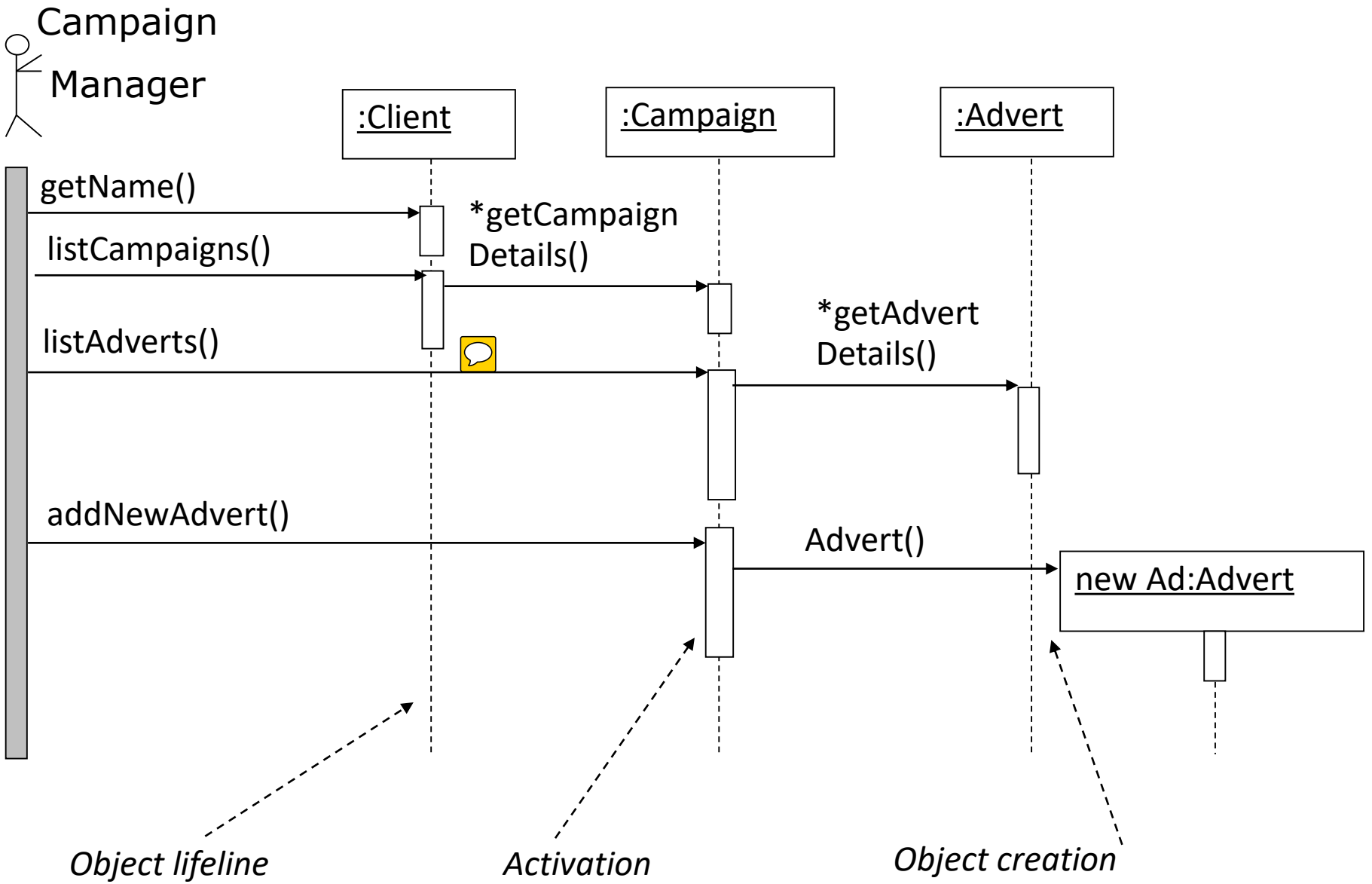
Sequence Diagram



Sequence Diagram



Sequence Diagram



Sequence Diagram

- When a message is sent to an object, it invokes a method/operation of that object. Once a message is received, the operation that it invokes starts to execute. The period of time during which an operation executes is known as an ***activation*** and is shown as a rectangular block along the lifeline.

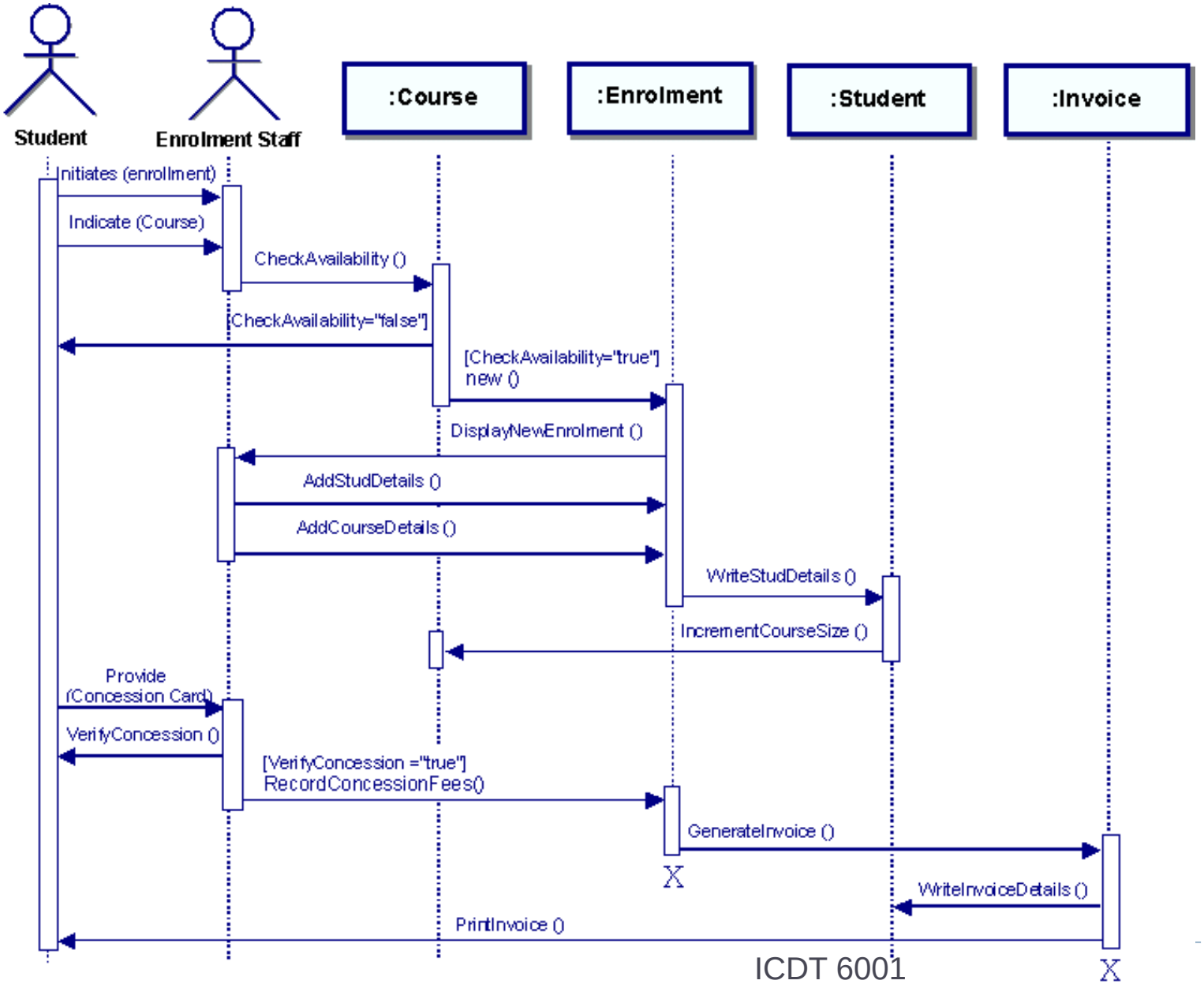
- The ***activation period*** of an operation includes any delay while the operation while the operation waits for a response from another operation that was invoked as part of its execution.

- A '*' before the message name indicates a loop or iteration. The conditions for stopping or continuing the loop may be shown beside the message name.

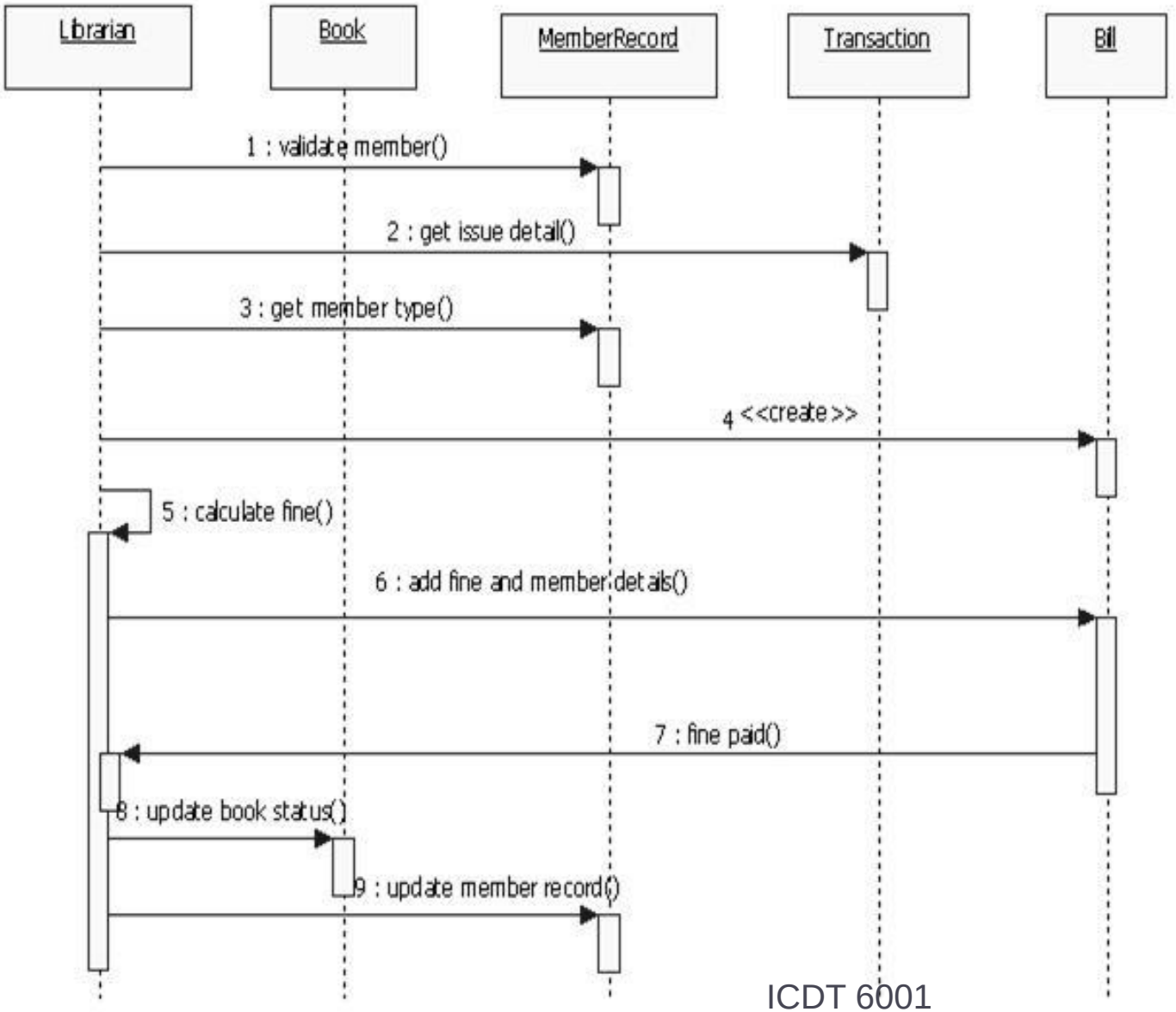
  e.g. **[For all client's campaigns] *getCampaignDetails()**

- An object can send a message to itself. This is known as a **reflexive message** and is shown by a message arrow that starts and ends at same object lifeline.

Sequence Diagram



Sequence Diagram



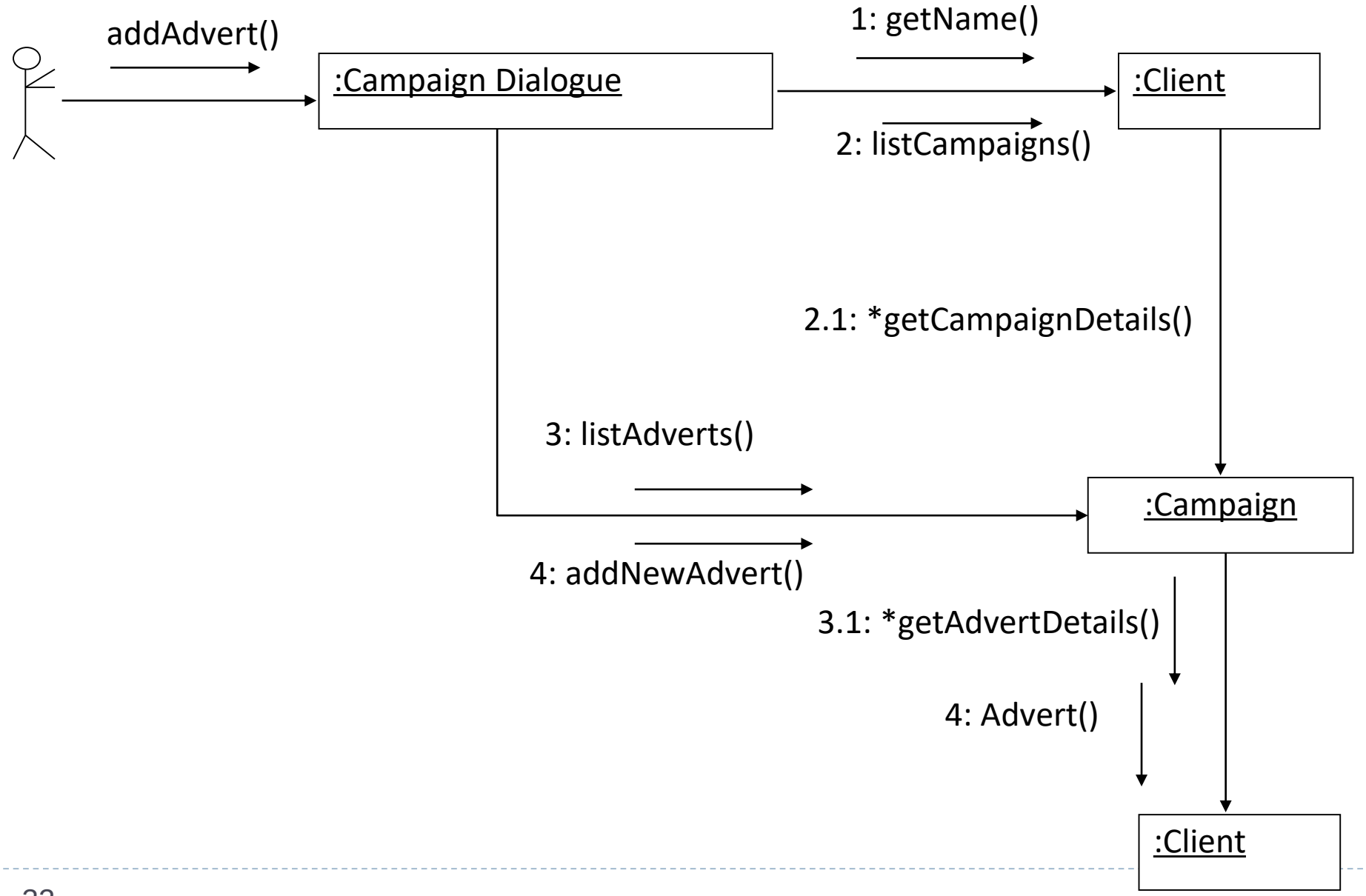
Collaboration Diagram

- Collaboration diagrams provide an alternative way for showing interactions between objects. They have many similarities with sequence diagrams in that they express the same information about object interactions for a use-case but in a different format.

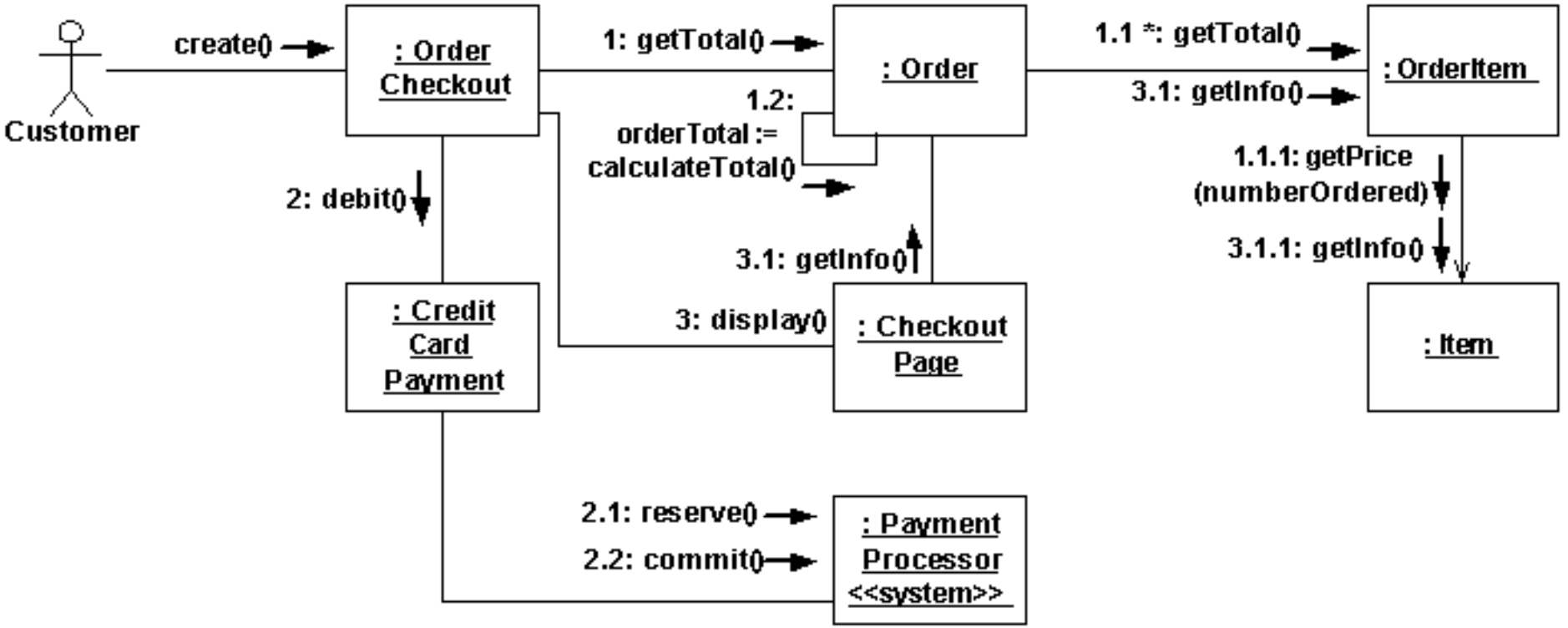
- A collaboration diagram represents a collaboration, which is a set of objects related in a particular context, and integration, which is a set of messages exchanged among the objects within the collaboration to achieve a desired outcome. In a collaboration diagram, each object is represented by an object icon, and links are used to indicate communication paths on which messages are transmitted.

- Unlike sequence diagrams, there is no explicit time dimension. Since the diagram has no time dimension, the order in which messages are exchanged is represented by sequence numbers. Some sequence numbers are written in nested style (e.g. 2.1, 3.1) to indicate iterative loops within the interaction.

Collaboration Diagram



Collaboration Diagram



State Transition Diagram

- While interactions diagrams helps identify the methods of classes, the state transition diagram depicts the sequence/order in which the method invocations take place in response to events. In other words, state chart diagrams show how an entity responds to various events by changing from one state to another and by triggering actions.

- Definitions – state, event, action, transition:

  A state in an interval of time during which an object holds a certain value (or performs a certain operation). For example, we say that the ATM machine is in the state of waiting for the user to enter a card. All the time the system is waiting, the objects of the system do not change values.

  An event is something that happens at a point in time, *such as user presses left button*. It *generally* causes an action/s to take place and the system to change state.

State Transition Diagram

Events can be

external – e.g. sensor detects card entry

internal – e.g. cardcheck indicates invalidity of card

single –e.g. card_valid

compound – e.g. card_valid && pin_ok

A transition is a change from one state to another as a result of an event. For example, when the sensor detects card entry, the system moves to another state where the card validity is being checked.

An action is a response from the system as a result of an event. For example, when the sensor detects card entry, the system launches the procedure of checking the validity of the card.

In brief, when an event occurs, a transition is fired and the actions associated with it are executed.

State Transition Diagram

Initial state denoted by  (solid filled circle)
The initial state is a notational convenience and an object does not remain in its initial state but must immediately move into another named state.

Final state denoted by  (solid circle within another)
An object cannot leave its final state once it has been entered.

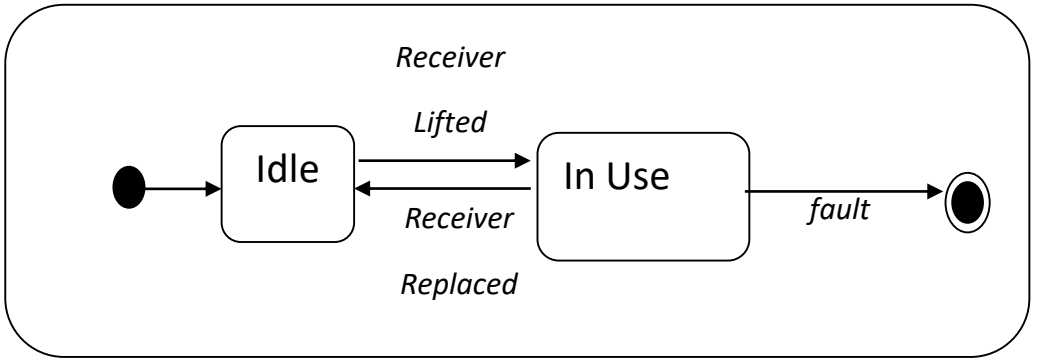
All other states are shown as rounded rectangles and labeled with a meaningful name.

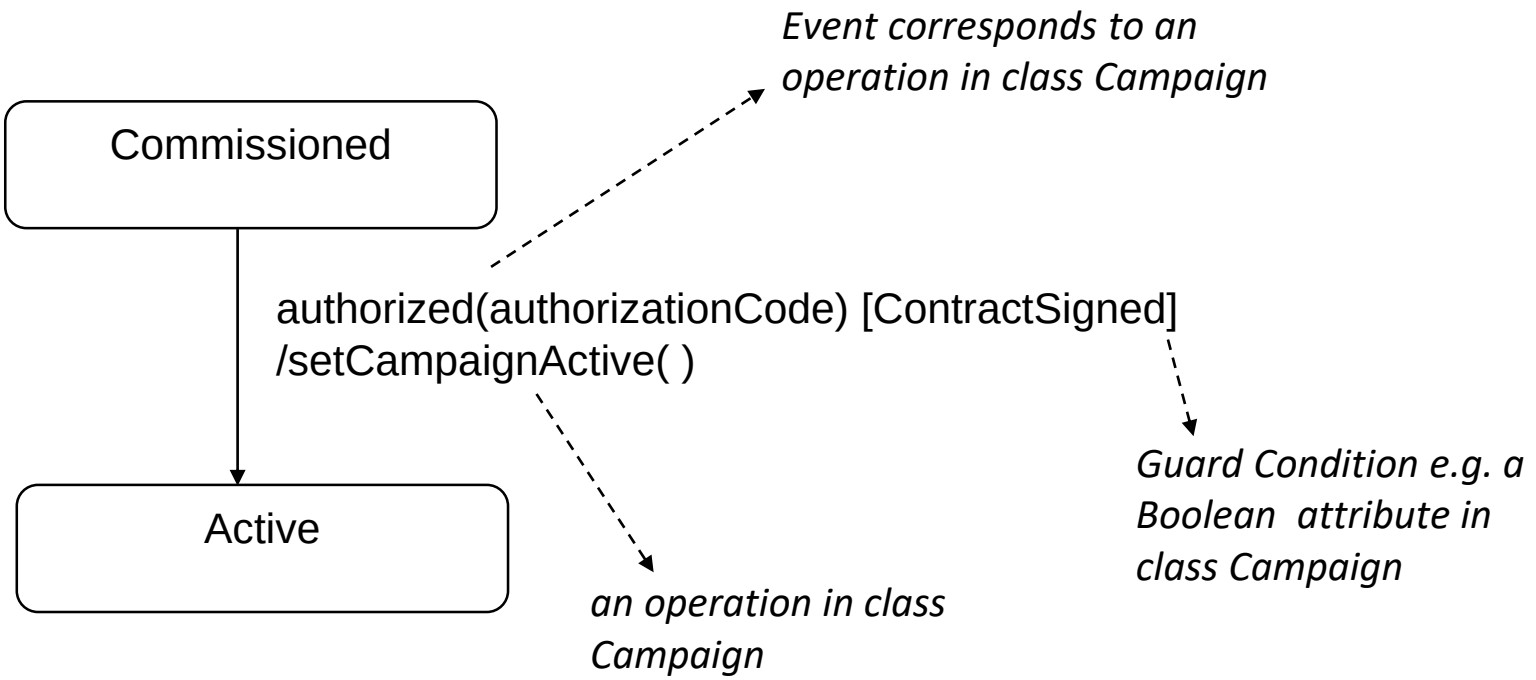
A transition is indicated by an arrow and is labeled with a transition string to indicate the event that triggers it:
[*event_name*(*parameter_list*)] [*guard_condition*] [*action_expression*]

[*guard_condition*] – Boolean expression that is evaluated at the time the event fires. Contained in square brackets. It can be a function involving parameters and also the attributes and operations of the object owning the state chart but returning a Boolean value.

[*action_expression*] – An action that is executed when an event triggers the transition. It may involve parameters of the triggering event and also the attributes and operations of the object owning the state chart. Begins with a " **/** ".

State Transition Diagram





State Transition Diagram

- So far, action-expressions have been associated with transitions. It is also useful to model internal actions that are triggered by events that do not change the state (e.g on entering a state, or exiting a state).

- Actions are listed in the internal transitions compartment of the current state.

- 2 kinds of internal events:

  entry, indicating actions to be taken on entering the state, and

  exit, indicating actions to be taken on leaving the state.

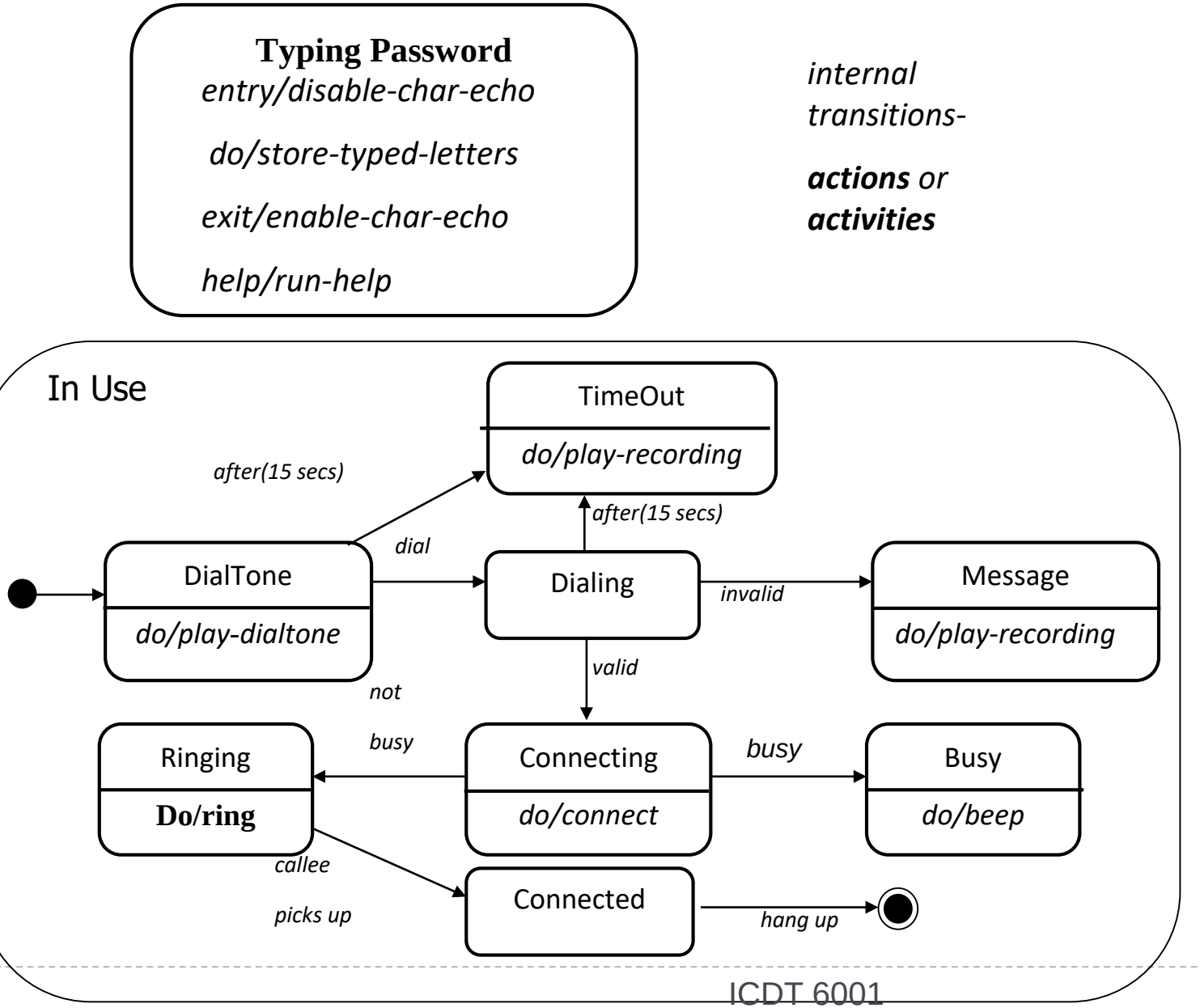
  do , keyword used to execute a specific activity

  NOTE: Actions are short-lived i.e occur within the time period of the state whereas activities can exceed the time the object is in the state which caused the activity to be triggered.

- The words *do*, *entry* and *exit* are the permissible UML keywords;

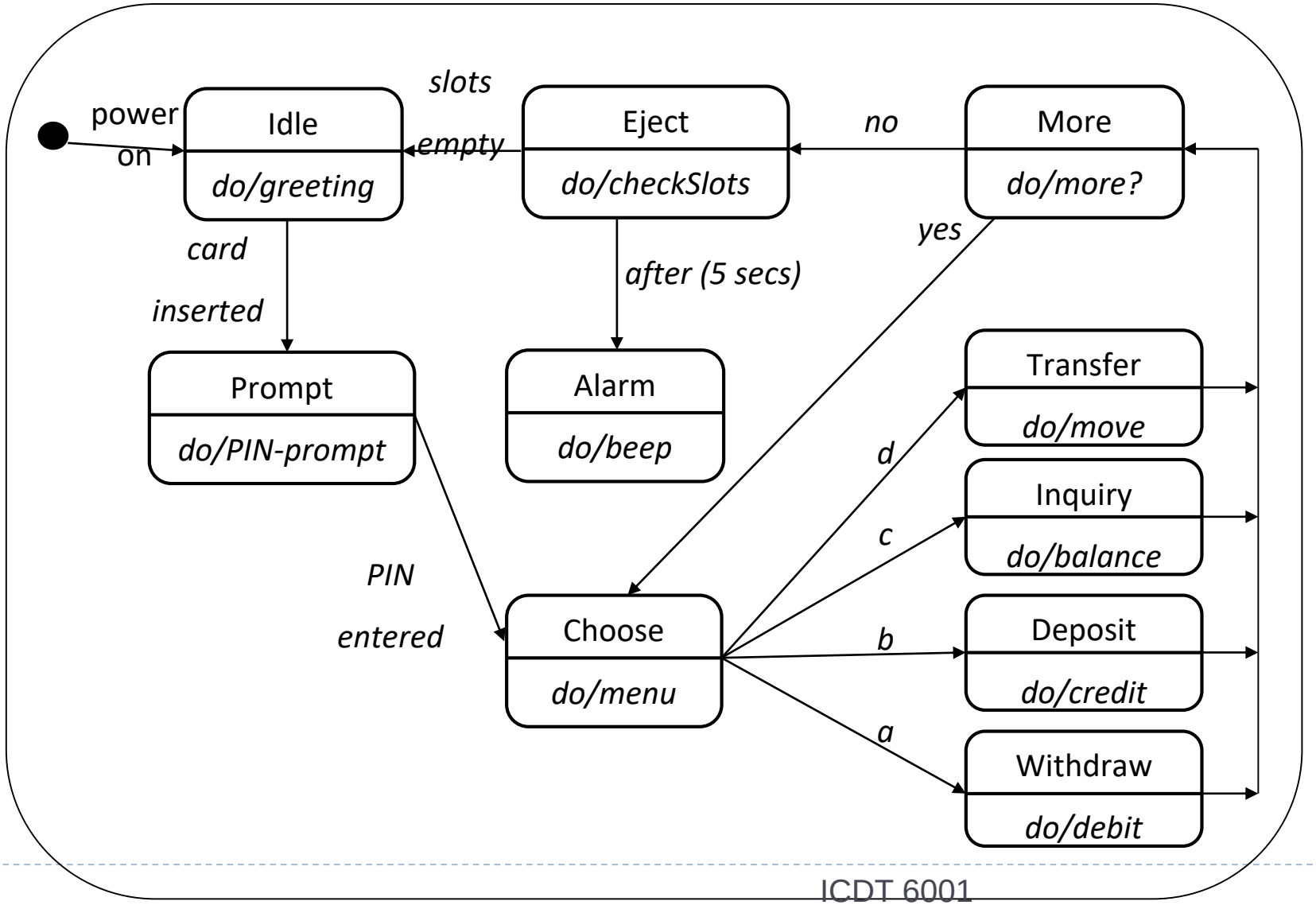
  UML also permits *arbitrary events* to be listed within the state, along with the action to be taken when they occur:

State Transition Diagram

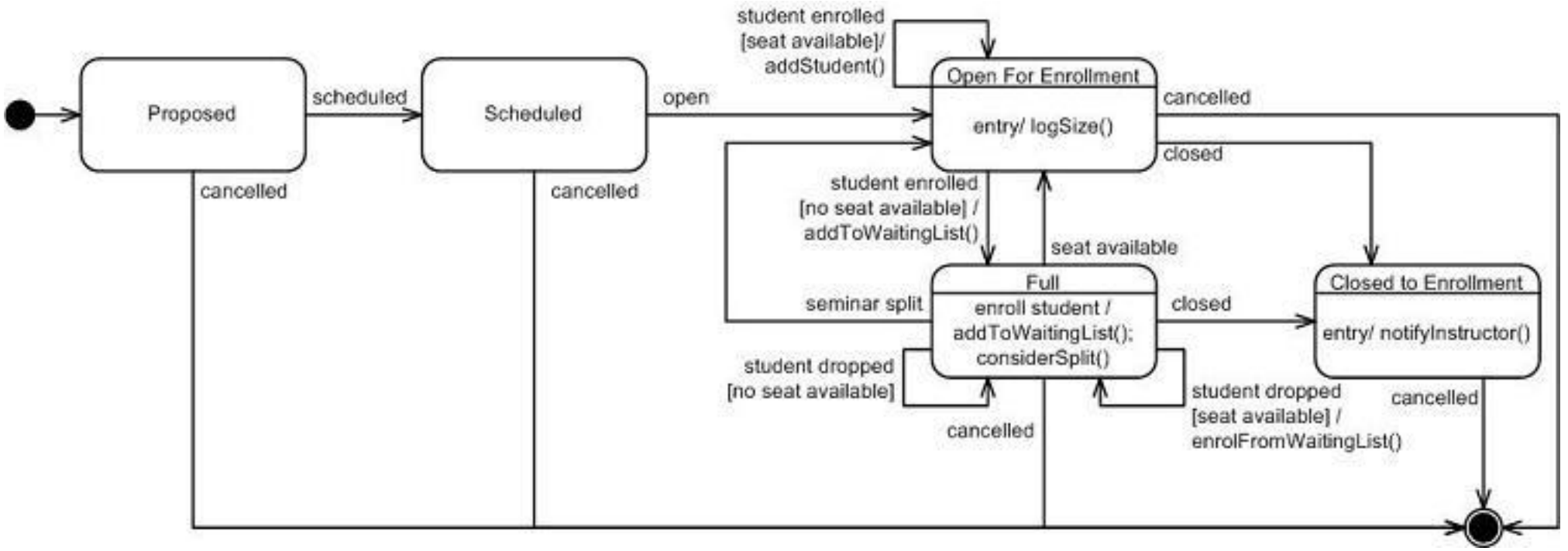


State Transition Diagram

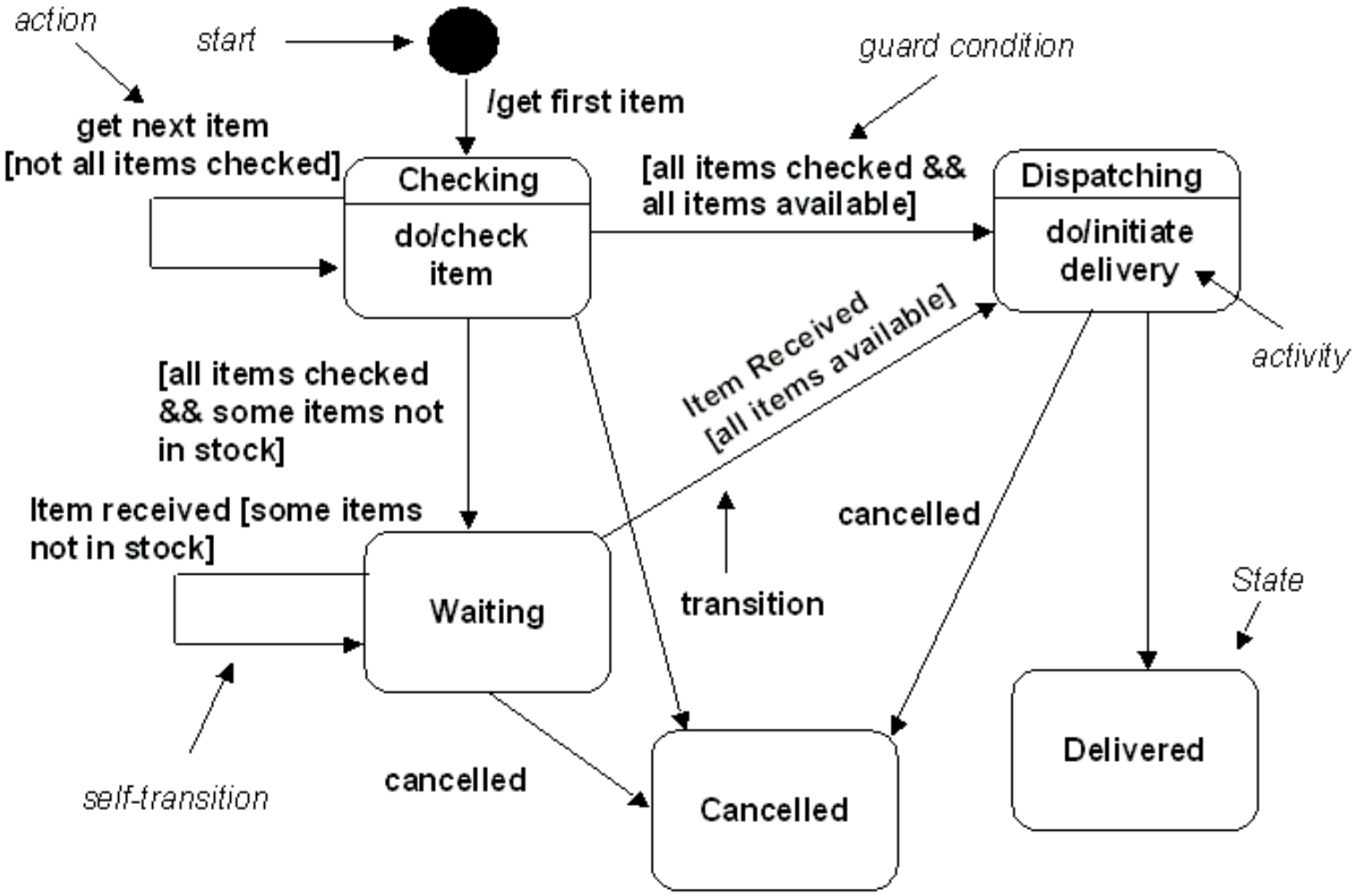
STD for a 4-function (deposit, withdrawal, balance, transfer) banking machine



State Transition Diagram



State Transition Diagram



State Transition Diagram