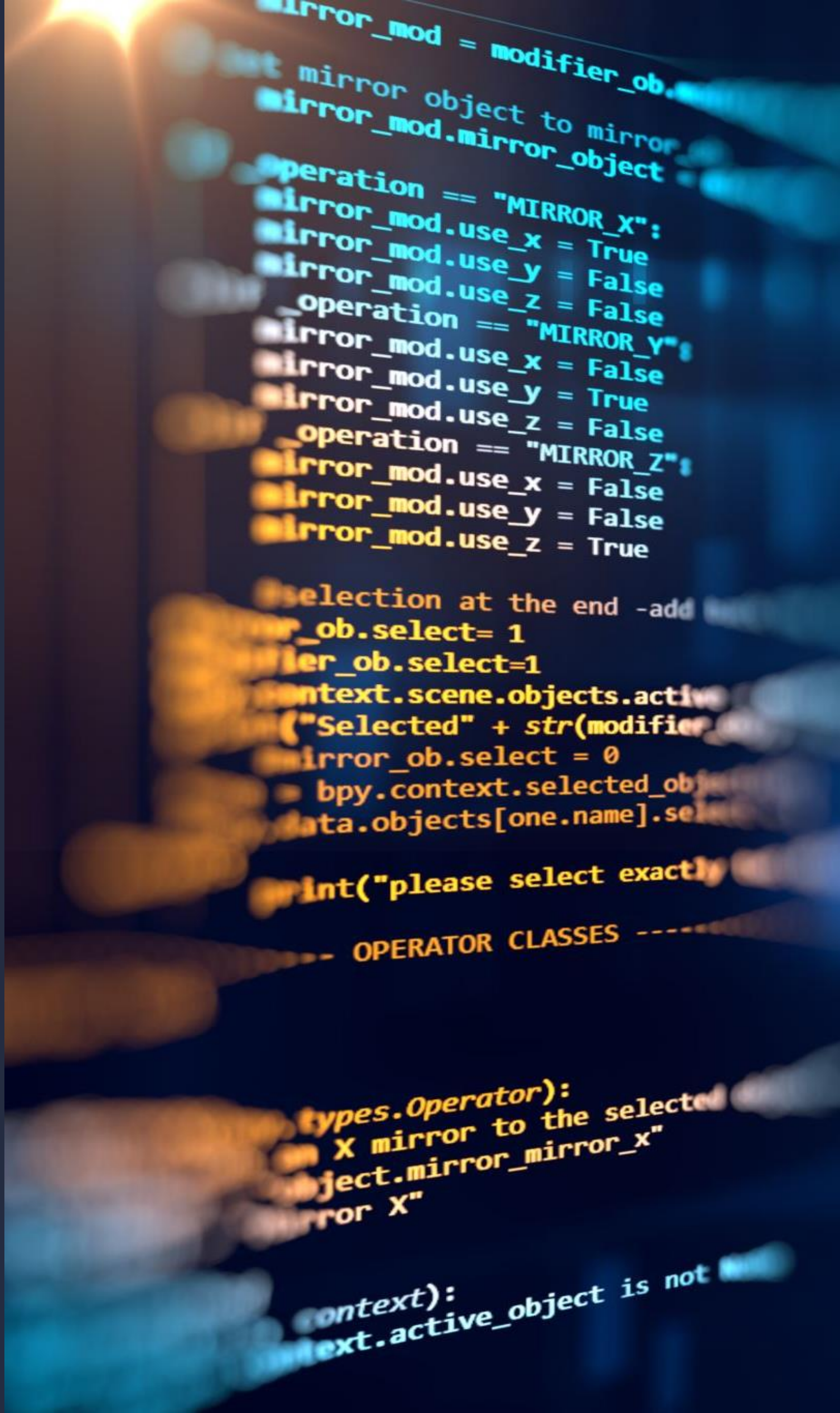
CSE216: SOFTWARE ENTERPRISE: PERSONAL PROCESSES AND QUALITY

Assist. Prof.

Dr. Noha El-Sayad



WEEK 1

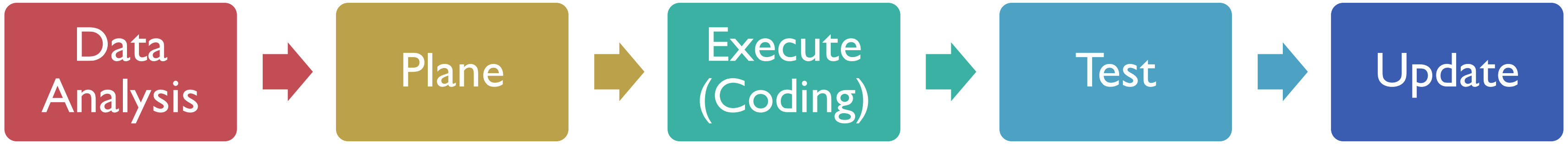
- Introduction To Software Engineering
- UML

Software Engineering: the systematic application of scientific and technological knowledge, methods, and experience to the design, implementation, testing, and documentation of software

A Problem Solving Activity

- *Analysis*: Understand the nature of the problem and break the problem into pieces
- *Synthesis*: Put the pieces together into a large structure
- *Techniques (method):* formal procedures for producing results using some well-defined notation
- *Methodologies*: collection of techniques applied across software development and unified by a philosophical approach
- *Tools:* instrument or automated systems to accomplish a technique

A Problem Solving Activity (Life Cycle)



Data Analysis → Plane → Execute (Coding) → Test → Update

Why are software systems so complex?

1. The problem domain is difficult
2. The development process is very difficult to manage
3. Software offers extreme flexibility

Three Ways Dealing with Complexity

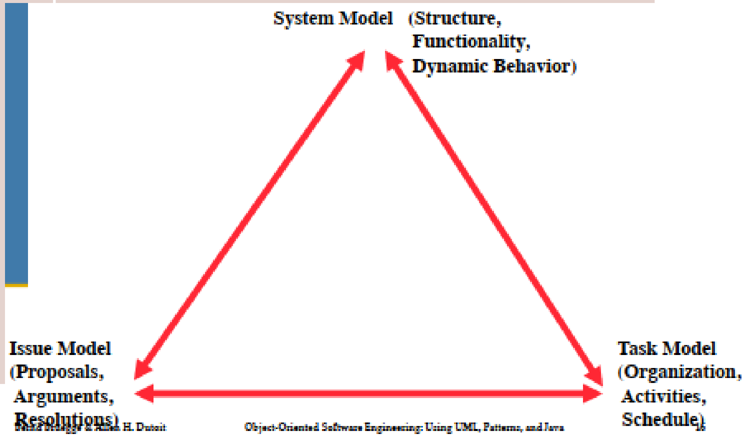
1. Abstraction
2. Decomposition
3. Hierarchy

1. Abstraction

- Inherent human limitation to deal with complexity
- Chunking: Group collection of objects
- Ignore unessential details => Models

Models are used to provide abstractions

System Model:	***Task Model***:	***Issues Model***:
- *Object Model*: • What is the structure of the system? • What are the objects and how are they related? - *Functional model*: • What are the functions of the system? • How is data flowing through the system? - *Dynamic model*: • How does the system react to external events? • How is the event flow in the system ?	- *PERT* (Program Evaluation & Review Technique) Chart: • What are the dependencies between the tasks? - *Schedule*: • How can this be done within the time limit? - *Org Chart*: • What are the roles in the project or organization?	- What are the open and closed issues? - What constraints were posed by the client? - What resolutions were made?


System Model (Structure, Functionality, Dynamic Behavior)
Issue Model (Proposals, Arguments, Resolutions)
Task Model (Organization, Activities, Schedule)
Bernd Bruegge & Allen H. Dutoit
Object-Oriented Software Engineering: Using UML, Patterns, and Java

1. Decomposition

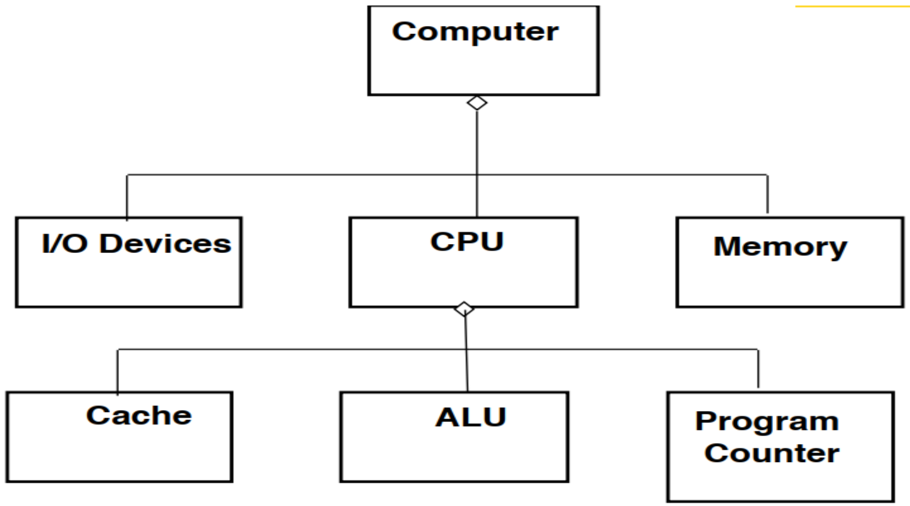
- A technique used to master complexity (“divide & conquer”)
- Decomposition have different two methods

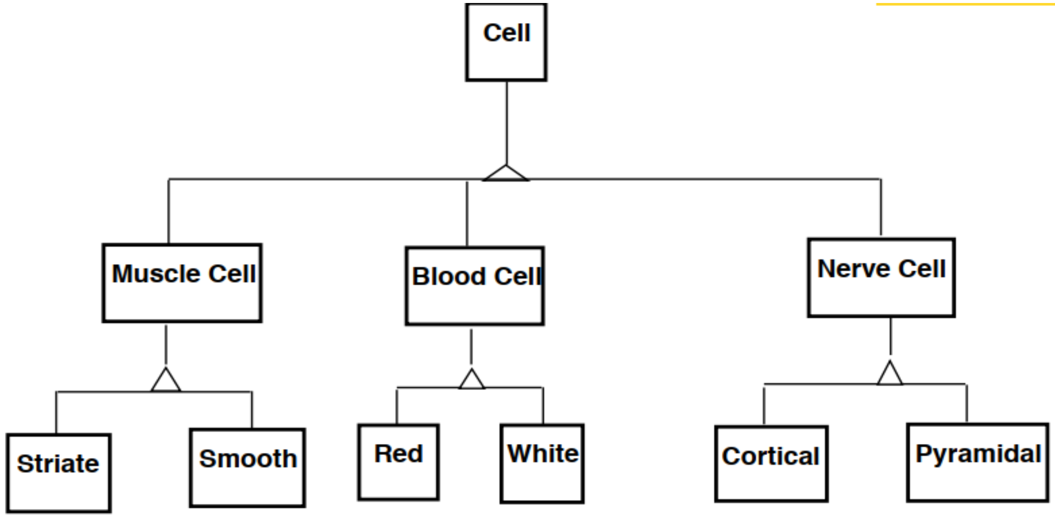
1. **Functional decomposition**
   - The system is decomposed into modules
   - Each module is a major processing step (function) in the application domain
   - Modules can be decomposed into smaller modules
2. **Object-oriented decomposition**
   - The system is decomposed into classes (“objects”)
   - Each class is a major abstraction in the application domain
   - Classes can be decomposed into smaller classes

3. Hierarchy

- We got abstractions and decomposition
  - This leads us to chunks (classes, objects) which we view with object model
- Another way to deal with complexity is to provide simple relationships between the chunks
- One of the most important relationships is hierarchy
- There are 2 important hierarchies

1. "Part of" hierarchy
2. "Is-kind-of" hierarchy





Modeling with UML

A *system* is an organized set of communicating parts

1. Natural system: A system whose ultimate purpose is may not be known

   Examples of natural systems: Universe, earth, ocean
2. Engineered system: A system which is designed and built by engineers for a specific purpose

   Examples of engineered systems: Airplane, watch, GPS

A *subsystems* The parts of the system can be considered as systems again

Examples of subsystems: Jet engine, battery, satellite

A *model* is an abstraction describing a system or a subsystem

A *view* depicts selected aspects of a model

A *notation* is a set of graphical or textual rules for depicting models and views

UML

Use case diagrams: Describe the functional behavior of the system as seen by the user

Class diagrams: Describe the static structure of the system: Objects, attributes, associations

Sequence diagrams: Describe the dynamic behavior between objects of the system

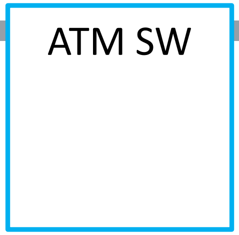
State chart diagrams: Describe the dynamic behavior of an individual object

Activity diagrams: Describe the dynamic behavior of a system, in particular the workflow.

1. (Use case diagrams)

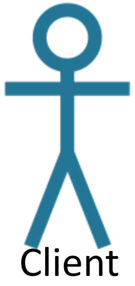
❑ **Elements**

1) System: the software to be developed.
Represented as a rectangle with a name label at the top.

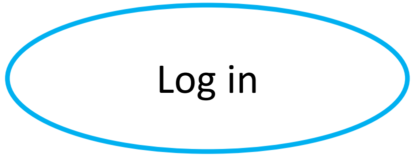


2) Actors: people or other external systems that interact with the system.
Represented as a stick figure with a name label.

1. Primary actors: who initiate actions appear to the left of the system
2. Secondary actors: who react to actions appear on the right of the system
3. All actors should be outside the system

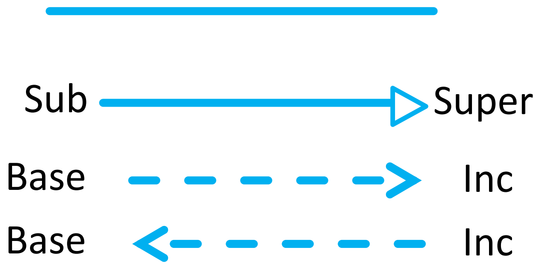


3) Use cases: all possible operations the system can support users with.
Represented as ovals with name label inside.

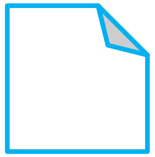


4) Relationships: links that define how actors and cases are related to each other .
Represented as:

1. Solid line no arrows: actor to case
2. Sloid line with empty triangle head: actor to actor or case to case
3. Dashed-line with an angle arrow head: two types:
    1. Include: base case must run included case always:
    2. Extends: base case runs included case if conditions are detected:

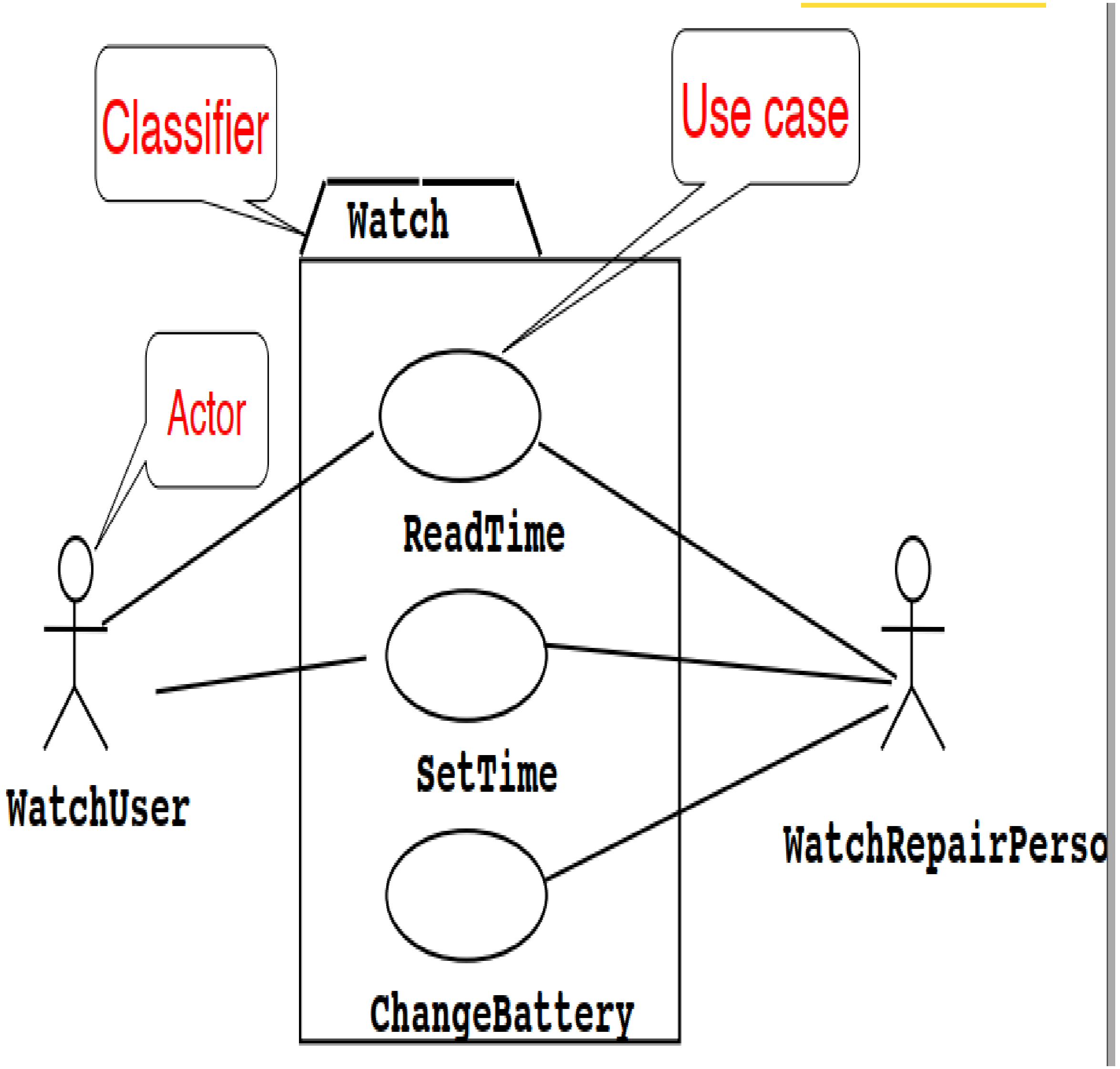


5) Notes: annotations includes in a document shape.
E.g. can define conditions for some include case.

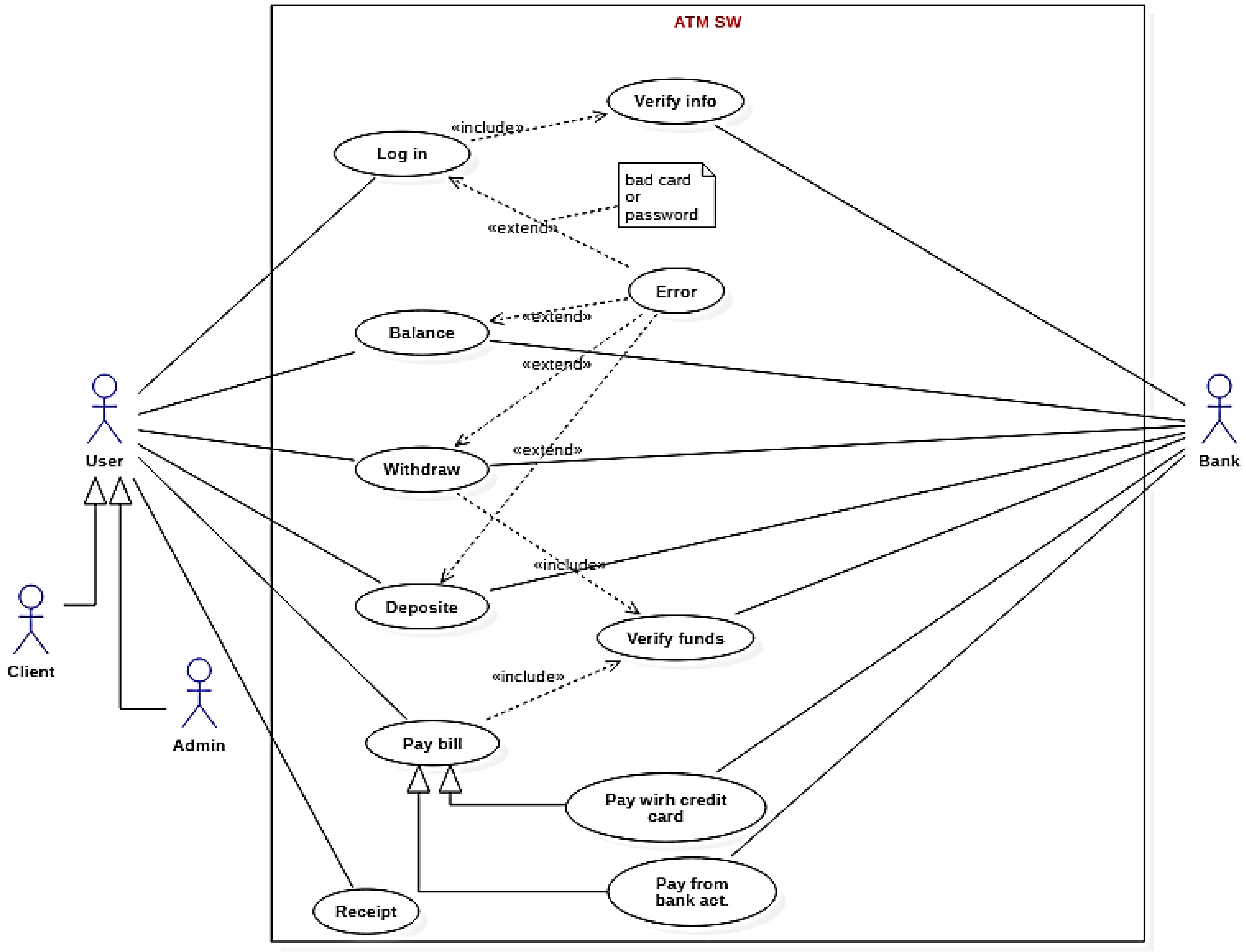


Exercise: Use Case Diagram

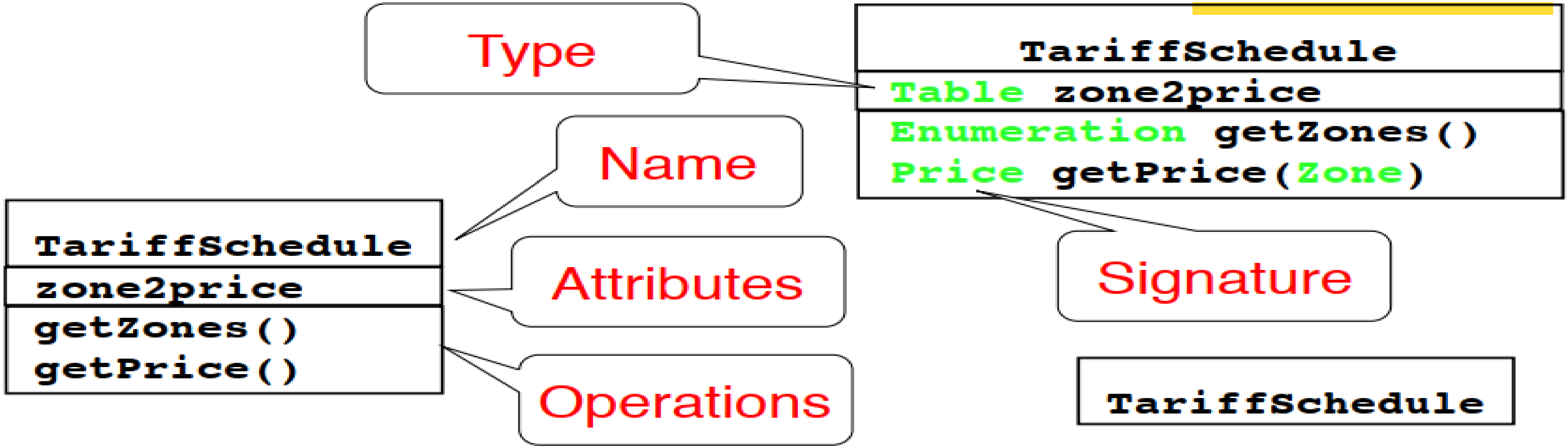
- Draw a Use Case diagram for a SimpleWatch. A user of the watch can read time and set time. The battery of the watch can be changed by a repair person.



Exercise 2


ATM SW
Verify info
Log in
«include»
bad card or password
«extend»
Error
Balance
«extend»
«extend»
Withdraw
«extend»
«include»
Deposite
Verify funds
«include»
Pay bill
Pay wirh credit card
Pay from bank act.
Receipt
User
Client
Admin
Bank

2. (Use Class diagrams)



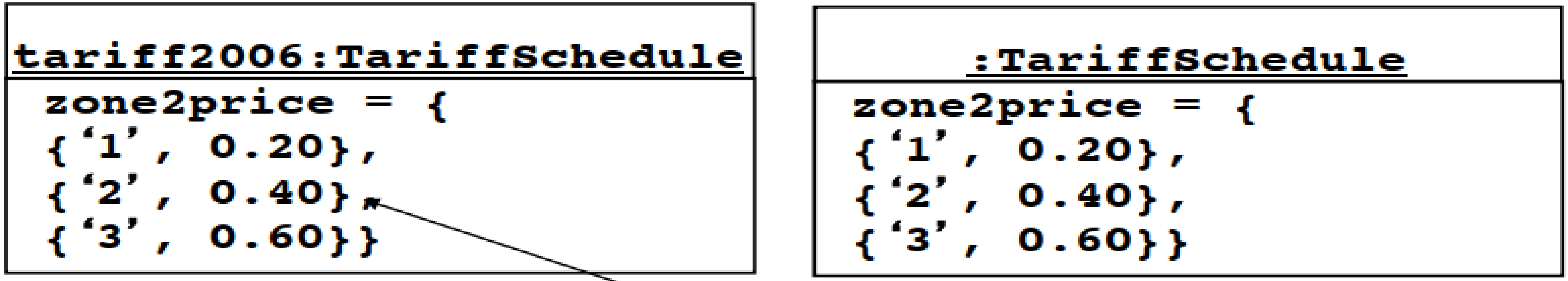
- A *class* represents a concept
- A class encapsulates state ***(attributes)*** and behavior ***(operations)***

Each attribute has a ***type***

Each operation has a ***signature***

The class name is the only mandatory information

Instances



<u>tariff2006:TariffSchedule</u>

```
zone2price = {
{'1', 0.20},
{'2', 0.40},
{'3', 0.60}}
```

<u>:TariffSchedule</u>

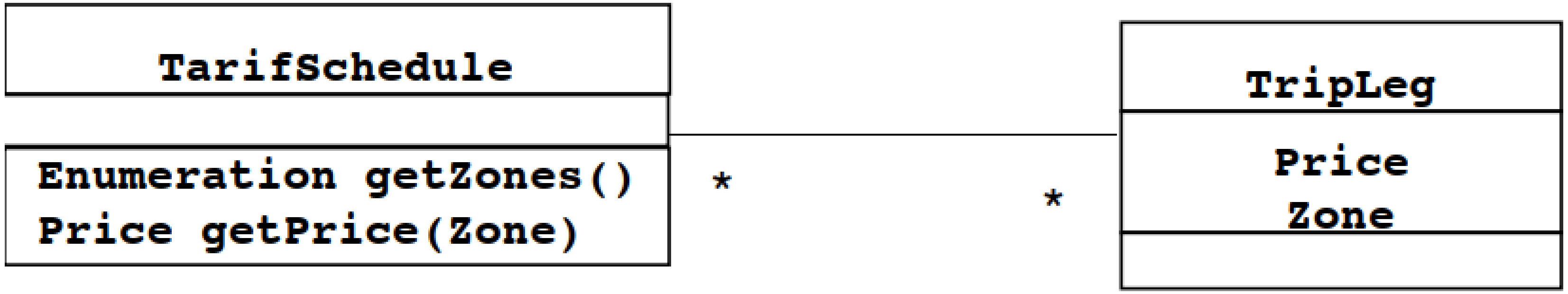
```
zone2price = {
{'1', 0.20},
{'2', 0.40},
{'3', 0.60}}
```

- An ***instance*** represents a phenomenon
- The attributes are represented with their ***values***
- The name of an instance is <u>underlined</u>
- The name can contain only the class name of the instance (anonymous instance)

Class vs. Object

- **Class**
  - An abstraction modeling an entity in the application or solution domain
  - The class is part of the system model (“Passenger”, “Ticket distributor”, “Server”, “TariffSchedule”)
- **Object**
  - A specific instance of a class (“Joe, the passenger who is purchasing a ticket from the ticket distributor”).

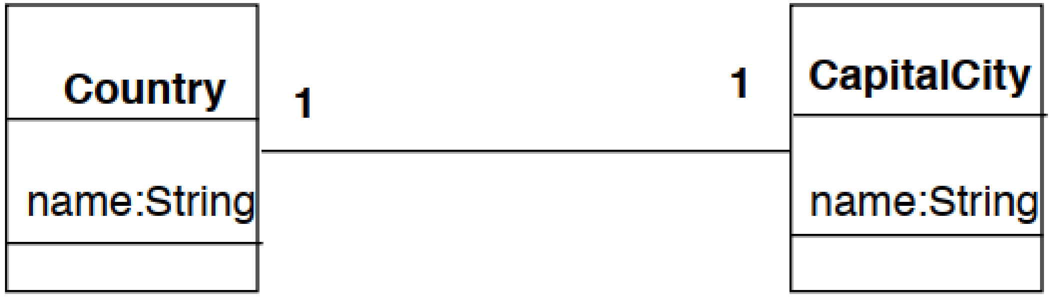
Associations



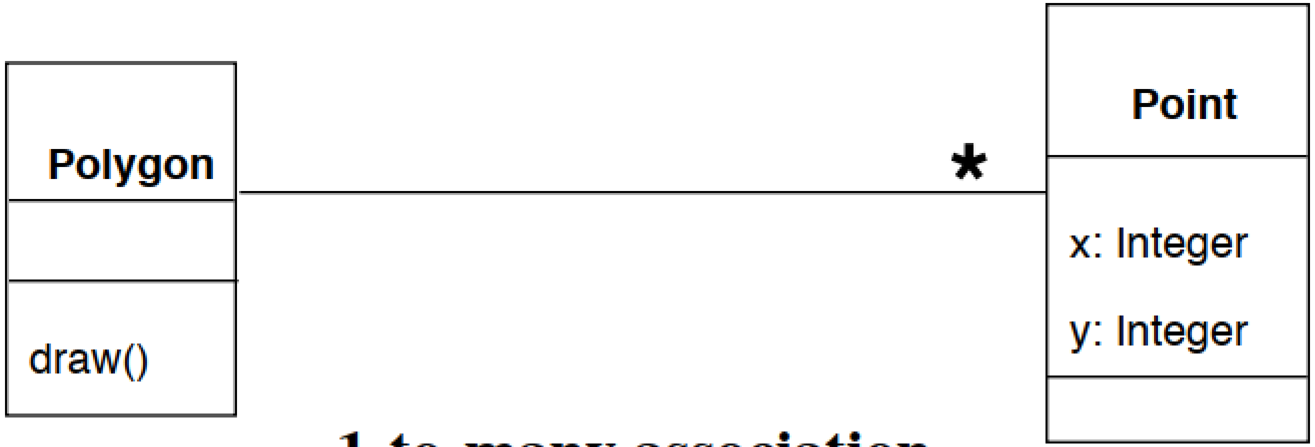
Associations denote relationships between classes.

The multiplicity of an association end denotes how many objects the instance of a class can legitimately reference.

1-to-1 and 1-to-many Associations

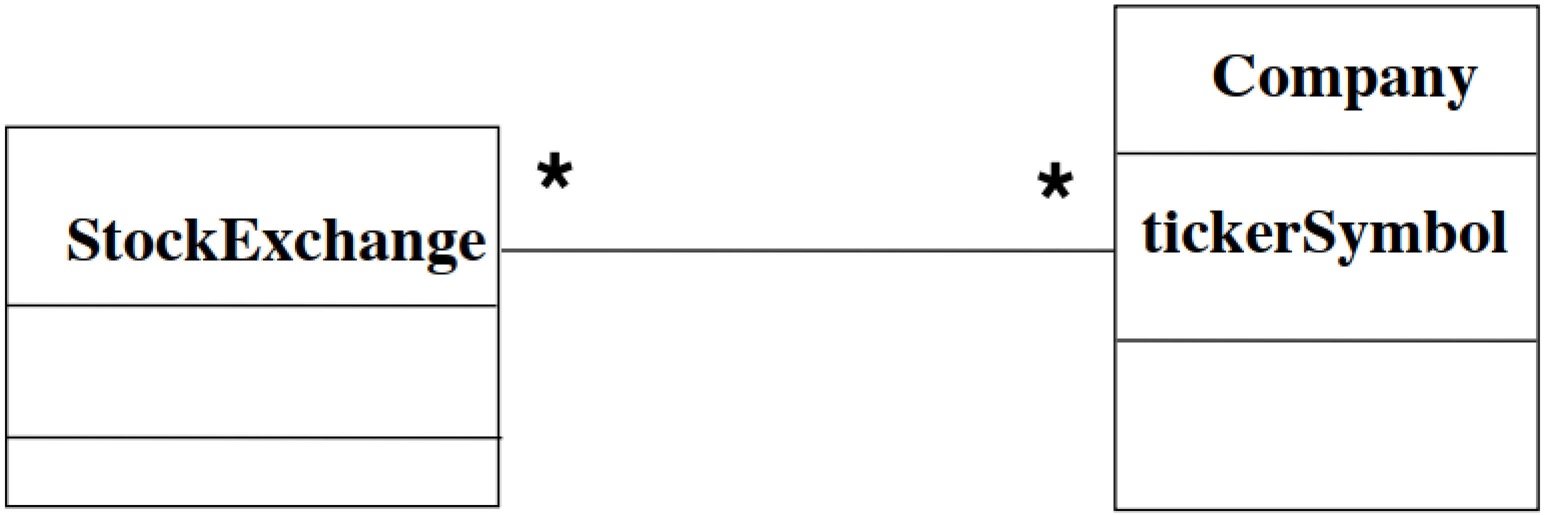


1-to-1 association



1-to-many association

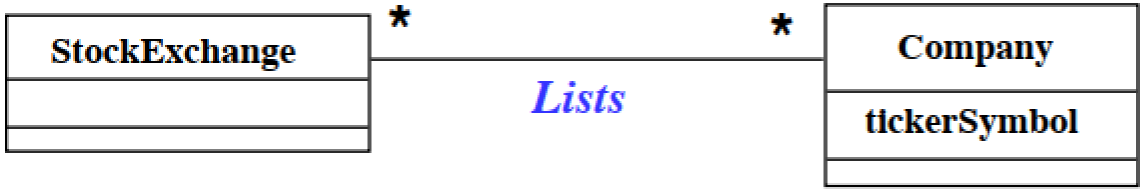
Many-to-Many Associations



Model-Driven Software Development

Reality: A stock exchange lists many companies. Each company is identified by a ticker symbol

Analysis results in analysis object model (UML Class Diagram):

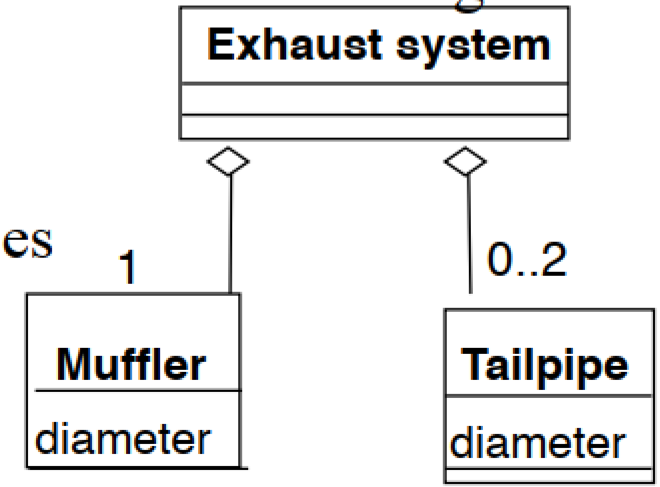


Implementation results in source code (Java):

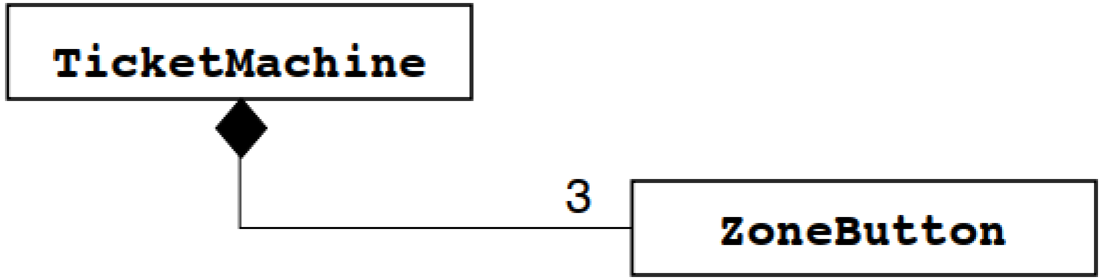
```
public class StockExchange {
    public Vector m_Company = new Vector();
 };
public class Company  {
    public int m_tickerSymbol;
    public Vector m_StockExchange = new Vector();
};
```

Aggregation

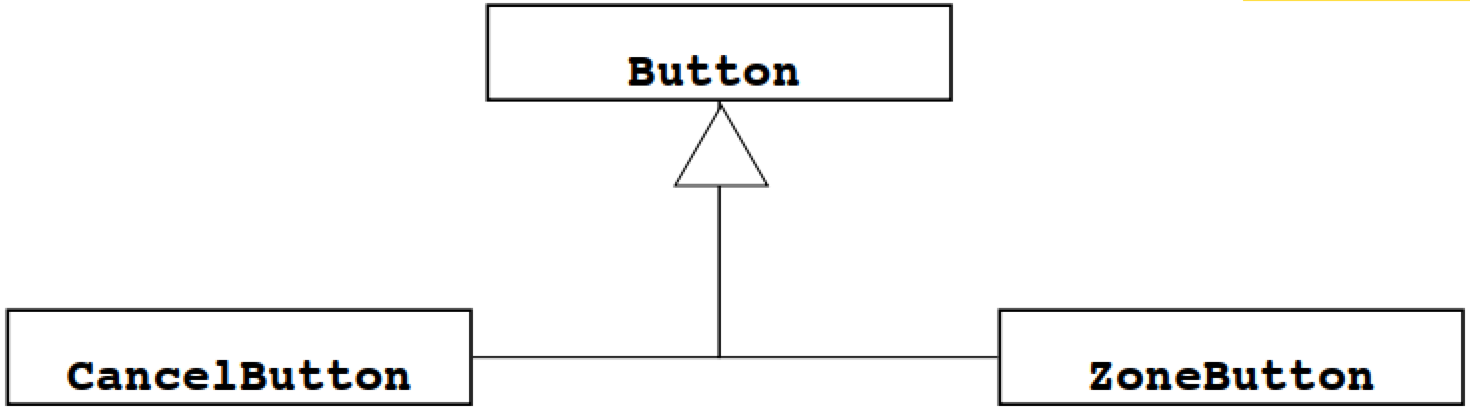
- An *aggregation* is a special case of association denoting a "consists-of" hierarchy
- The *aggregate* is the parent class, the components are the children classes



A solid diamond denotes *composition*: A strong form of aggregation where the *life time of the component instances* is controlled by the aggregate. That is, the parts don't exist on their own ("the whole controls/destroys the parts")



Inheritance

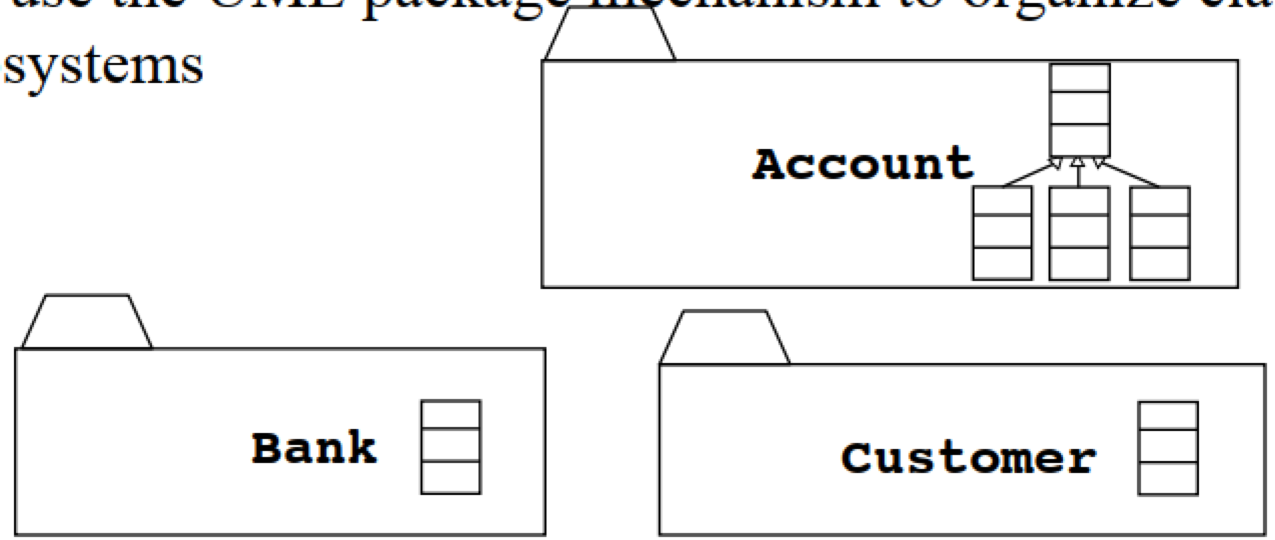


- *Inheritance* is another special case of an association denoting a “kind-of” hierarchy
- Inheritance simplifies the analysis model by introducing a taxonomy
- The **children classes** inherit the attributes and operations of the **parent class.**

Packages

Packages help you to organize UML models to increase their readability

We can use the UML package mechanism to organize classes into subsystems

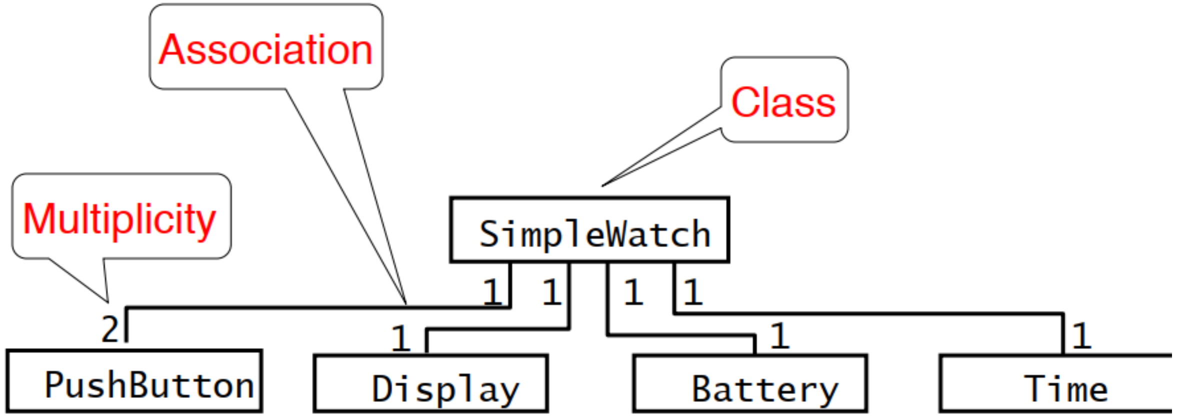


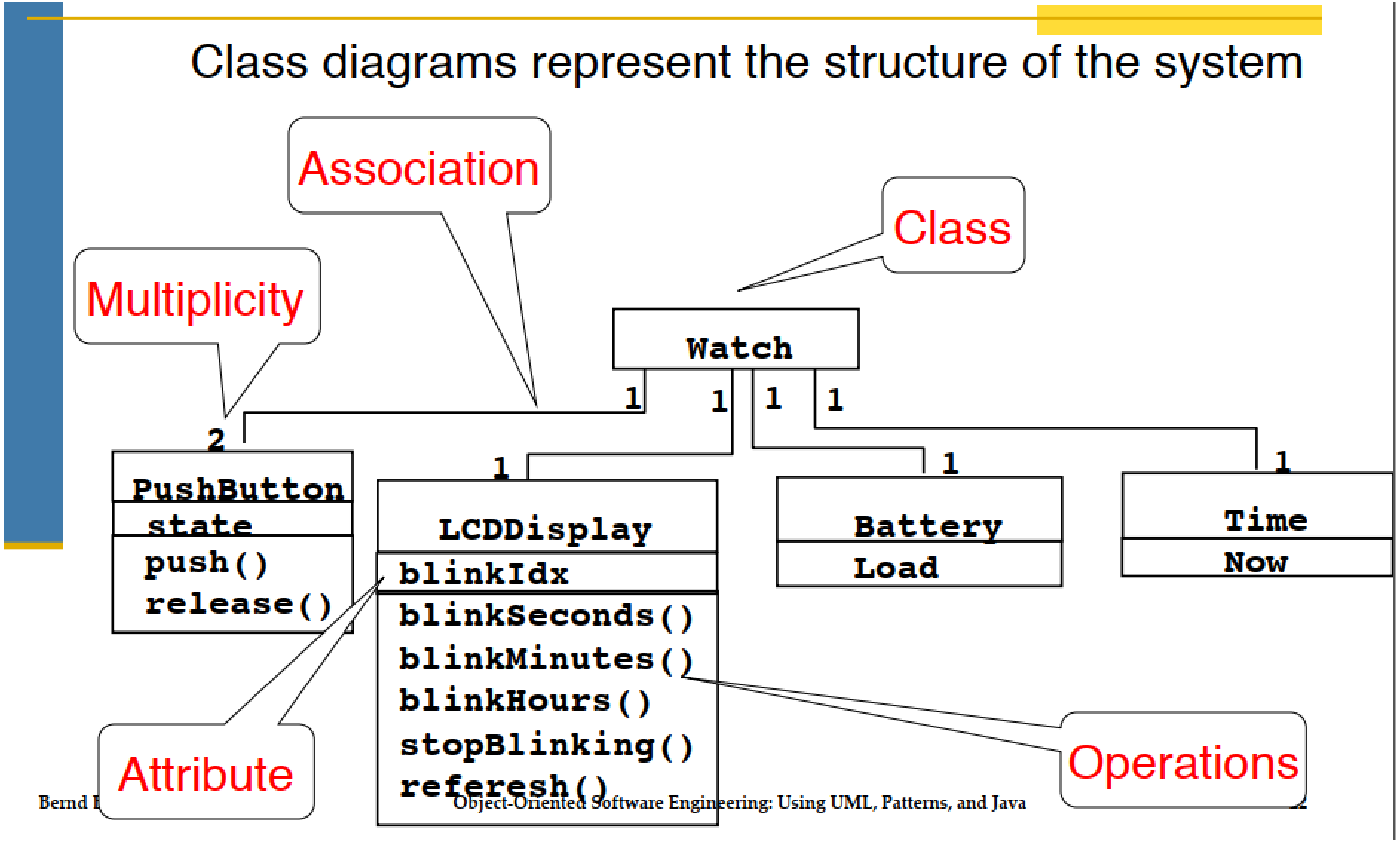
Any complex system can be decomposed into subsystems, where each subsystem is modeled as a package.

Exercise: Class diagrams

- Draw a class diagram for a SimpleWatch that has 2 push buttons to set the time, a LCD display to view the time, has a battery, and shows current time.

Exercise: Class diagrams





Class diagrams represent the structure of the system

CLASS STRUCTURE

1) Client – User: sub – super

2) Admin – User: sub – super

3) User – ATM: directed association User is not a structural part of ATM so cannot use aggregation.

4) User – FamilyCards: must exist in User Cannot exist by it self, so composition is used.

5) FamilyCards – Card: card is part of FamilyCards. It is Team – member relation so, aggregation. Card can exist by it self so it is NOT composition relation.

6) Transaction – ATM: transaction Cannot exist outside ATM it is part Of ATM software design so Composition is used.

7) ATM – Account: same as in ATM – User

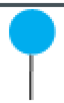
8) ATM, Account – JUnitTestible: Interface realization

9) ATM – Bio: dependency

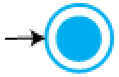
10) BioMetricV – Voice, Iris: nesting (source code inclusion)

11) Account-User: account aggregates one or more users

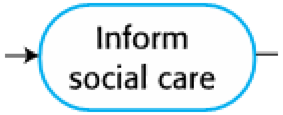
3. (UML Activity diagrams)



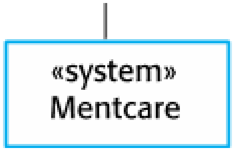
The **start** of a process is indicated by a filled circle



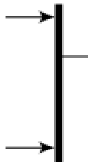
The **end** by a filled circle inside another circle



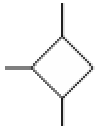
Rectangles with round corners represent **activities**, that is, the specific sub-processes that must be carried out



Systems that are used to support different Sub-processes



Join or Fork: solid bar (synchronous) indicates activity coordination. When the flow from more than one activity leads to a solid bar, they all must be completed before proceeding to the next activity unless, these activities branch off of a decision block (diamond)



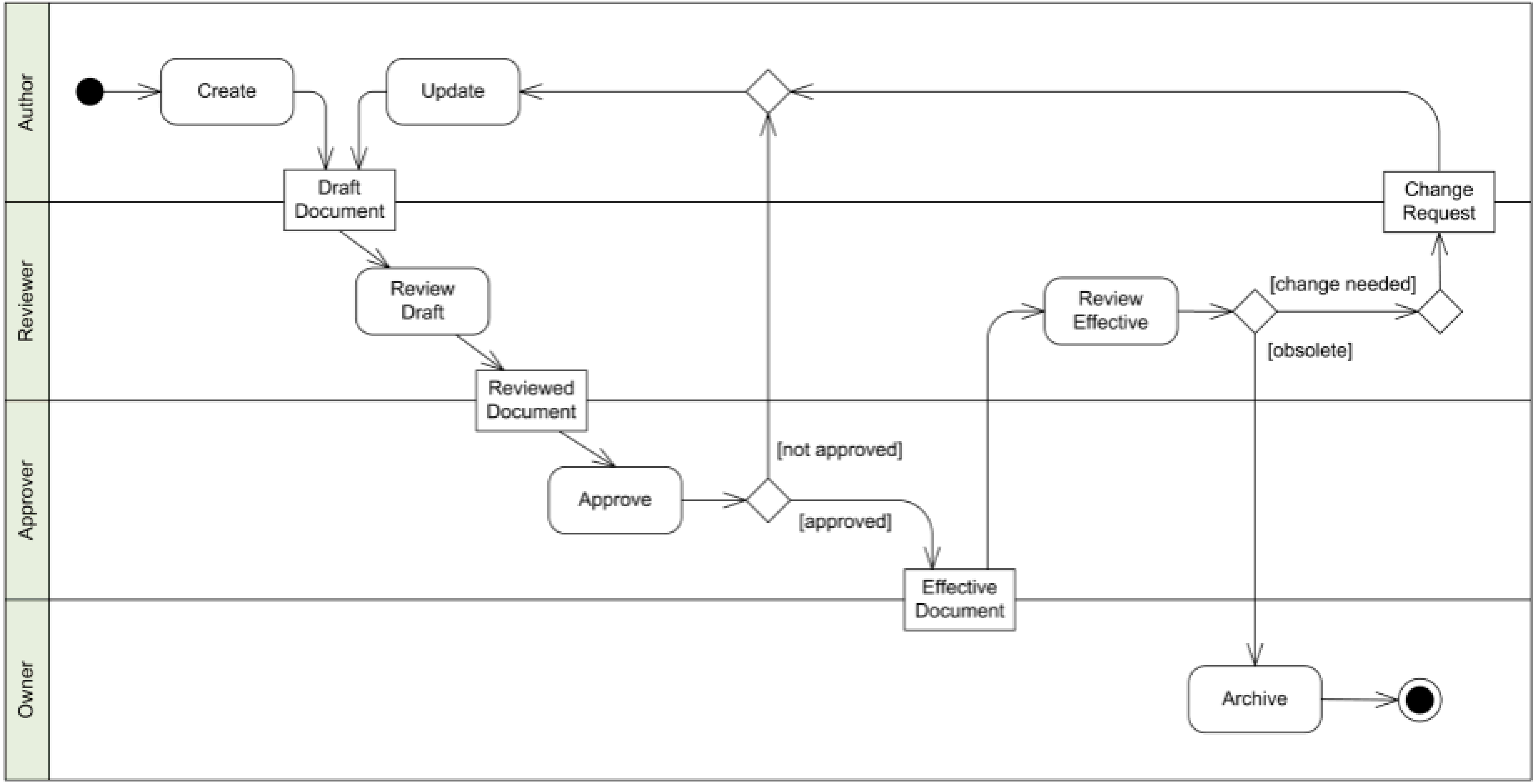
Merge or Branch block (asynchronous): decision, option

[dangerous]

Branch Arrows may be **annotated** with guards (in square brackets) that specify when that flow is followed

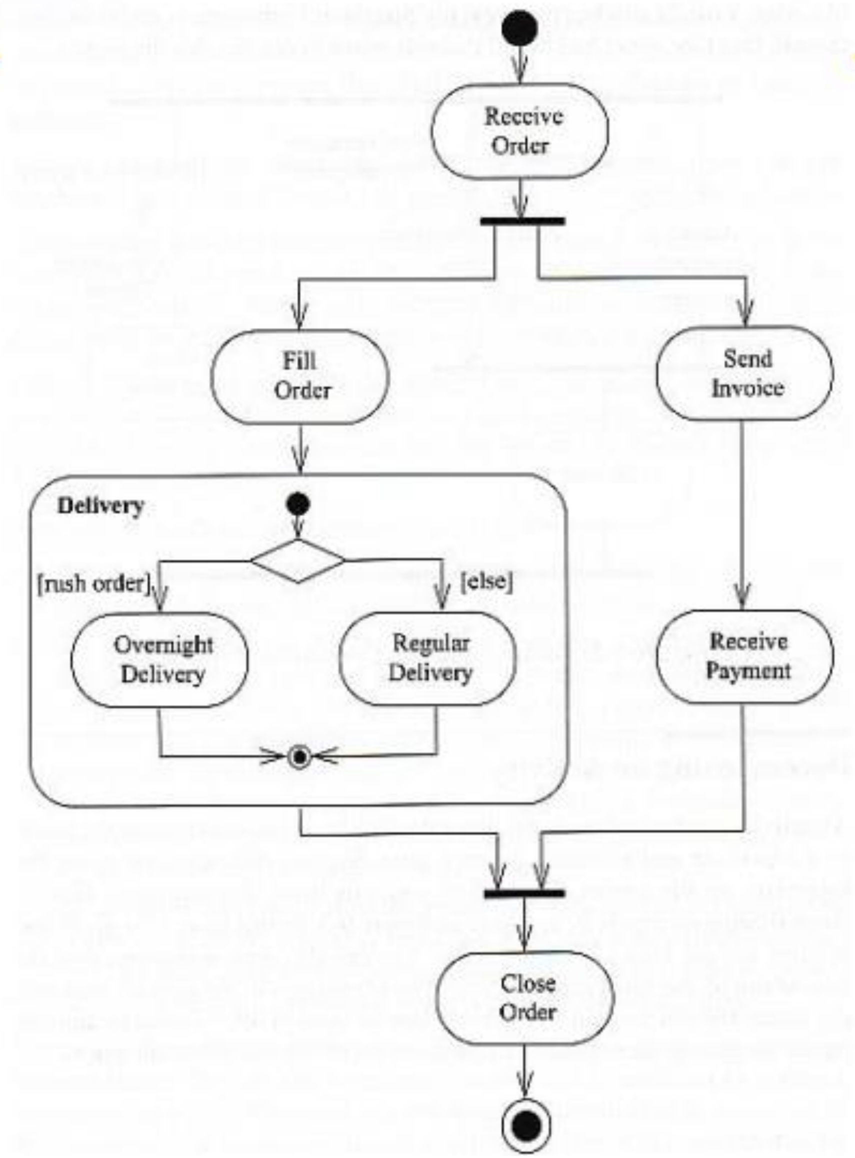
More elements: Activity (box for the entire process), condition (a note for pre, post of an activity), control flow (arrow), flow final node (not end node it is a branch end), expansion region (iterative sub process -- loop), exception handling, interrupt, partition and swim lane (for each object a vertical box to enclose all object related activities)
See all these in the PDF reference.

Activity Diagram – document management process



Exercise: Activity Diagram

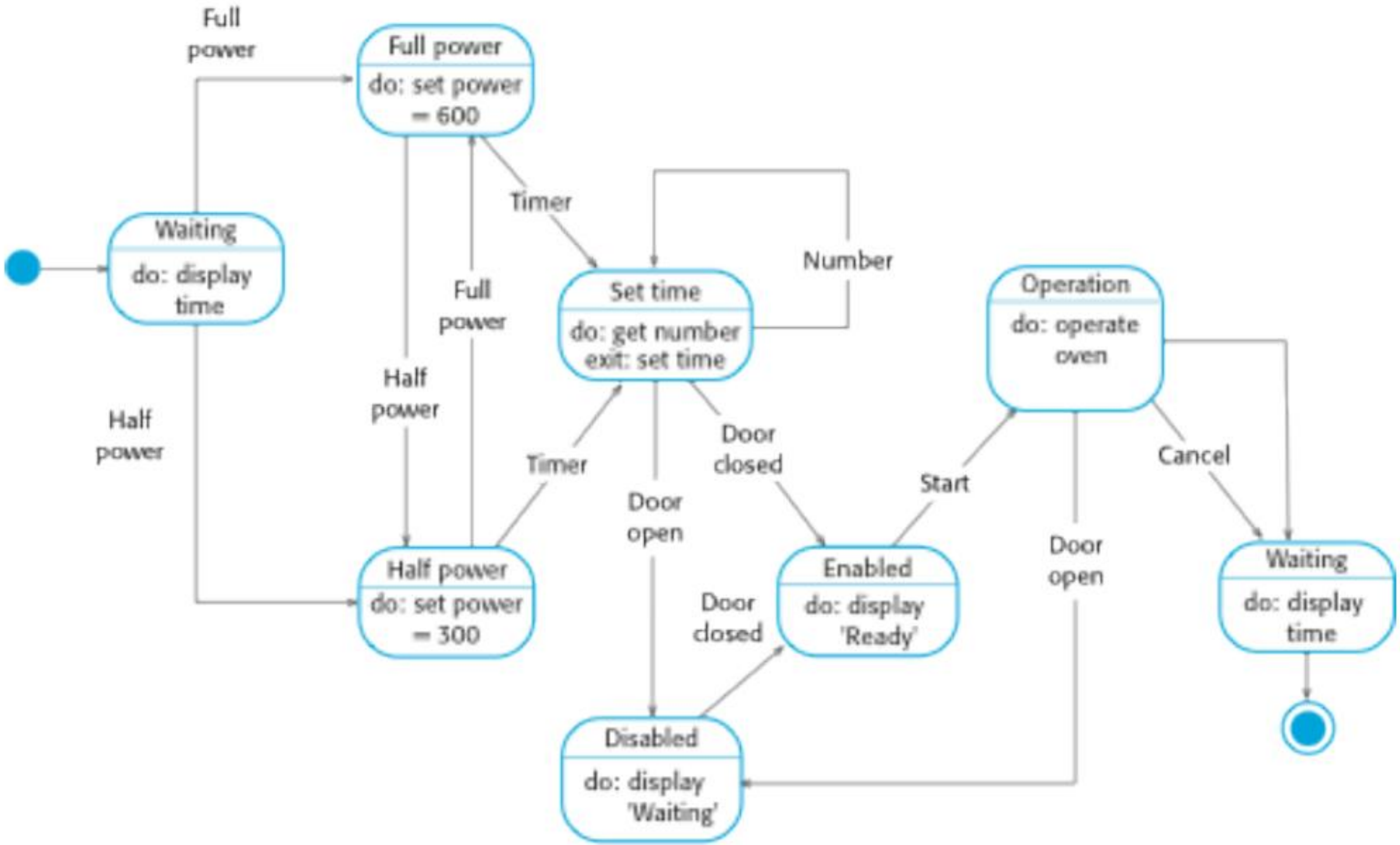
- Draw an activity diagram for fulfillment of an order, (i.e., an order is received, it is processed, invoice is produced, delivered either overnight or regular, and payment is received before the order is closed).



3. (UML - STATE MACHINE)

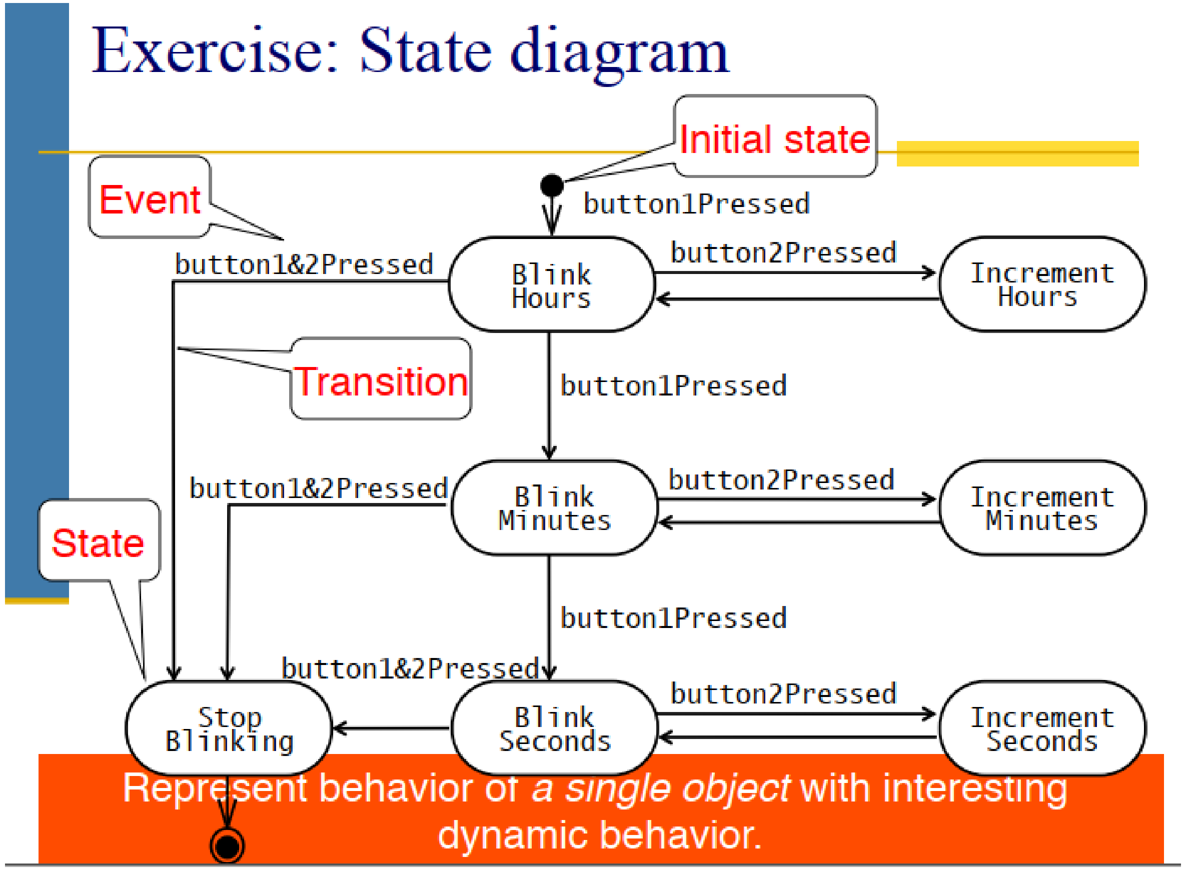
- Model the behaviour of the system in response to external and internal events.
- Show the system’s responses to stimuli so are often used for modelling real-time systems.
- Show system states as nodes and events as arcs between these nodes. When an event occurs, the system moves from one state to another.
- State diagrams are an integral part of the UML and are used to represent state machine models.

State diagram of a microwave oven



Exercise: State Diagram

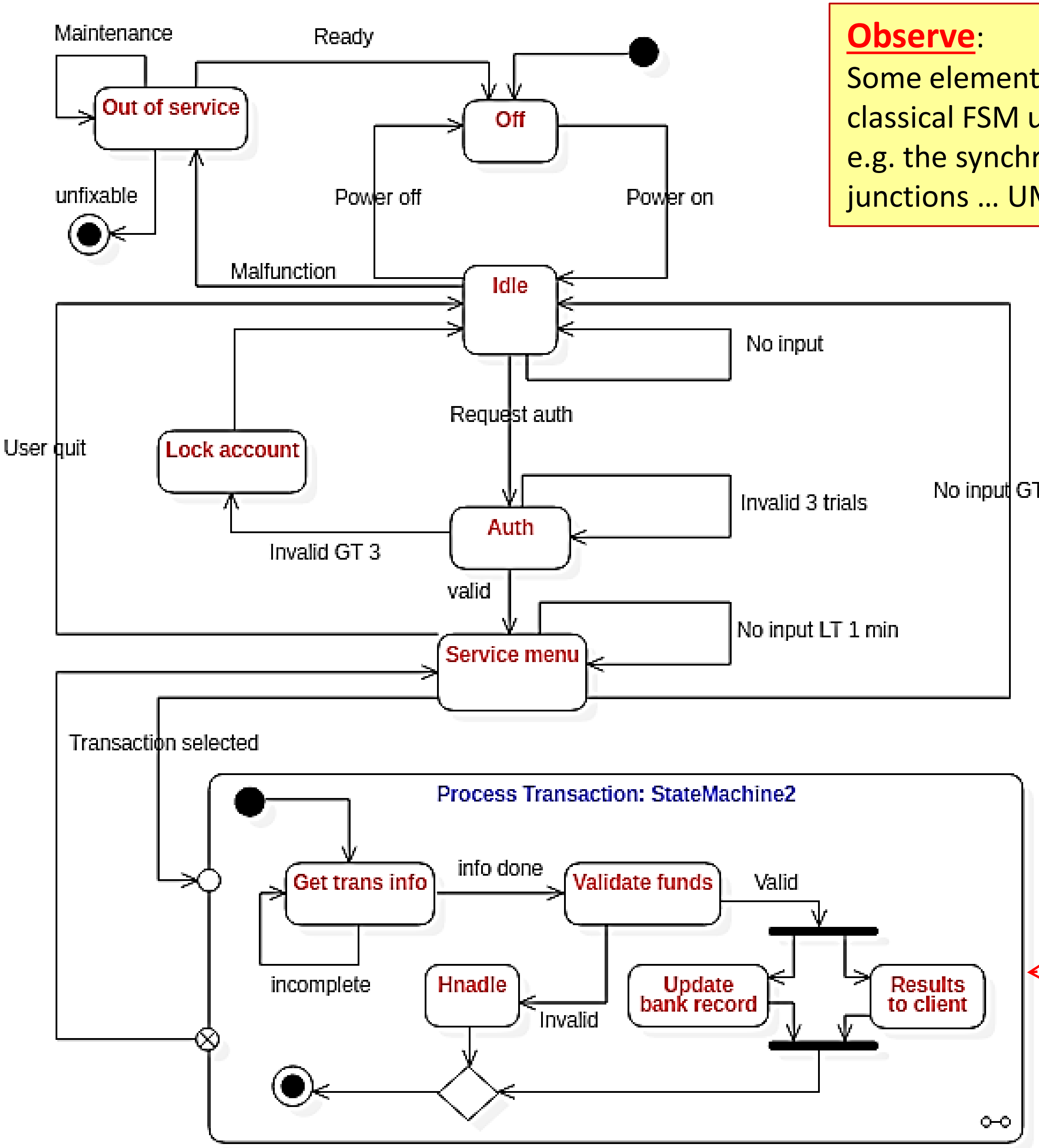
- Draw a state diagram that models the behavior of LCD display of the SimpleWatch.



Exercise: State diagram

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

5) UML – STATE MACHINE

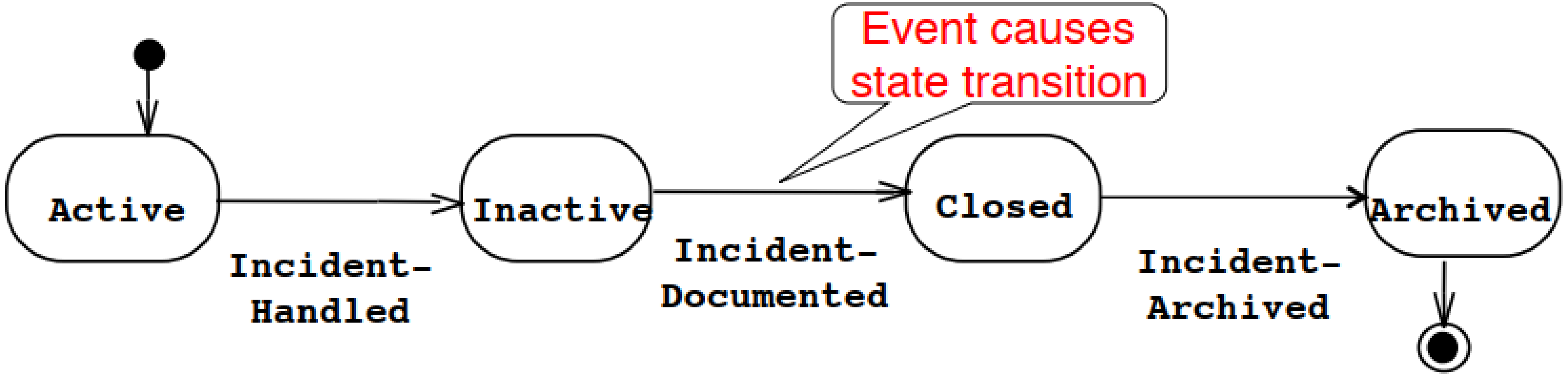


Observe:
Some elements are not standard in classical FSM used here:
e.g. the synchronous/asynchronous junctions … UML allows that.

Activity Diagram vs. Statechart Diagram

Statechart Diagram for Incident

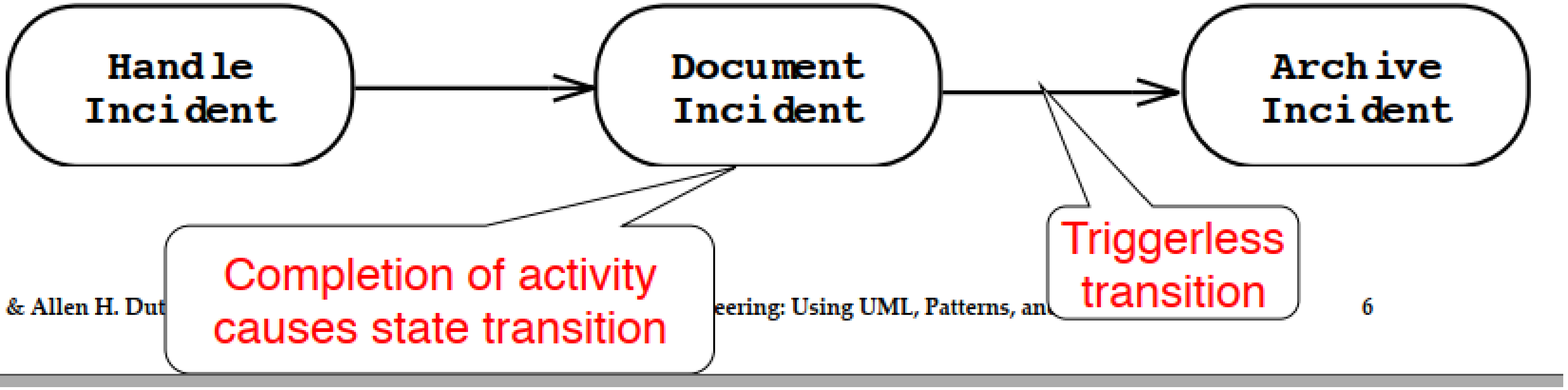
Focus on the set of attributes of a single abstraction (object, system)



Event causes state transition

Activity Diagram for Incident

(Focus on dataflow in a system)



Completion of activity causes state transition

Triggerless transition