Week 2

CS2212

Introduction to Software Engineering

Announcements

Ask Questions Live

cs1.ca/ask

Some images and content from Software Engineering: A Practitioner's Approach 9th Edition By Roger Pressman and Bruce Maxim

Announcements

- Team requests are due by Friday (January 20th) by midnight
- Join a group on OWL (via the Site Info tab) to request a group.
- Next week, groups will be finalized, and your team will be assigned to a TA.
- Weekly group meetings start next week (week of January 23rd)
- Your assigned TA will reach out to your team via e-mail to set up a meeting time/date.

Announcements

you only have to include two of the buildings, but you have
one feature you must have is having differenty layers, another one is be able to have discription
and you have to able to mark the favorites. you could label and highlight favorites
have a list of interests
have a text based search
be able to scroll up and down, but zooming is not required
able to click on the map to add a point of interests
having a help session telling the user how to use the sys
and have a developer mode, and you have to ask for a pin to get into the developer mode
ffinifhsing these basic features you

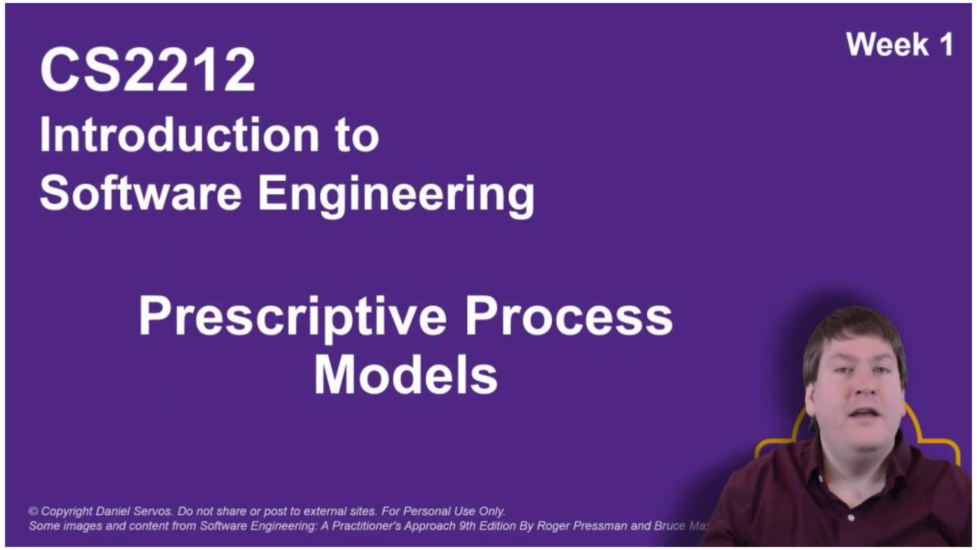
- Descriptions for three of the project components have been posted to OWL:

Project grade is not only project implementation!

  - Project Management *(first draft of team contact due February 3rd)*
  - Peer Review
  - Requirements Documentation *(due February 3rd)*
- Project specification also on OWL.
- Make sure to read all of these as soon as possible.

Announcements

- Two videos to watch this week (found on Week 2 OWL tab):

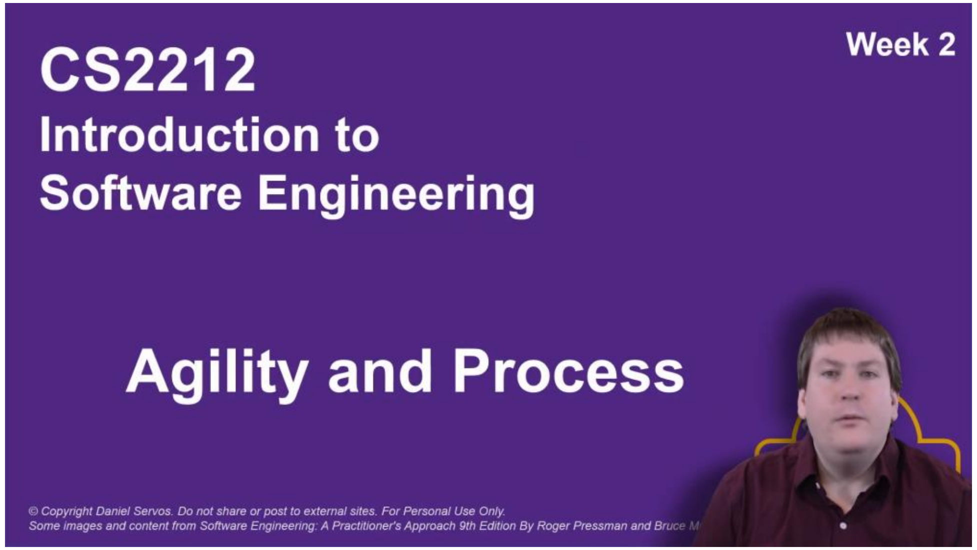


Week 1

CS2212
Introduction to
Software Engineering

Prescriptive Process
Models

Some images and content from Software Engineering: A Practitioner's Approach 9th Edition By Roger Pressman and Bruce Ma



Week 2

CS2212
Introduction to
Software Engineering

Agility and Process

Some images and content from Software Engineering: A Practitioner's Approach 9th Edition By Roger Pressman and Bruce M

- Videos will pick up from where we left off on Process Models.

Announcements

- Two readings for this week:
  - **Chapter 3:** Agile and Process
  - **Appendix 1:** An Introduction to UML
- Also recommend reading Chapter 4 this week, will be covered next week but it’s good to stay ahead.

Week 2

CS2212

Introduction to Software Engineering

UML: Unified Modeling Language

Ask Questions Live

cs1.ca/ask

Some images and content from Software Engineering: A Practitioner's Approach 9th Edition By Roger Pressman and Bruce Maxim

What is UML?

- **UML** stands for **Unified Modeling Language**
  - **Unified**: It brings together several techniques and notations for design as well as a collection of diagrams.
  - **Modeling**: It describes a software system and its design at a high level of abstraction.
  - **Language**: It provides the means to communicate this design in a logical, consistent, and comprehensible fashion.
- UML is an open standard controlled by the Object Management Group (OMG).

What is UML?

Goals of UML:

- Enable the modeling of **object-oriented designs**.
- Visually depict various aspects of the overall design of a solution.
- Provide extensibility and specialization mechanisms to extend core concepts.
- Be **independent of particular programming languages** and development processes.
- Support higher-level development concepts such as collaborations, frameworks, patterns, and components.

What is UML?

- **Tools for creating UML diagrams:**
  - Microsoft Visio
  - diagrams.net Online Diagram Editor
  - DIA Diagram Editor (http://dia-installer.de/)
  - Many other options.

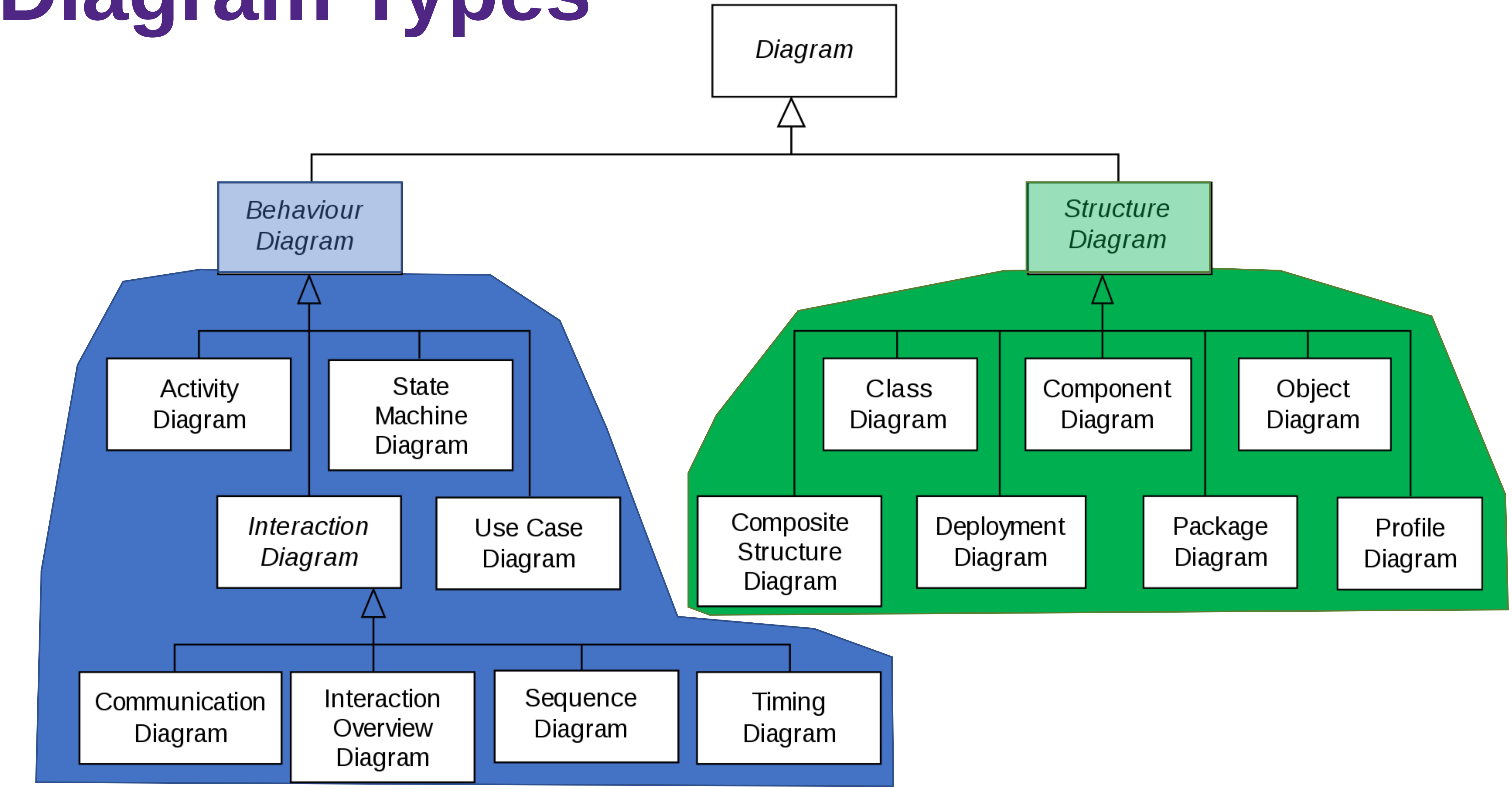
I recommend using diagrams.net, it is free and web based.

Diagram Types

13 Official Diagrams as of UML 2.0

Diagram	Purpose
Activity	Procedural and parallel behavior
Class most- seen in real life	Class, features, and relationships
Communication	Interaction between objects; emphasis on links
Component	Structure and connections of components
Composite Structure	Runtime decomposition of a class
Deployment	Deployment of artifacts to nodes
Interaction Overview	Mix of sequence and activity diagrams
Object	Example configuration of instances
Package	Compile-time hierarchic structure
Sequence	Interaction between objects; emphasis on sequence
State Machine	How events change an object over its life
Timing	Interaction between objects; emphasis on timing
Use Case	How users interact with a system

Diagram Types



Building Blocks

- There are a number of notations or *“building blocks”* that are common to most UML diagrams.
- We will go through a few before we get into individual diagrams.

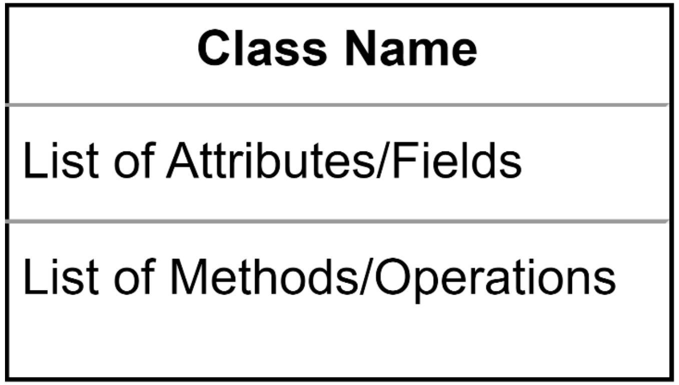
Building Blocks: Things

Structural Things

- Define a static part of the model
- Represent physical and conceptual elements

Class

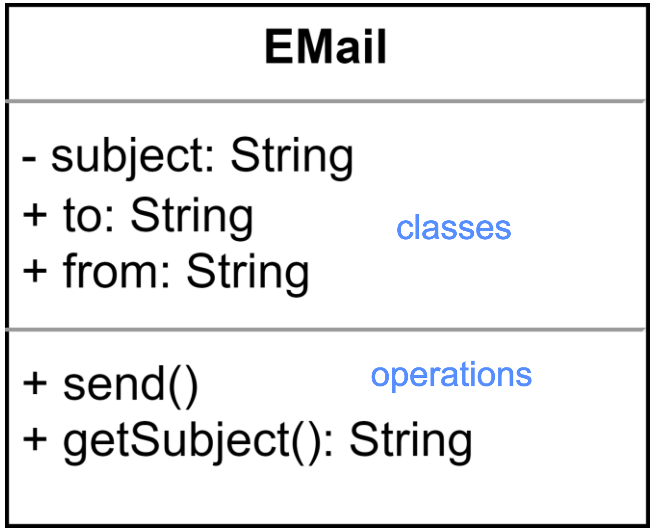
- Class represents a set of objects having similar responsibilities.



Class Name

List of Attributes/Fields

List of Methods/Operations



EMail

- subject: String
+ to: String
+ from: String

classes

+ send()
+ getSubject(): String

operations

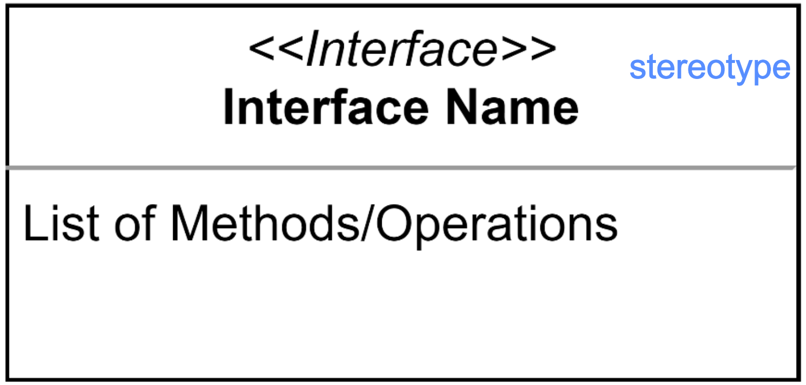
Building Blocks: Things

Structural Things

- Define a static part of the model
- Represent physical and conceptual elements

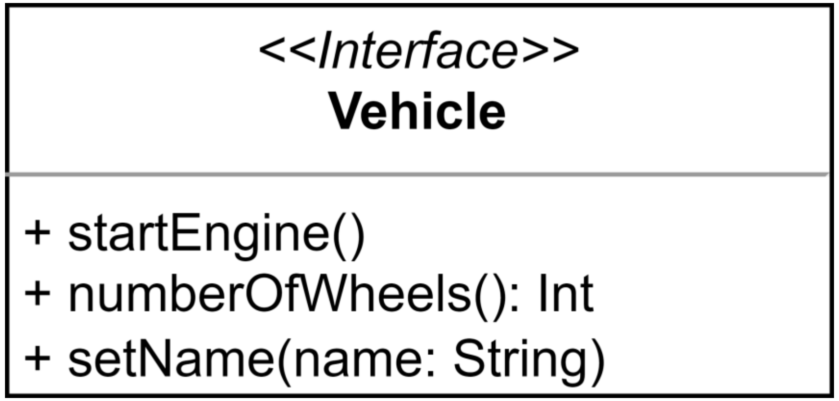
Interface

- Interface defines a set of operations, which specify the responsibility of a class.



<<Interface>>
Interface Name
stereotype

List of Methods/Operations



<<Interface>>
Vehicle

+ startEngine()
+ numberOfWheels(): Int
+ setName(name: String)

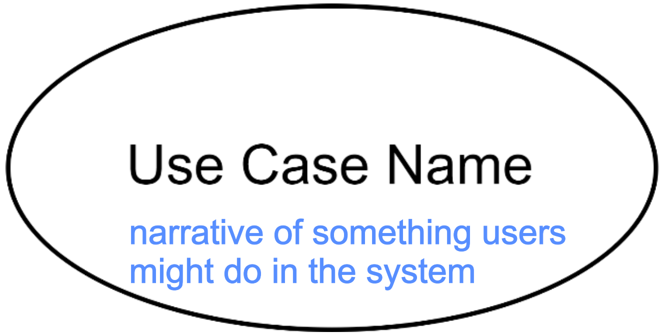
Building Blocks: Things

Structural Things

- Define a static part of the model
- Represent physical and conceptual elements

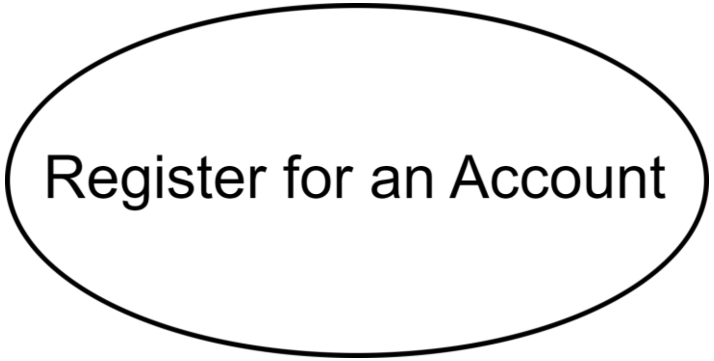
Use Case

- Use case represents a set of actions performed by a system for a specific goal.



Use Case Name

narrative of something users might do in the system



Register for an Account

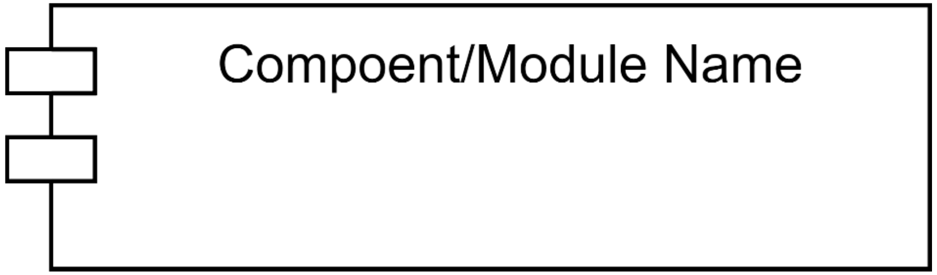
Building Blocks: Things

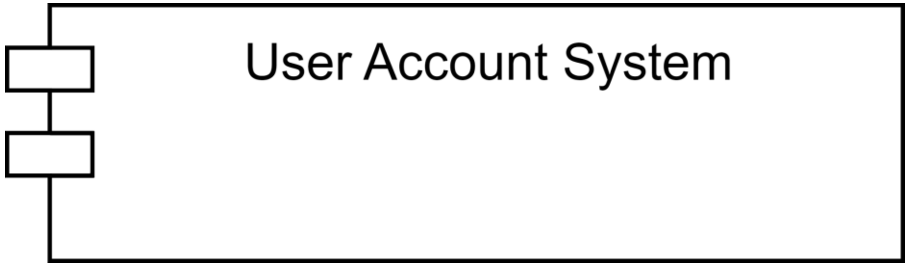
Structural Things

- Define a static part of the model
- Represent physical and conceptual elements

Component

- Component describes a modular part of a system. This may be a collection of classes, interfaces, etc.





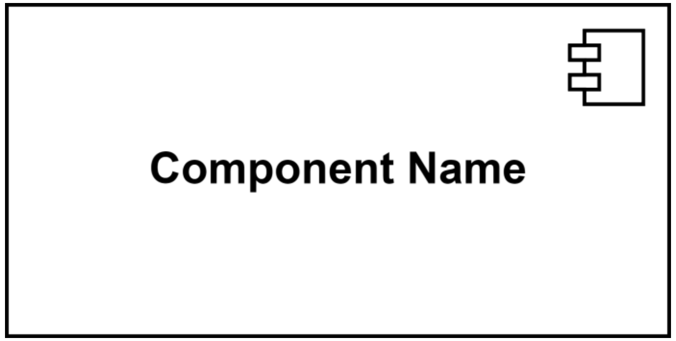
Building Blocks: Things

Structural Things

- Define a static part of the model
- Represent physical and conceptual elements

Component

- Alternative notations for components:



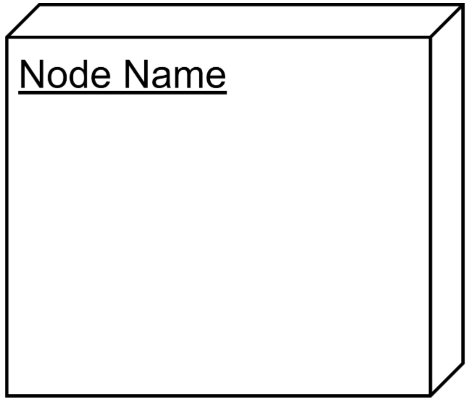
Building Blocks: Things

Structural Things

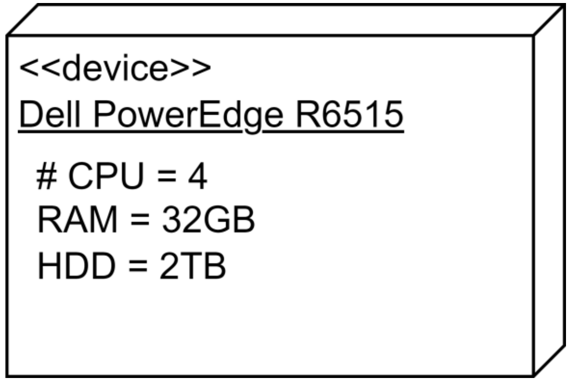
- Define a static part of the model
- Represent physical and conceptual elements

Node

- A node can be defined as a physical element that exists at run time (e.g. a hardware device).



Node Name



<<device>>

Dell PowerEdge R6515

# CPU = 4
RAM = 32GB
HDD = 2TB

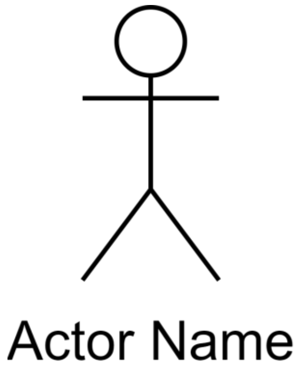
Building Blocks: Things

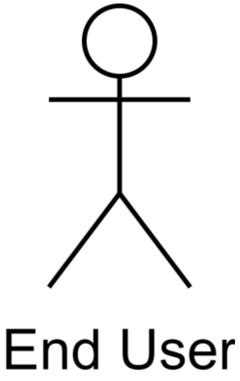
Structural Things

- Define a static part of the model
- Represent physical and conceptual elements

Actor

- Actor specifies a role played by a user or any other **external** system that interacts with our system.





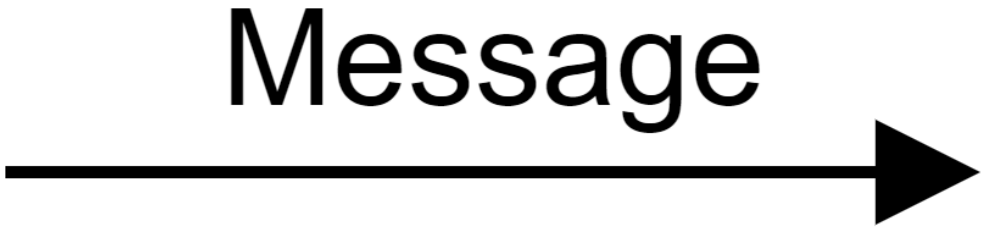
Building Blocks: Things

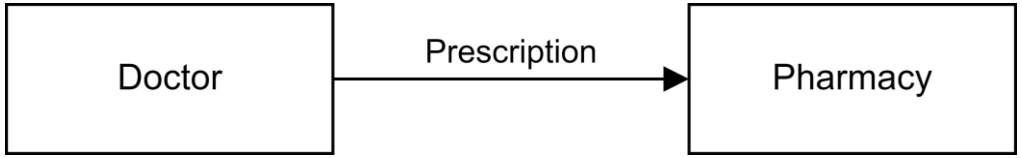
Behavioral Things

- Defines dynamic parts of the model.

Interaction

- Interaction is defined as a behavior that consists of a group of messages exchanged among elements to accomplish a specific task.





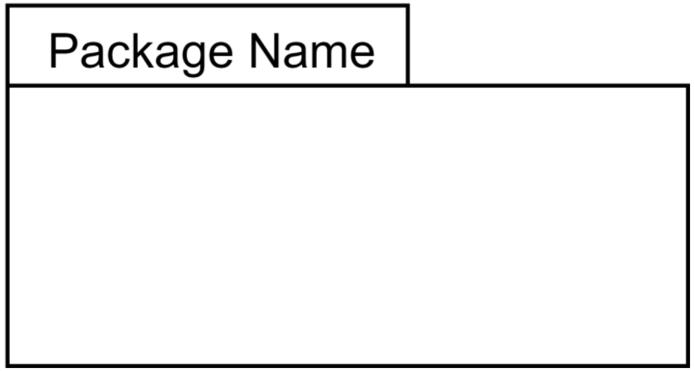
Building Blocks: Things

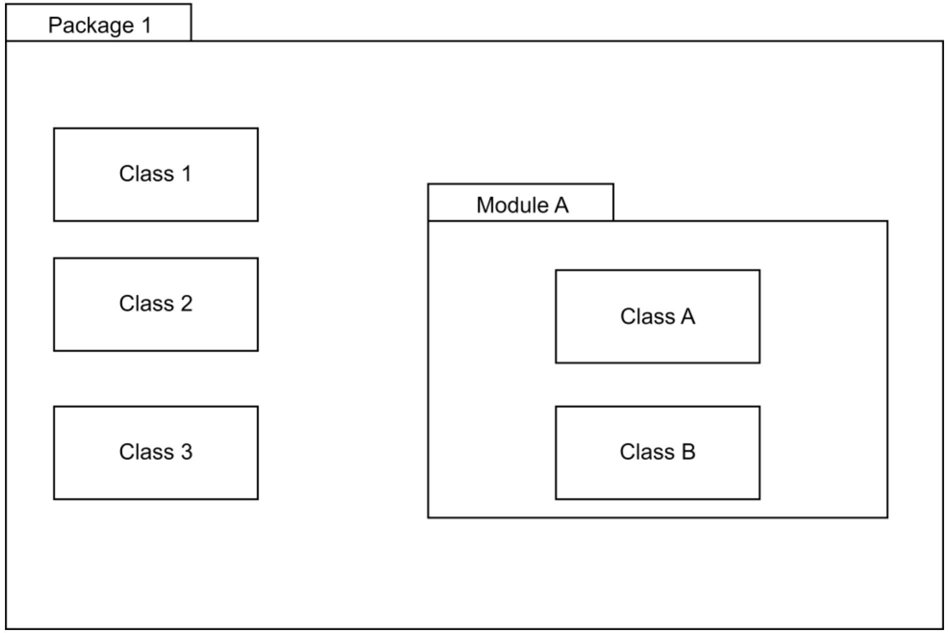
Grouping Things

- Mechanisms to group elements of a UML model together. There is only one grouping thing available

Package

- Packages group other things and provide a common namespace.





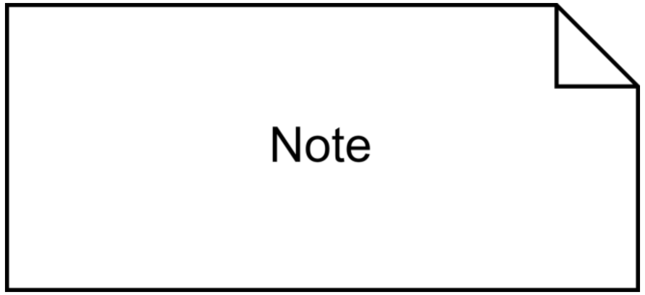
Building Blocks: Things

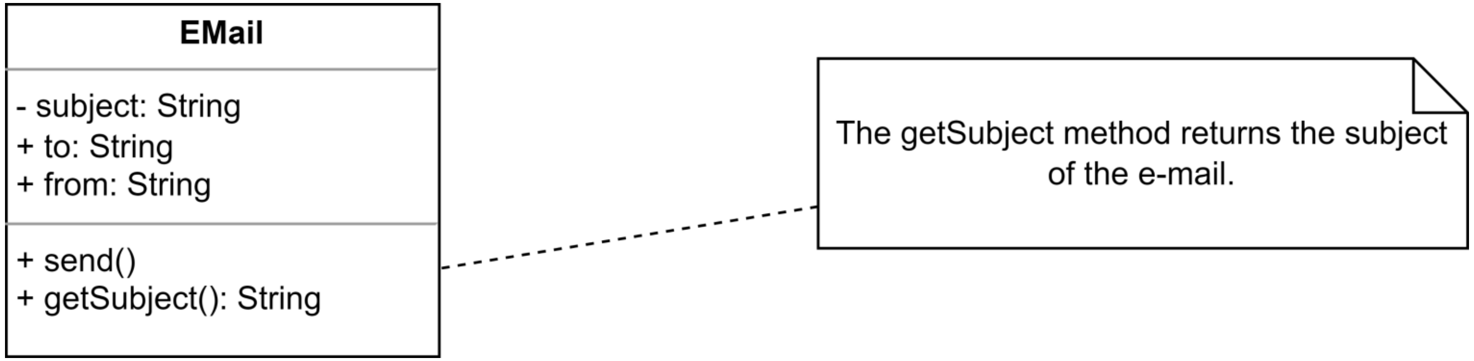
Annotational Things

- Mechanism to capture remarks, descriptions, and comments of UML model elements.

Note

- A note is used to render comments, constraints, etc. of an UML element.





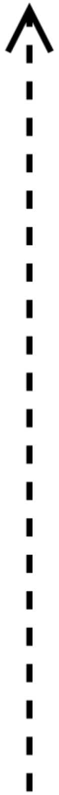
Building Blocks: Relationships

- Relationships show how the elements are associated with each other and this association describes the functionality of the system.
- **Relationship Types:**
  - Dependency
  - Association
  - Generalization
  - Realization

Building Blocks: Relationships

- Relationships show how the elements are associated with each other and this association describes the functionality of the system.

- **Relationship Types:**
  - **Dependency**
  - Association
  - Generalization
  - Realization



A dependency is a relationship between two things in which change in one element also affects the other.

Typically, dependency relationships do not have names or labels.

Arrow indicates direction of dependency.

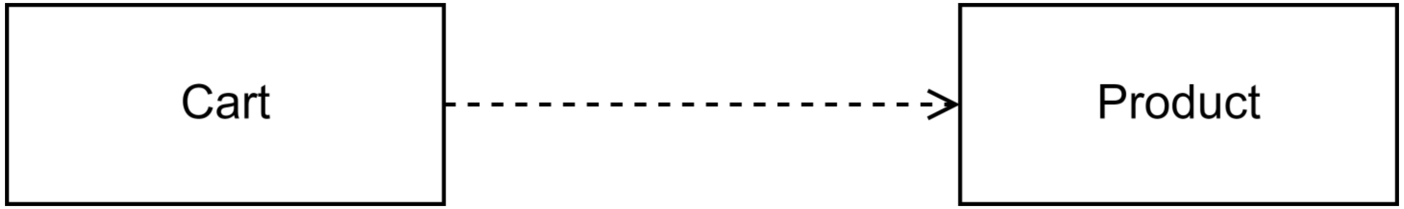
Building Blocks: Relationships

- Relationships show how the elements are associated with each other and this association describes the functionality of the system.

- **Relationship Types:**
    - **Dependency**
    - Association
    - Generalization
    - Realization

Example 1:

In an e-commerce application, a Cart class depends on a Product class because the Cart class uses the Product class as a parameter for an add operation. In a class diagram, a dependency relationship points from the Cart class to the Product class. As the following figure illustrates, the Cart class is, therefore, the client, and the Product class is the supplier.



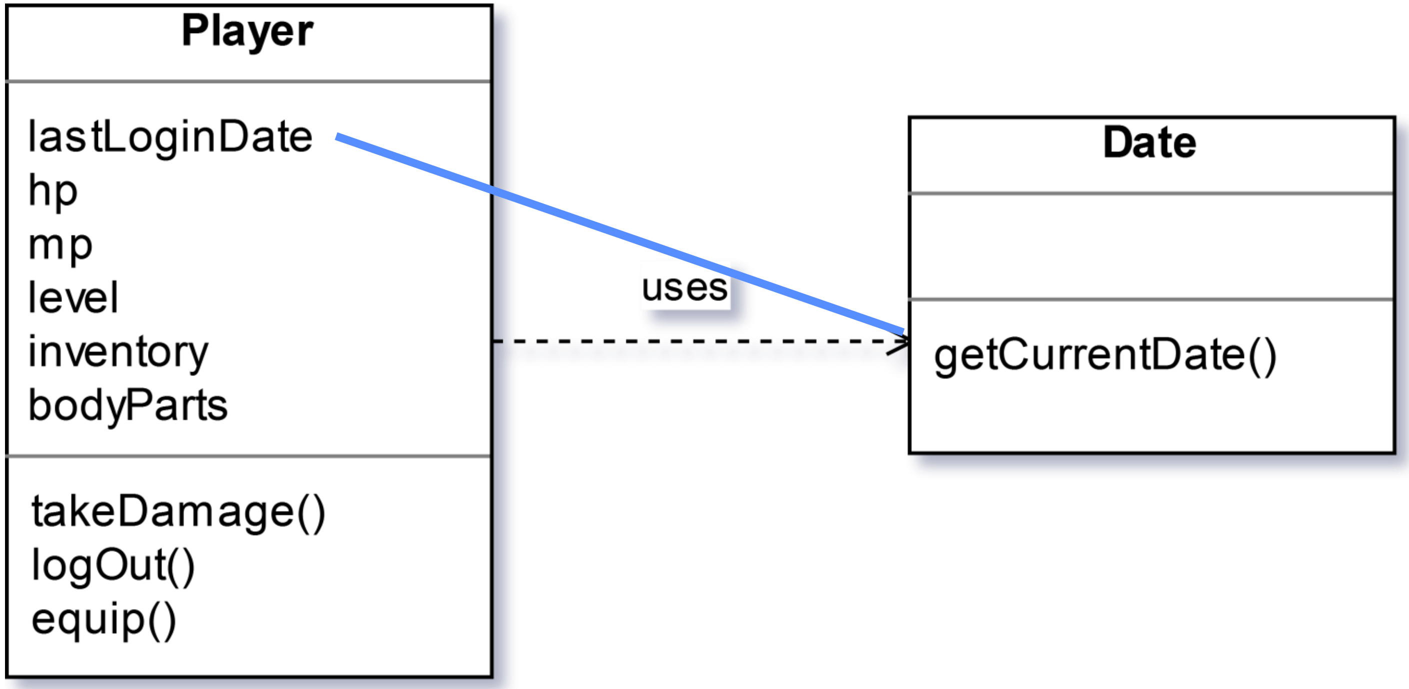
Building Blocks: Relationships

- Relationships show how the elements are associated with each other and this association describes the functionality of the system.

- **Relationship Types:**

  - **Dependency**
  - Association
  - Generalization
  - Realization

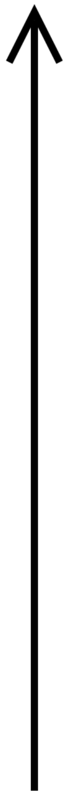
Example 2:



Building Blocks: Relationships

- Relationships show how the elements are associated with each other and this association describes the functionality of the system.

- **Relationship Types:**
  - Dependency
  - **Association**
  - Generalization
  - Realization



An association is a structural relationship that represents how two entities are linked or connected to each other within a system.

If two classes in a model need to communicate with each other, there must be a link between them, and that can be represented by an association.

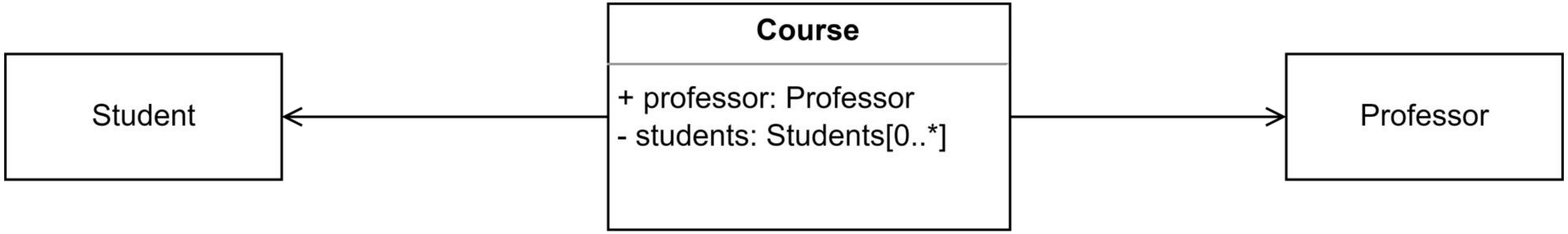
The arrow points in the direction of navigability.

Building Blocks: Relationships

- Relationships show how the elements are associated with each other and this association describes the functionality of the system.

- **Relationship Types:**
    - Dependency
    - **Association**
    - Generalization
    - Realization

Example 1: Courses have one Professor and multiple Students

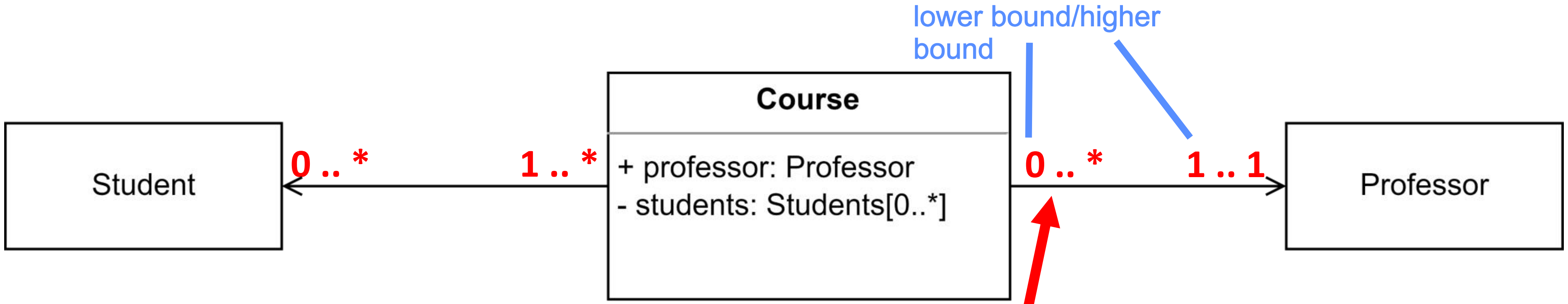


Building Blocks: Relationships

- Relationships show how the elements are associated with each other and this association describes the functionality of the system.

- **Relationship Types:**
  - Dependency
  - **Association**
  - Generalization
  - Realization

Example 2: With multiplicity shown



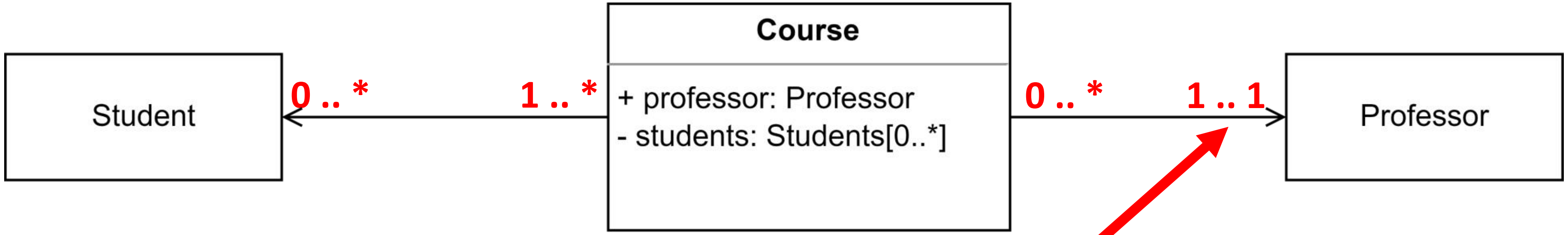
Professors teach zero or more courses

Building Blocks: Relationships

- Relationships show how the elements are associated with each other and this association describes the functionality of the system.

- **Relationship Types:**
  - Dependency
  - **Association**
  - Generalization
  - Realization

Example 2: With multiplicity shown



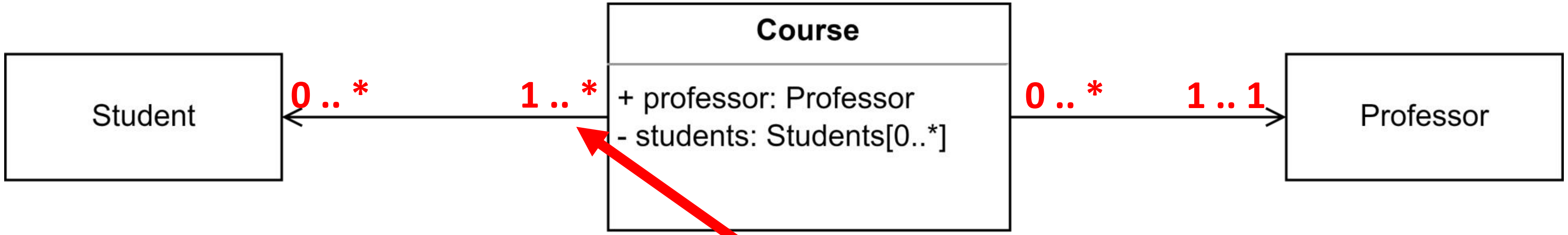
Each course has exactly one professor

Building Blocks: Relationships

- Relationships show how the elements are associated with each other and this association describes the functionality of the system.

- **Relationship Types:**
  - Dependency
  - **Association**
  - Generalization
  - Realization

Example 2: With multiplicity shown



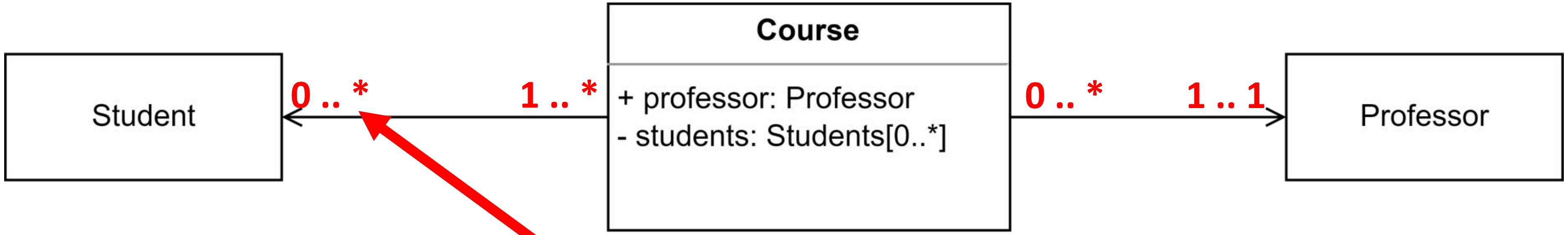
Students take one or more courses

Building Blocks: Relationships

- Relationships show how the elements are associated with each other and this association describes the functionality of the system.

- **Relationship Types:**
  - Dependency
  - **Association**
  - Generalization
  - Realization

Example 2: With multiplicity shown



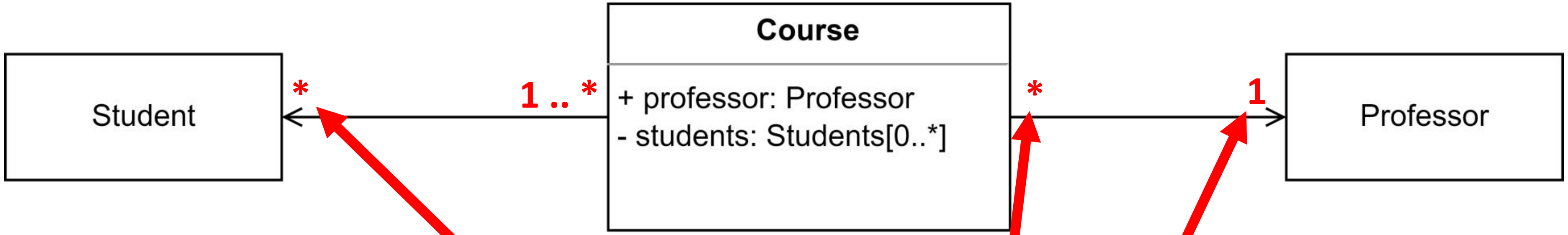
A course has zero or more students

Building Blocks: Relationships

- Relationships show how the elements are associated with each other and this association describes the functionality of the system.

- **Relationship Types:**
  - Dependency
  - **Association**
  - Generalization
  - Realization

Example 2: With multiplicity shown



Can use shorthand in some cases
(1..1 = 1 and 0..* = *)

Building Blocks: Relationships

- Relationships show how the elements are associated with each other and this association describes the functionality of the system.

- **Relationship Types:**
  - Dependency
  - **Association**
  - Generalization
  - Realization

Multiplicity

*	= zero to many
1..*	= one to many
1	= exactly one
1..1	= exactly one
0..1	= zero to one
0..*	= zero to many

Can use other numbers:

2..5	= two to five
8..*	= eight to many
3	= exactly three

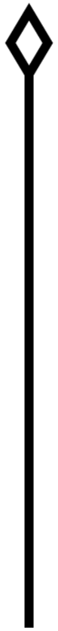
Building Blocks: Relationships

- Relationships show how the elements are associated with each other and this association describes the functionality of the system.

- **Relationship Types:**
  - Dependency
  - **Association**
  - Generalization
  - Realization

Associations can be further subdivided into Aggregation and Composition relationships.

Aggregation



- Child element can exist independent of a parent.
- It constitutes a **Has-a** relationship. e.g. someone has a sth
- It forms a **weak** association.
- **Example:** A doctor has patients, when the doctor gets transfer to another hospital, the patients do not accompany to a new workplace.

Building Blocks: Relationships

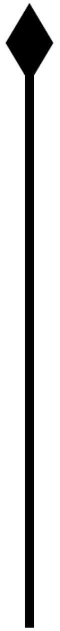
- Relationships show how the elements are associated with each other and this association describes the functionality of the system.

- **Relationship Types:**

  - Dependency
  - **Association**
  - Generalization
  - Realization

Associations can be further subdivided into Aggregation and Composition relationships.

Composition



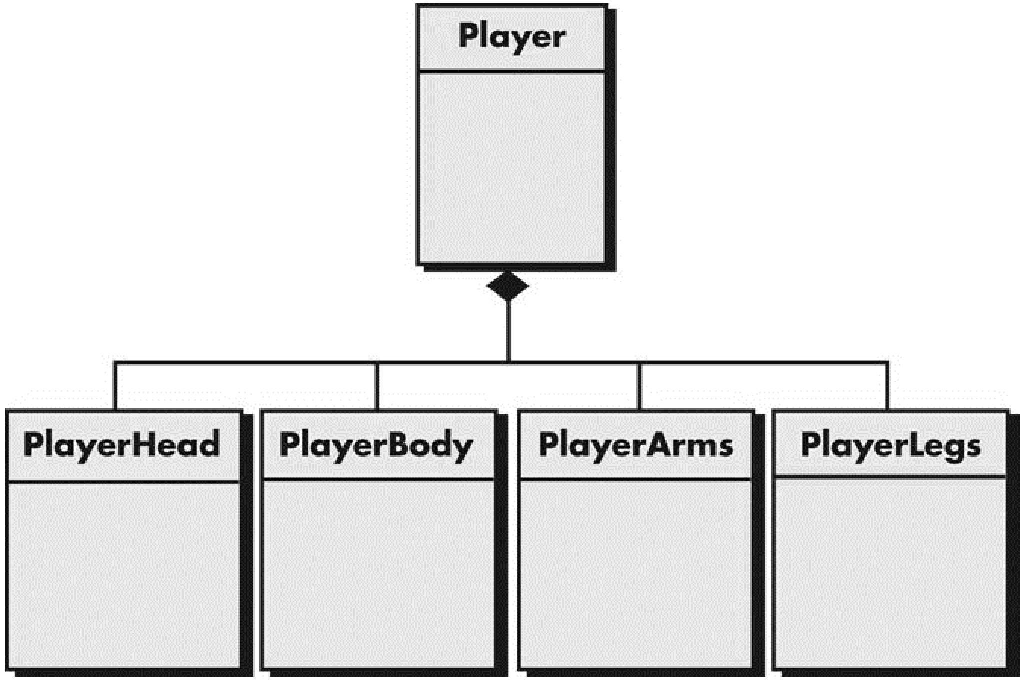
- The child element cannot exist independent of the parent.
- It constitutes a **Part-of** relationship.
- It forms a **strong** association.
- **Example:** A hospital and its wards. If the hospital is destroyed, the wards also get destroyed.

Building Blocks: Relationships

- Relationships show how the elements are associated with each other and this association describes the functionality of the system.

- **Relationship Types:**
  - Dependency
  - **Association**
  - Generalization
  - Realization

Composition Example



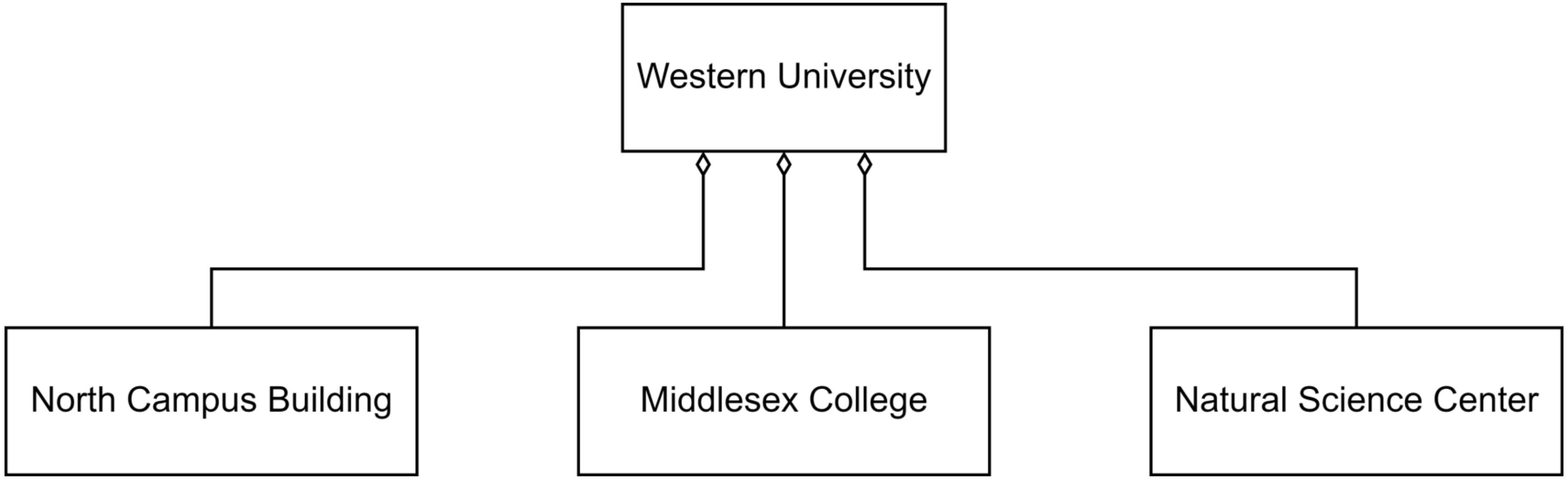
PlayerHead, PlayerBody, etc. are part of the Player, they cannot exist without the Player.

Building Blocks: Relationships

- Relationships show how the elements are associated with each other and this association describes the functionality of the system.

- **Relationship Types:**
  - Dependency
  - **Association**
  - Generalization
  - Realization

Aggregation Example

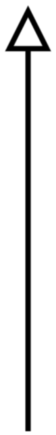


If the university was shutdown, the buildings would still exist

Building Blocks: Relationships

- Relationships show how the elements are associated with each other and this association describes the functionality of the system.

- **Relationship Types:**
  - Dependency
  - Association
  - **Generalization**
  - Realization



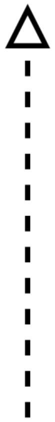
Generalizations can be defined as a relationship which connects a specialized element with a generalized element. It basically describes the inheritance relationship in the world of objects.

The arrow points to the parent object in the relationship.

Building Blocks: Relationships

- Relationships show how the elements are associated with each other and this association describes the functionality of the system.

- **Relationship Types:**
  - Dependency
  - Association
  - Generalization
  - **Realization**



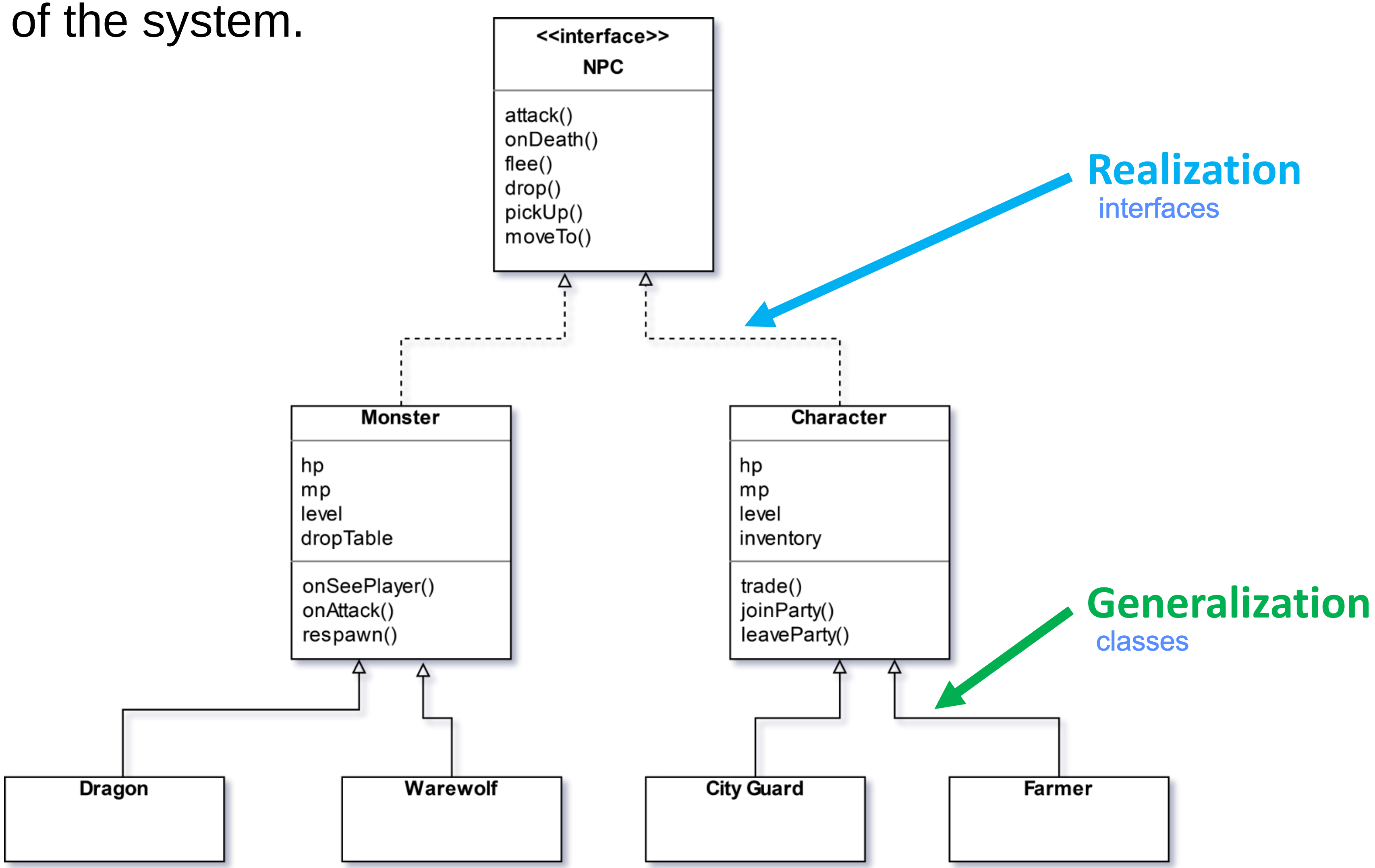
Realizations can be defined as a relationship in which two elements are connected. One element describes some responsibility, which is not implemented and the other one implements them. This relationship exists in case of interfaces.

The arrow points to the parent object in the relationship (the interface).

Building Blocks: Relationships

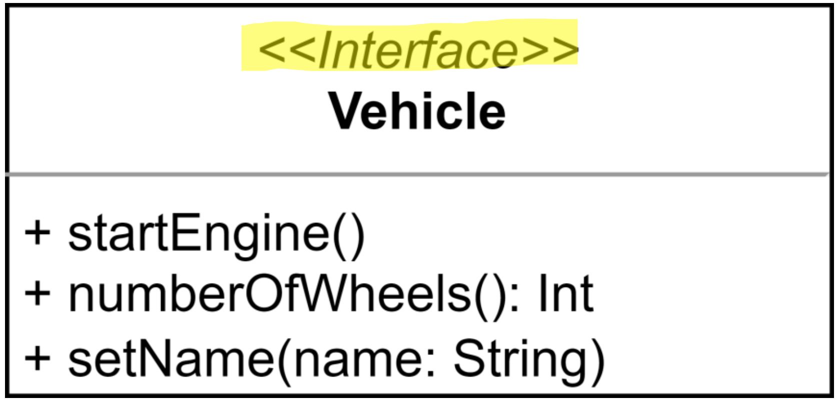
- Relationships show how the elements are associated with each other and this association describes the functionality of the system.

- **Relationship Types:**
  - Dependency
  - Association
  - **Generalization**
  - **Realization**



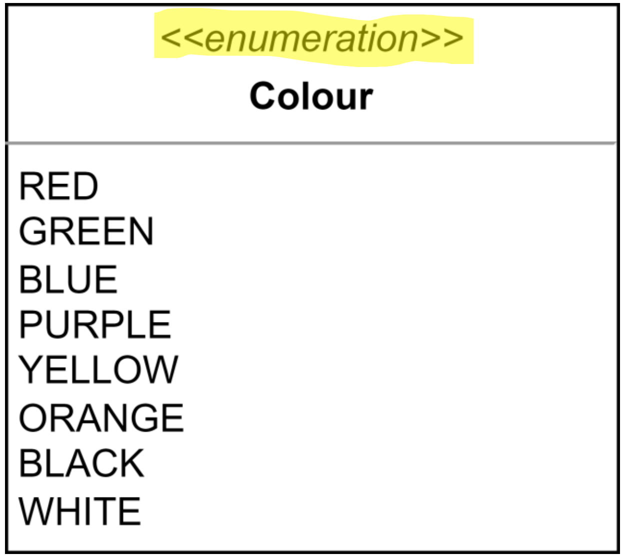
Building Blocks: Stereotypes

- Model element that identifies the purpose of other model elements.
- You can use a stereotype to refine the meaning of a model element.
- Graphically, a stereotype is rendered as a name enclosed by guillemets (« » or, if guillemets proper are unavailable, << >>) and placed above the name of another element.
- **Examples:**



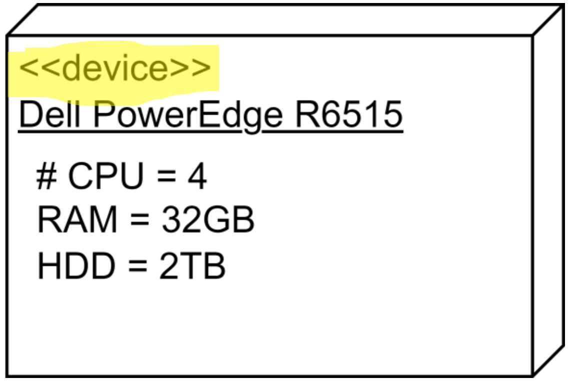
<<Interface>>
Vehicle

+ startEngine()
+ numberOfWheels(): Int
+ setName(name: String)



<<enumeration>>
Colour

RED
GREEN
BLUE
PURPLE
YELLOW
ORANGE
BLACK
WHITE

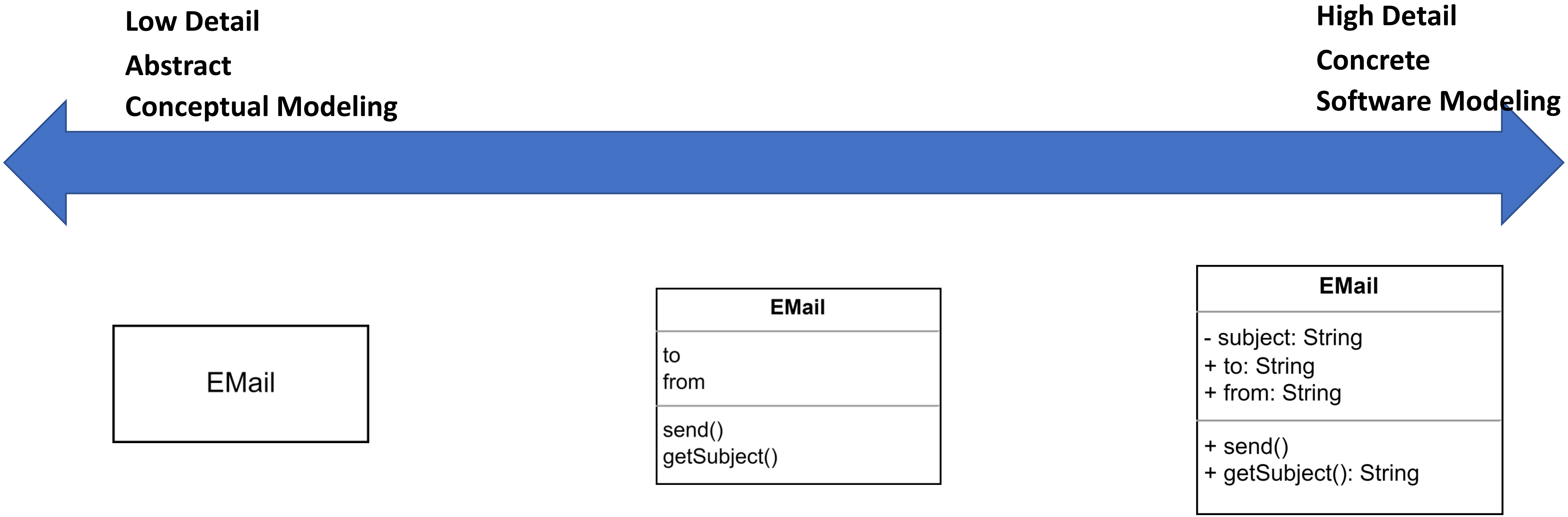


<<device>>
Dell PowerEdge R6515

# CPU = 4
RAM = 32GB
HDD = 2TB

UML Perspectives & Level of Detail

- UML allows you to give differing level of details in your diagrams based on the concept you are focusing on, your target audience, and if you are using it for **conceptual** or **software modeling**.



Low Detail

Abstract

Conceptual Modeling

High Detail

Concrete

Software Modeling

EMail

EMail

to
from

send()
getSubject()

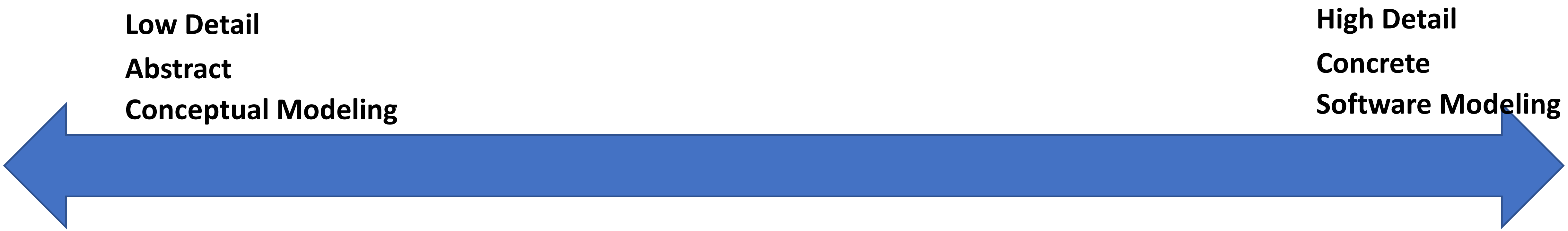
EMail

- subject: String
+ to: String
+ from: String

+ send()
+ getSubject(): String

UML Perspectives & Level of Detail

- UML allows you to give differing level of details in your diagrams based on the concept you are focusing on, your target audience, and if you are using it for **conceptual** or **software modeling**.



Low Detail

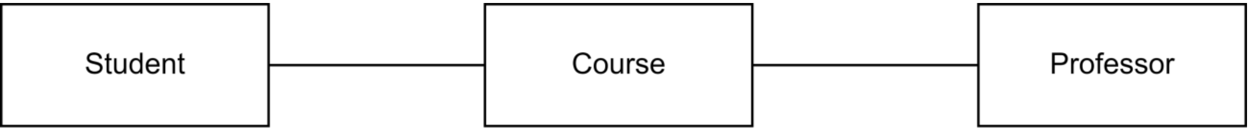
Abstract

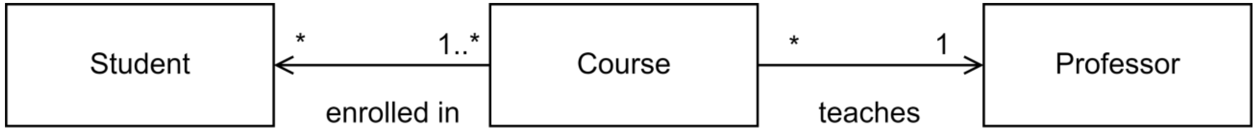
Conceptual Modeling

High Detail

Concrete

Software Modeling





Week 2

CS2212

Introduction to Software Engineering

UML:
Class Diagrams

Ask Questions Live

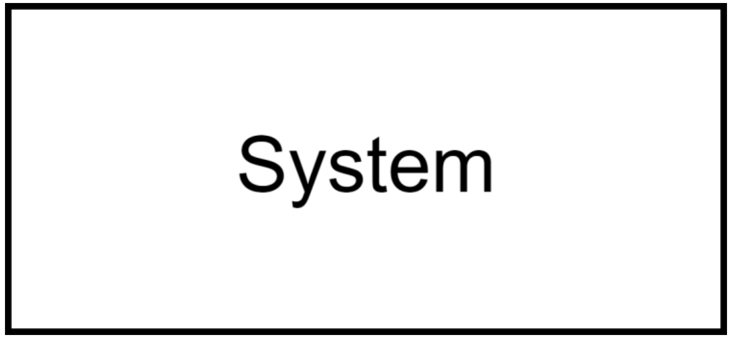
cs1.ca/ask

UML Class Diagram

- A class diagram describes the **types of objects in the system** and the various kinds of **static relationships** that exist among them.
- Also shows **attributes** (fields) and **operations** (methods) of a class.
- Most popular and commonly seen UML diagram type.
- Can be used for both **requirements engineering** (analysis classes) and **design modeling** (design classes).

UML Class Diagram

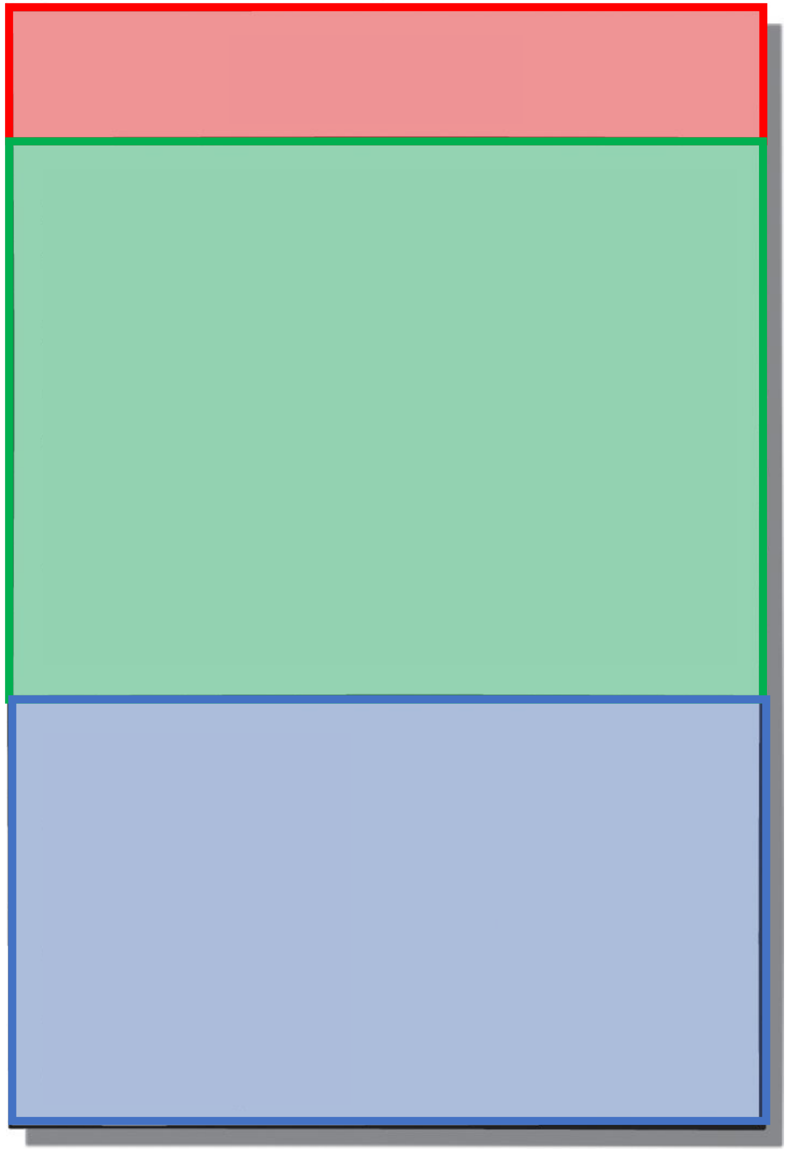
- **Most basic class is just a box:**



- **but we can add far more detail…**

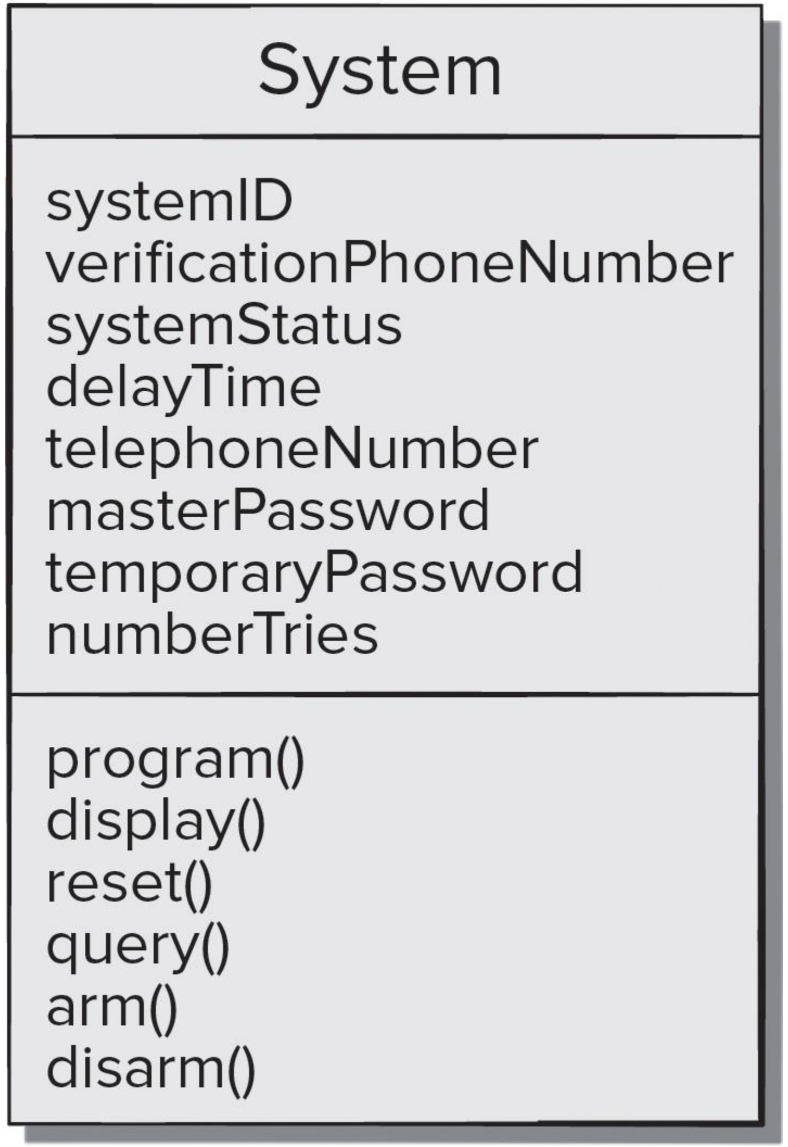
UML Class Diagram

- Components of a Class:
  - Class Name
  - Attributes/Fields
  - Operations/Methods



Adding Details

- For both **attributes** and **operations** in a class diagram, you can **optionally** specify **additional levels of detail**, depending on what is needed from the model in question
  - **For attributes**, you can add types
  - **For operations**, you can add return types and parameter names and types
  - For both **attributes** and **operations**, you can add visibility that indicates which other classes can see and access them (use + for public, - for private, and # for protected).



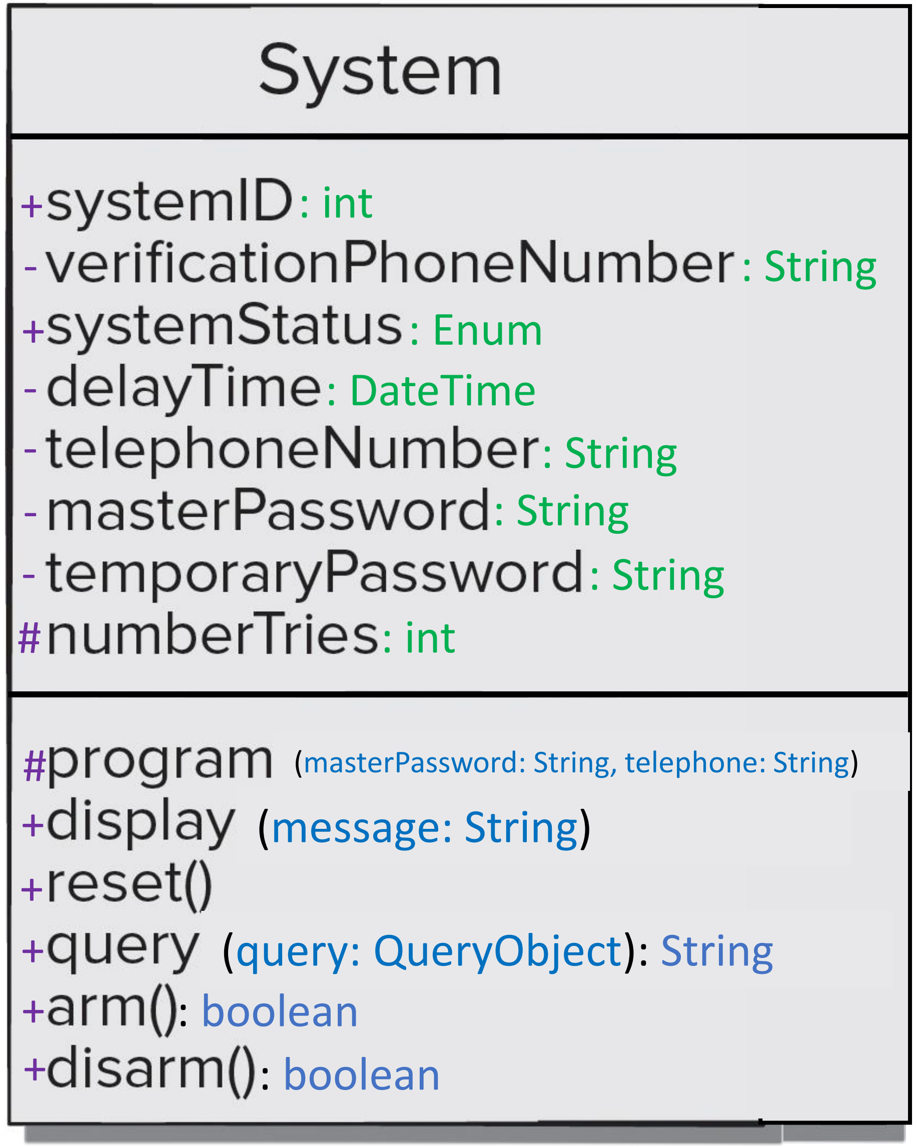
System

systemID
verificationPhoneNumber
systemStatus
delayTime
telephoneNumber
masterPassword
temporaryPassword
numberTries

program()
display()
reset()
query()
arm()
disarm()

Adding Details

- For both **attributes** and **operations** in a class diagram, you can **optionally** specify **additional levels of detail**, depending on what is needed from the model in question
  - **For attributes**, you can add types
  - **For operations**, you can add return types and parameter names and types
  - For both **attributes** and **operations**, you can add visibility that indicates which other classes can see and access them (use + for public, - for private, and # for protected).



System

+systemID: int
-verificationPhoneNumber: String
+systemStatus: Enum
-delayTime: DateTime
-telephoneNumber: String
-masterPassword: String
-temporaryPassword: String
#numberTries: int

#program (masterPassword: String, telephone: String)
+display (message: String)
+reset()
+query (query: QueryObject): String
+arm(): boolean
+disarm(): boolean

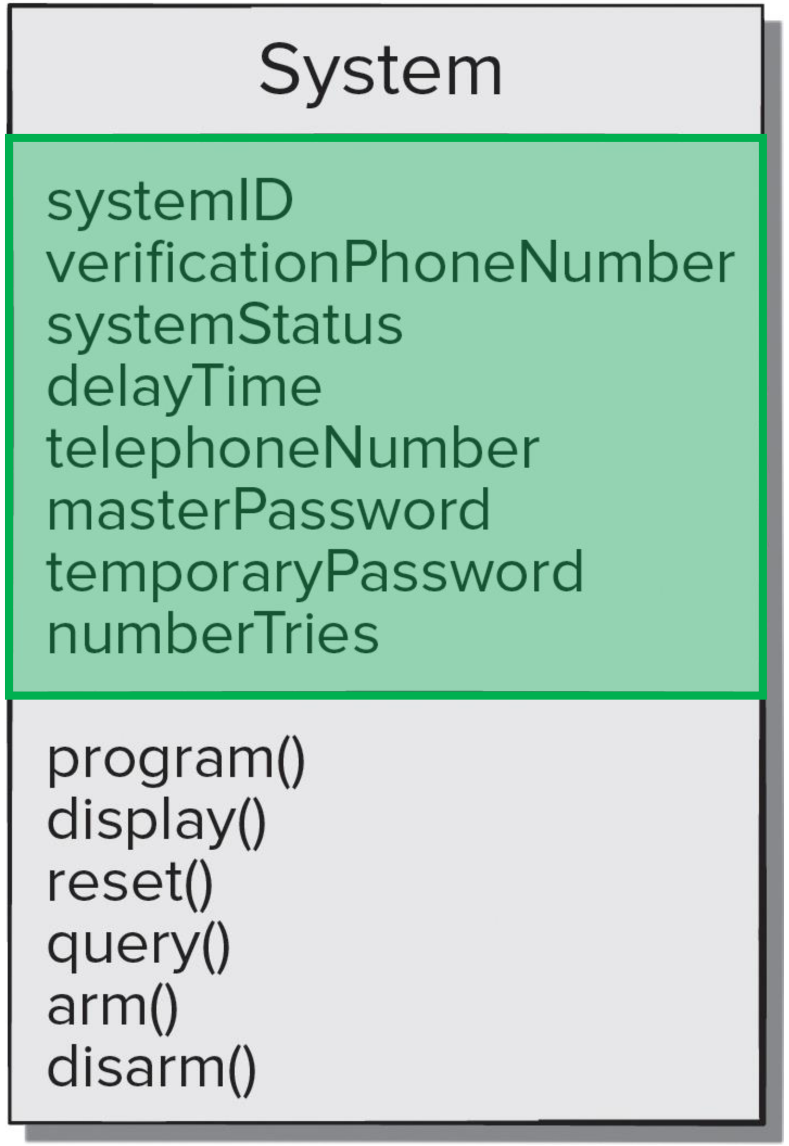
Attributes

Basic Syntax:

```
visibility name: type multiplicity = default
```

Examples:

```
    systemID
    - password: String = "myPass"
    + userIDs: int [1..*]
    # numberTries: int
```



System

systemID
verificationPhoneNumber
systemStatus
delayTime
telephoneNumber
masterPassword
temporaryPassword
numberTries

program()
display()
reset()
query()
arm()
disarm()

Operations

Basic Syntax:

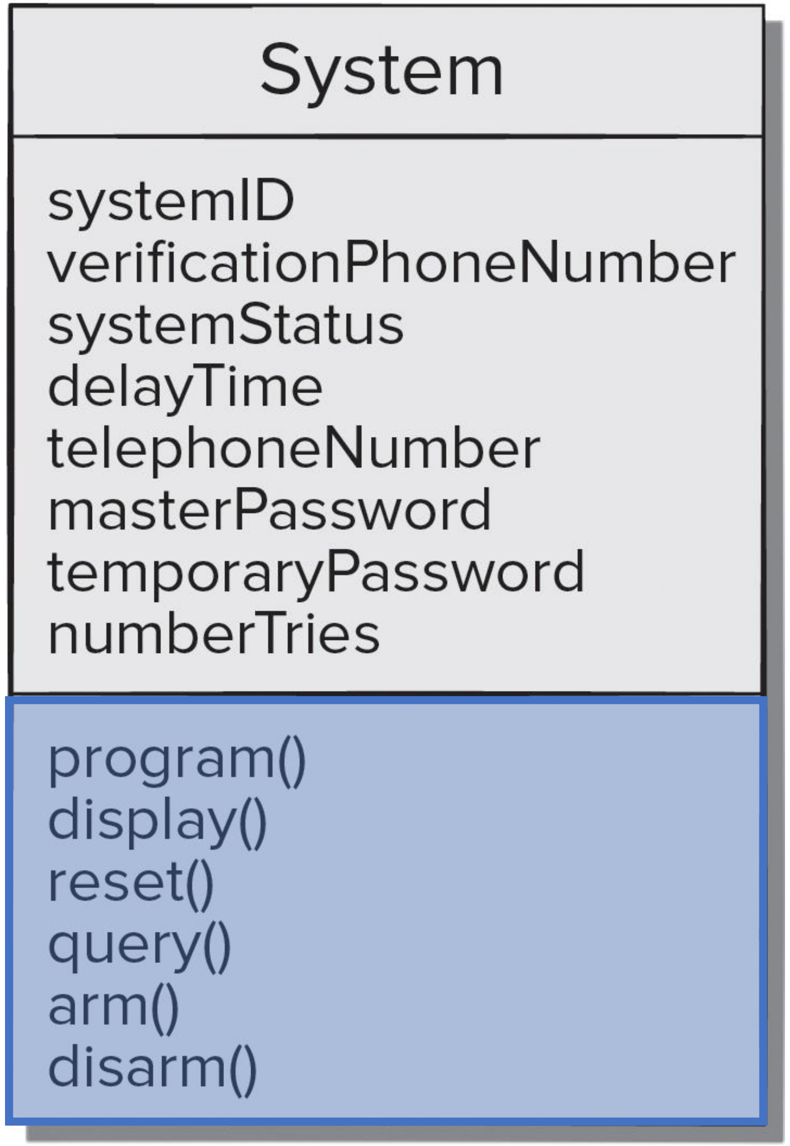
```
visibility name(parameter-list) : return-type
```

parmater-list is a set of:

```
name: type = default-value
```

Examples:

```
program()
+ display(message: String)
+ program(masterPassword: String, telephone: String)
- getOwnerName(): String
```



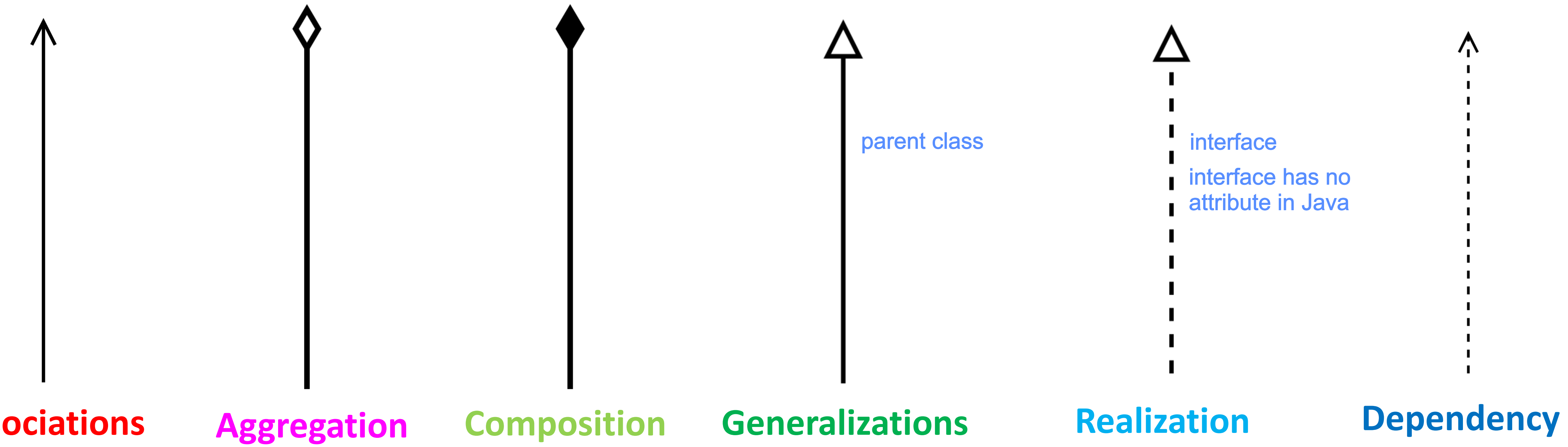
System

systemID
verificationPhoneNumber
systemStatus
delayTime
telephoneNumber
masterPassword
temporaryPassword
numberTries

program()
display()
reset()
query()
arm()
disarm()

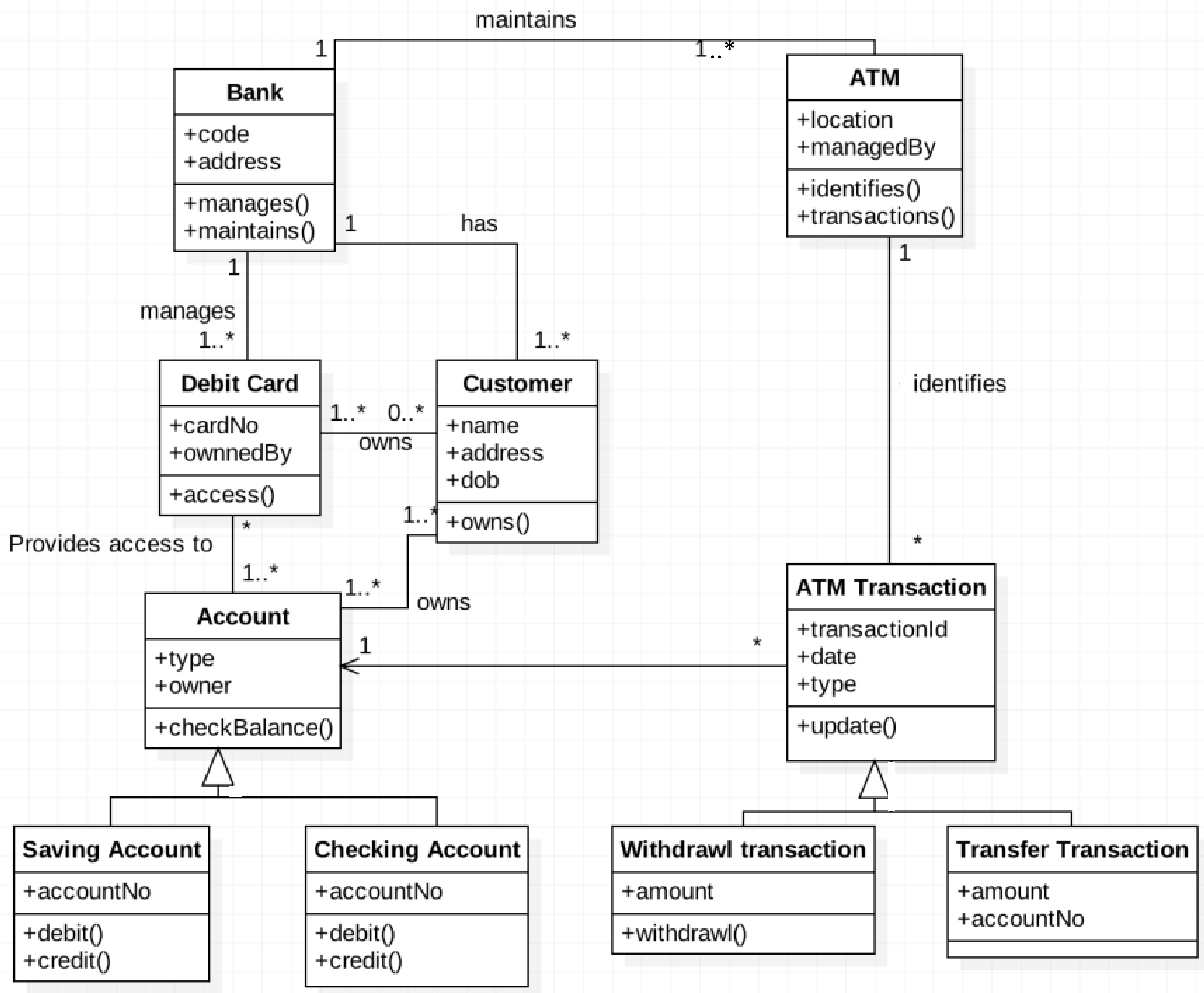
Relationships

- Class diagrams use the same relationship building blocks we have already seen.



Class Diagram

Example



maintains 1 1..*

Bank
+code
+address
+manages()
+maintains()

ATM
+location
+managedBy
+identifies()
+transactions()

1 has 1..*

1 manages 1..*

1 identifies *

Debit Card
+cardNo
+ownnedBy
+access()

1..* 0..* owns

Customer
+name
+address
+dob
+owns()

Provides access to * 1..*

1..* 1..* owns

Account
+type
+owner
+checkBalance()

1 *

ATM Transaction
+transactionId
+date
+type
+update()

Saving Account
+accountNo
+debit()
+credit()

Checking Account
+accountNo
+debit()
+credit()

Withdrawl transaction
+amount
+withdrawl()

Transfer Transaction
+amount
+accountNo

Source: groups.umd.umich.edu

Activity 1: Reverse Engineering a Class

```
public class Person {
    private String name;
    private int age;

    public Person(String initialName) {
        this.name = initialName;
        this.age = 0;
    }

    public void printPerson() {
        System.out.println(this.name + ", age " +   this.age + " years");
    }

    public String getName() {
        return this.name;
    }
}
```

Working on your own or in a small group create a UML class diagram with a single class based on the code shown to the left.

Add as much detail as you can to the class diagram.

Activity 2: Reverse Engineering a Class

```
public class Book {
    private String name;
    private String publisher;
    private ArrayList<Person> authors;

    public ArrayList<Person> getAuthors() {
        return this.authors;
    }

    public void addAuthor(Person author) {
        this.authors.add(author);
    }
}
```

We are now adding a Book class to our system and a relationship between Person and Book.

A Person can be the author of zero or more books.

A book can be authored by 1 or more people.

Your class diagram should now have two classes Book and Person.

Activity 3: Reverse Engineering a Class

```
public class Music implements Media {
    private String title;
    private int lengthInSeconds;
    private ArrayList<Person> authors;

    public ArrayList<Person> getAuthors() {
        return this.authors;
    }

    public void addAuthor(Person author) {
        this.authors.add(author);
    }
}
```

We now want to track more than just books that may be authored by people. For example, we may also want to track Music.

Add an **interface** to your class diagram called Media that has methods getAuthor() and addAuthor(). Assume that the Book class now implements this interface.

Also add the Music class shown to the left to your class diagram.

Note: *you will likely have to redraw your old diagram.*

Activity 4: Reverse Engine

Finally we want to add a Customer class what will implement the Person class and add a customerID, username, and password as well as a genID() method.

```
public class Customer extends Person {
      public int customerID;
      private String username;
      private String password;

      public Customer(String initName, String uname, String pass) {
            super(initName);
            this.customerID = genID();
            this.username = uname;
            this.password = pass;
      }

      private int genID() {
            //this method generates a new unique ID for this customer
            return newID;
      }
}
```

Week 2

CS2212

Introduction to Software Engineering

UML: Activity Diagrams

Ask Questions Live

cs1.ca/ask

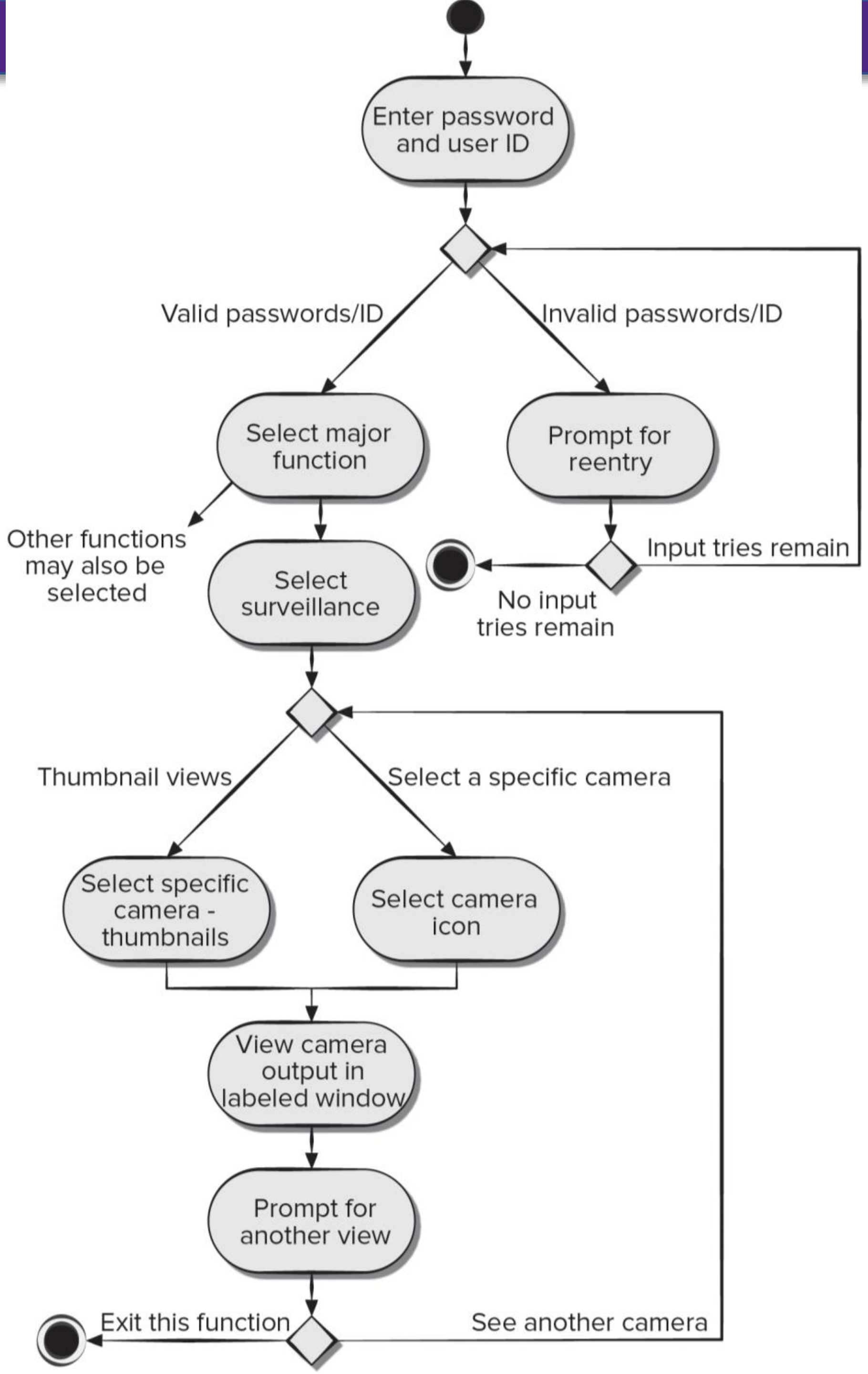
Some images and content from Software Engineering: A Practitioner's Approach 9th Edition By Roger Pressman and Bruce Maxim

UML Activity Diagrams

- Provides a **graphical representation** of the **flow of interaction**.
- Represent **how a system reacts to internal events**.
- May **add additional detail** not directly mentioned (but implied) by a **use case**.
- Similar to a **flowchart** except an **activity diagram** can show **concurrent flows**.
- Can be used for diagraming code and algorithms or more conceptual elements (e.g. used to document **use cases**).

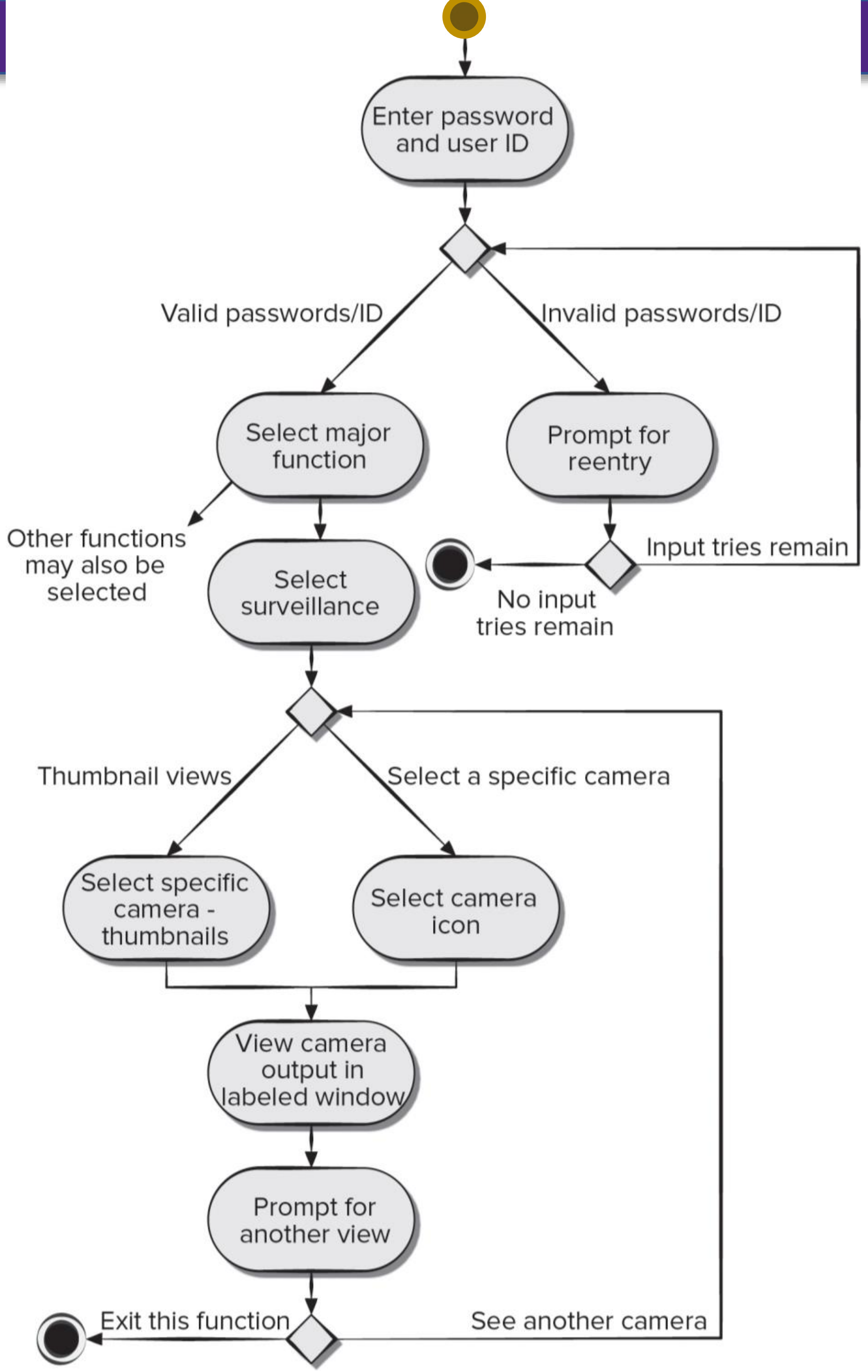
Activity Diagrams

- Initial Node
- Action Node
- Final Node
- Decision Node
- Flow
- Fork
- Join



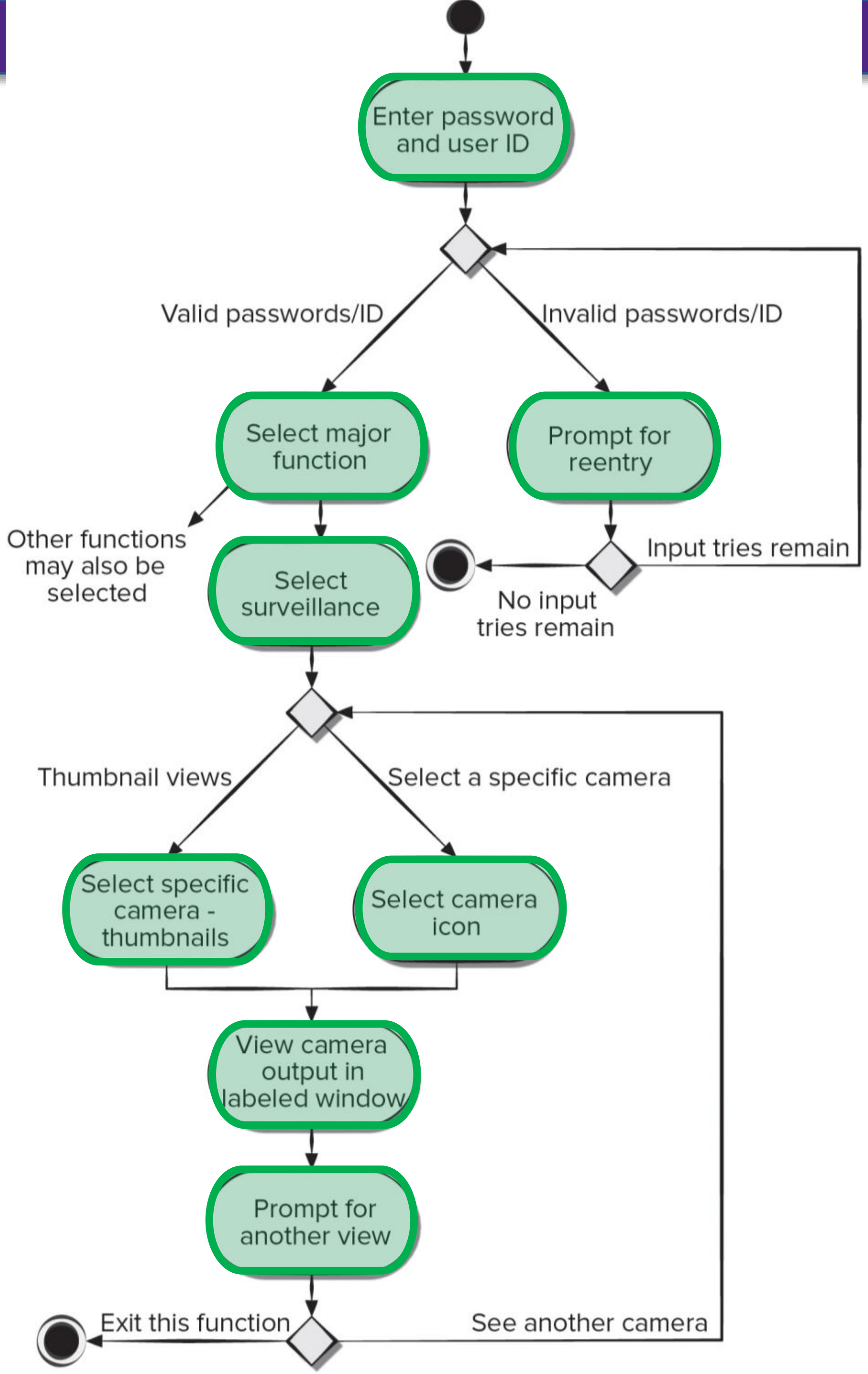
Activity Diagrams

- **Initial Node**
- Action Node
- Final Node
- Decision Node
- Flow
- Fork
- Join



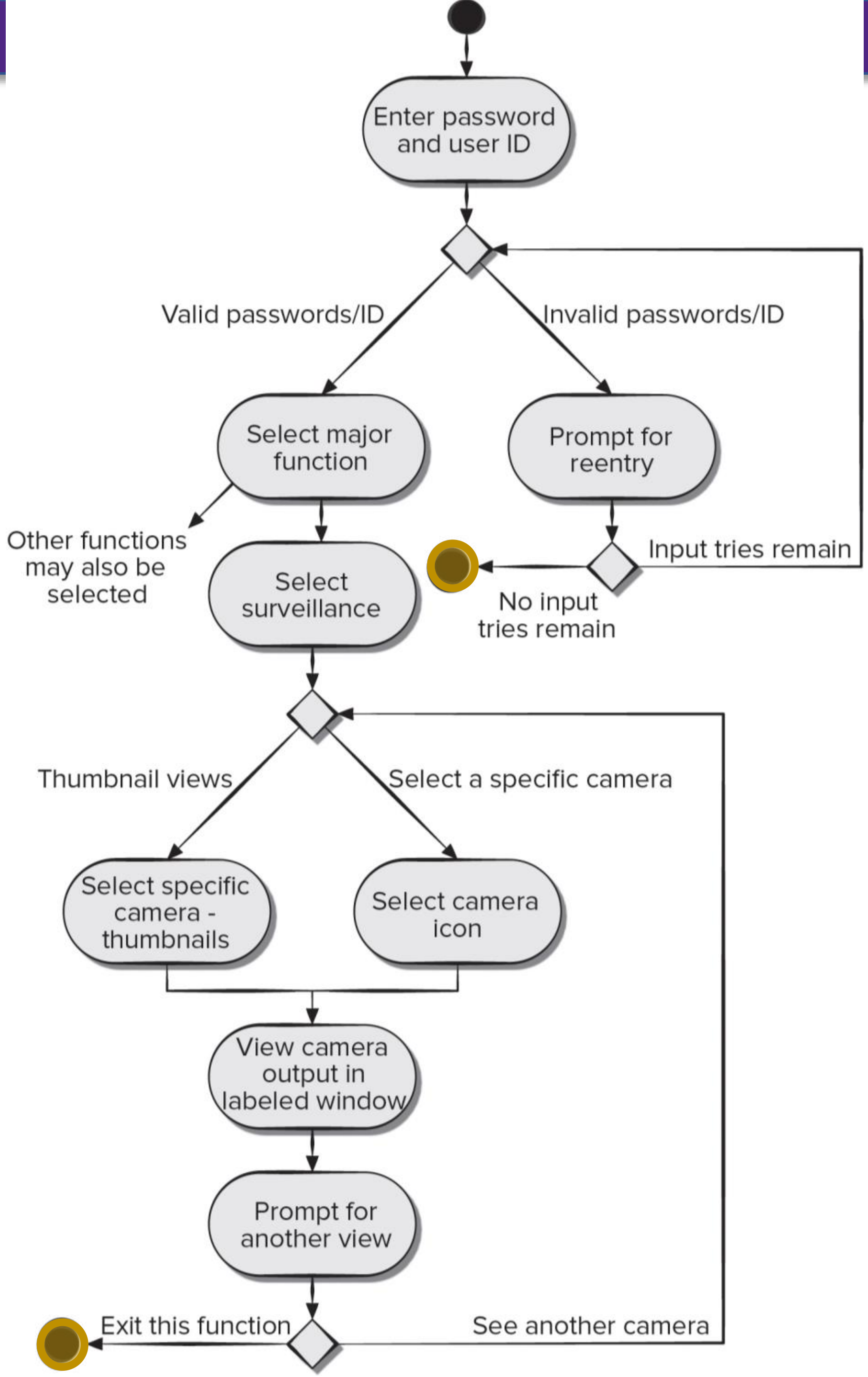
Activity Diagrams

- Initial Node
- **Action Node**
- Final Node
- Decision Node
- Flow
- Fork
- Join



Activity Diagrams

- Initial Node
- Action Node
- **Final Node**
- Decision Node
- Flow
- Fork
- Join



Activity Diagrams

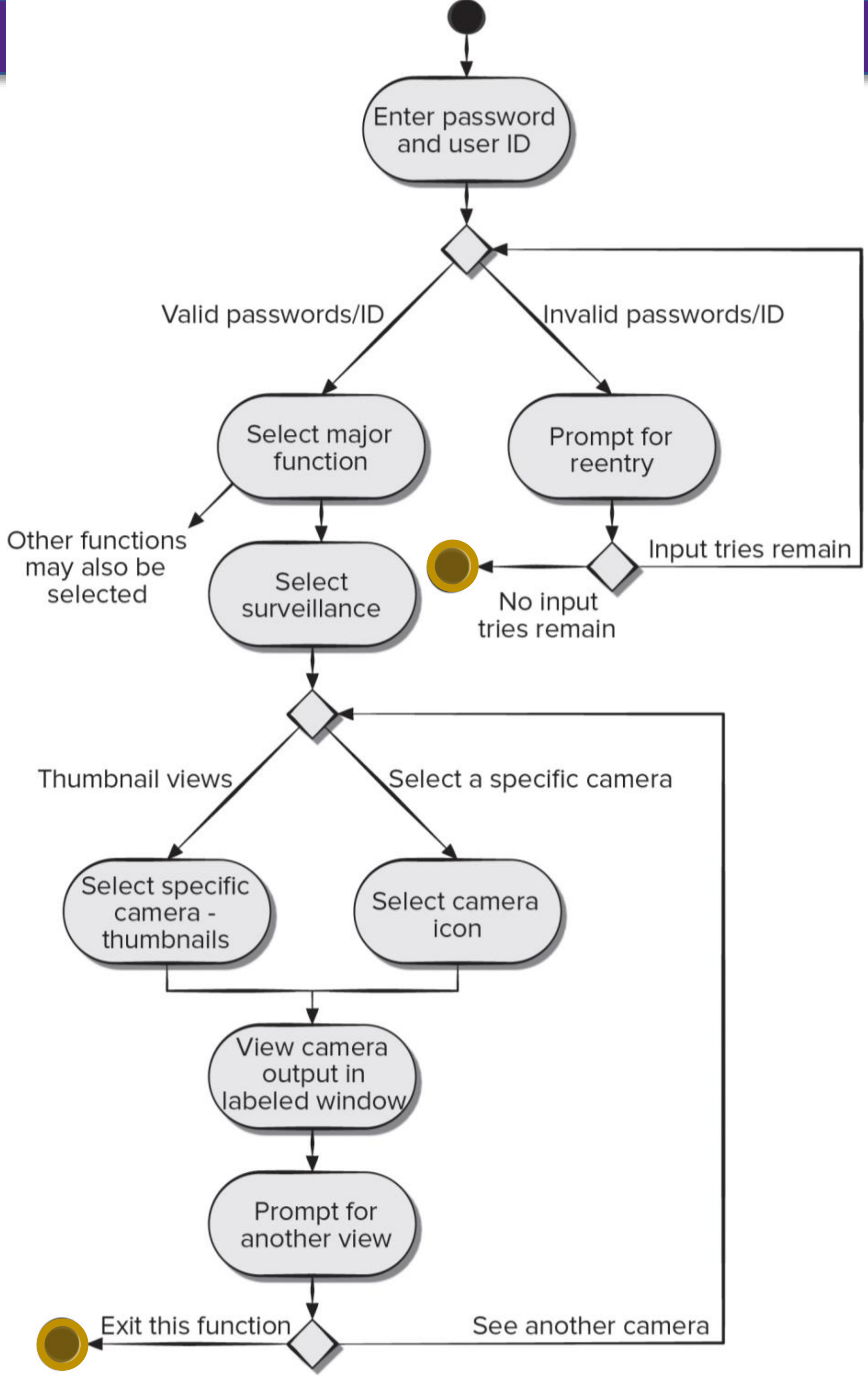
- Initial Node
- Action Node
- **Final Node**
- Decision Node
- Flow
- Fork
- Join

The Final Nodes shown here are Activity Final Nodes, they end an activity as a whole.

There are also Flow Final Nodes that look like this:

⊗

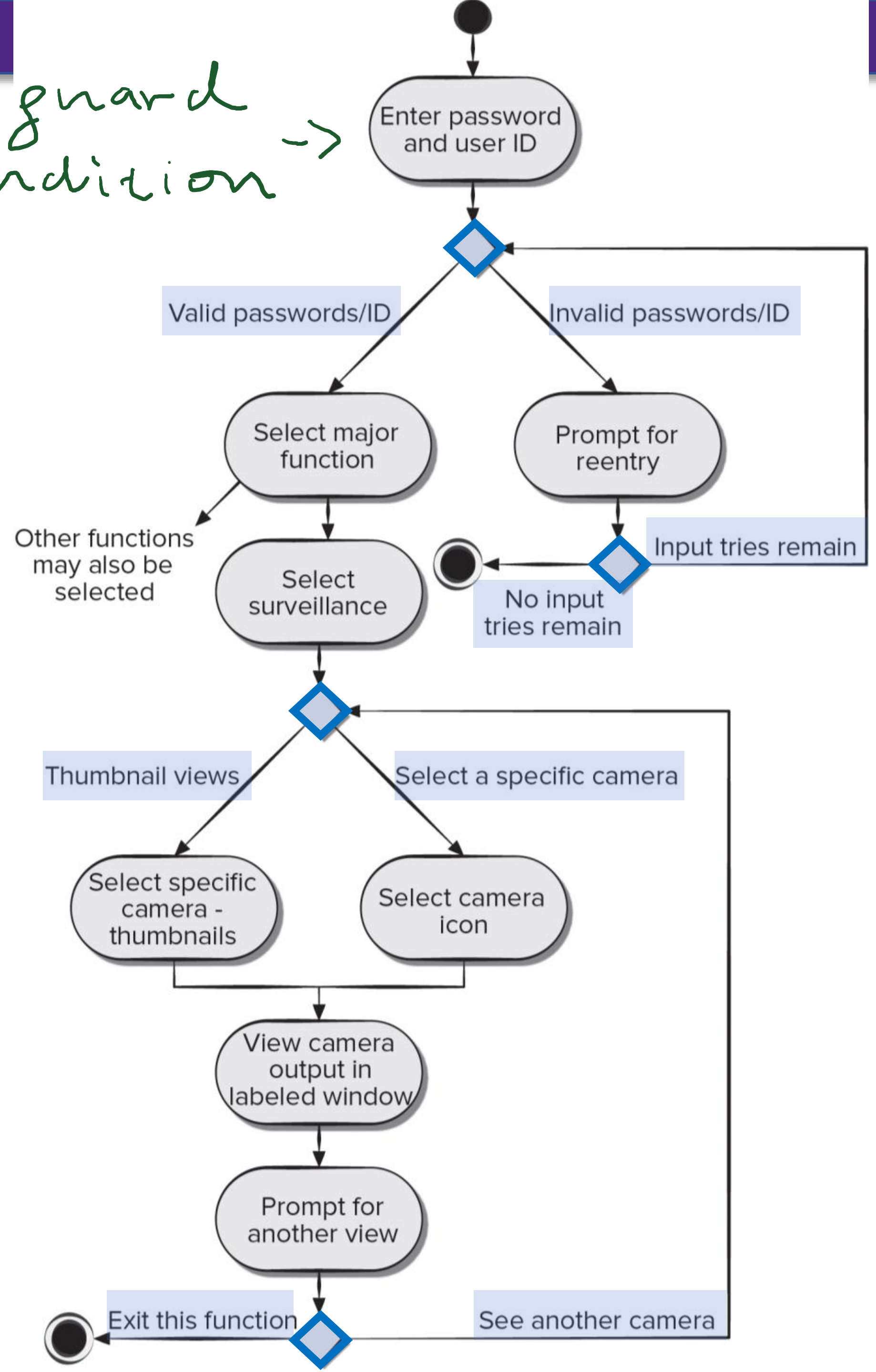
Flow Final Nodes denote the end of the signal of control and not the whole activity.



Activity Diagrams

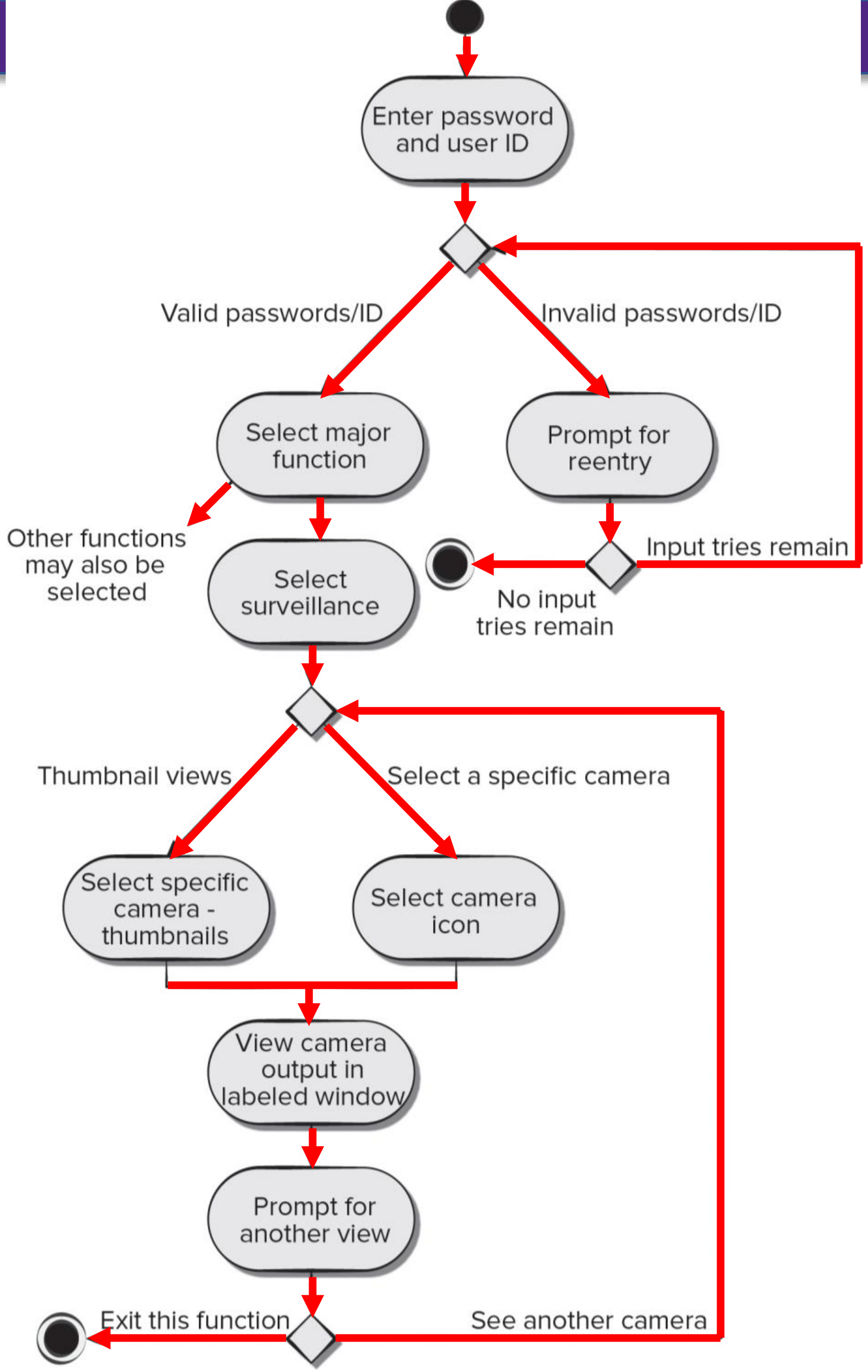
- Initial Node
- Action Node
- Final Node
- **Decision Node**
- Flow
- Fork
- Join

guard condition ->



Activity Diagrams

- Initial Node
- Action Node
- Final Node
- Decision Node
- **Flow**
- Fork
- Join

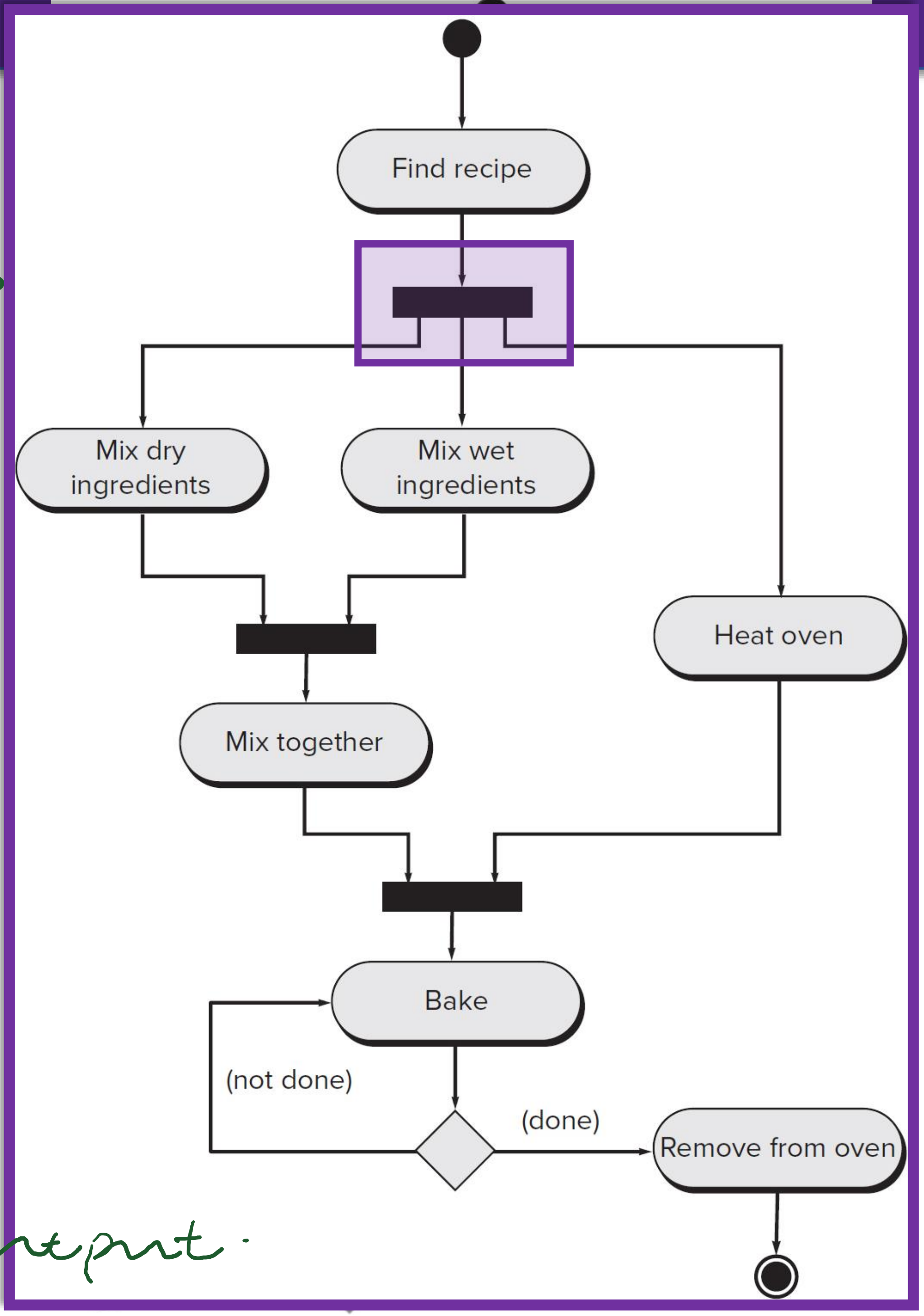


Activity Diagrams

- Initial Node

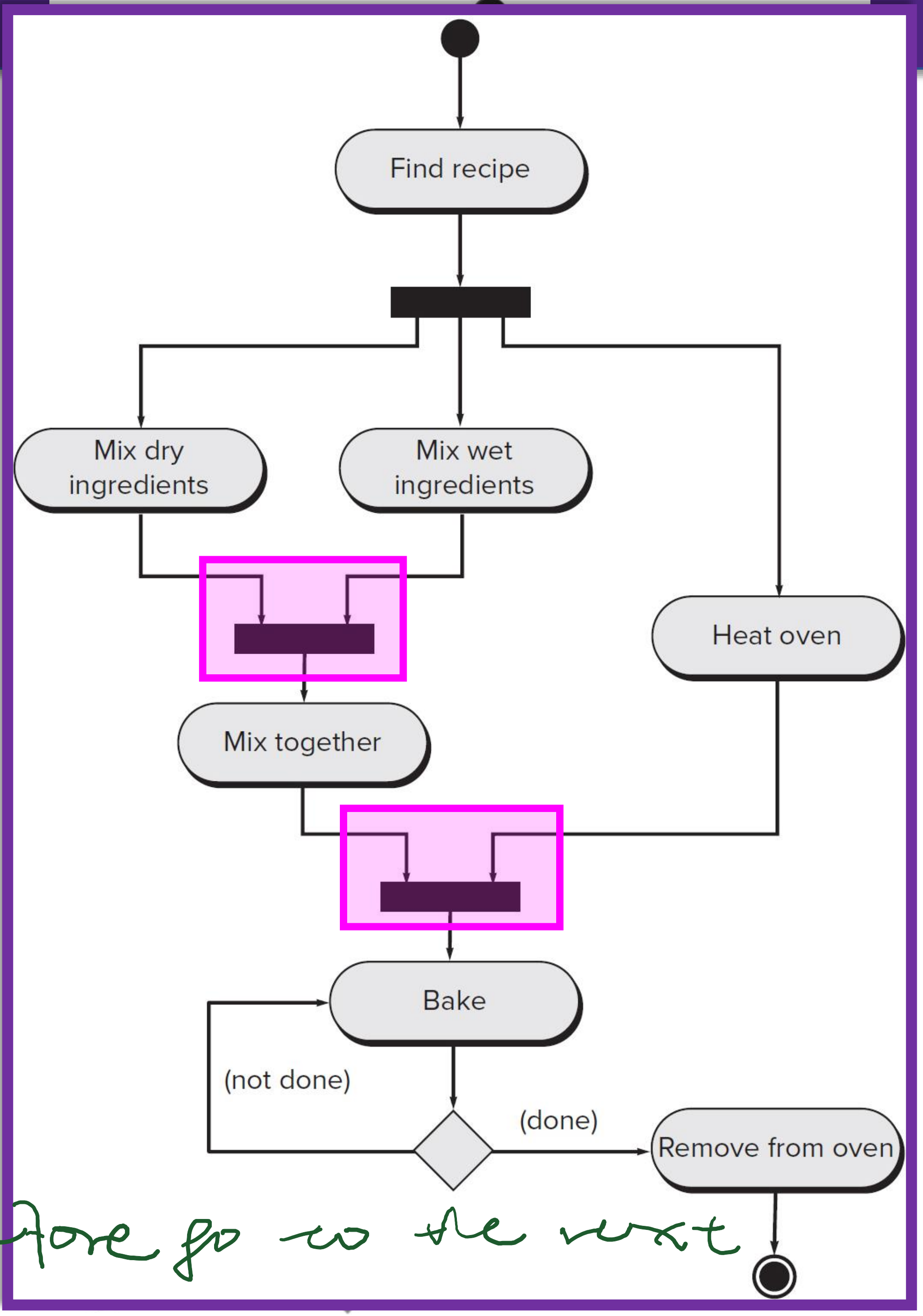
not a actual → system

- Action Node
- Final Node
- Decision Node
- Flow
- **Fork** : go down all steps at the same time ; only one go in but two or more output.
- Join

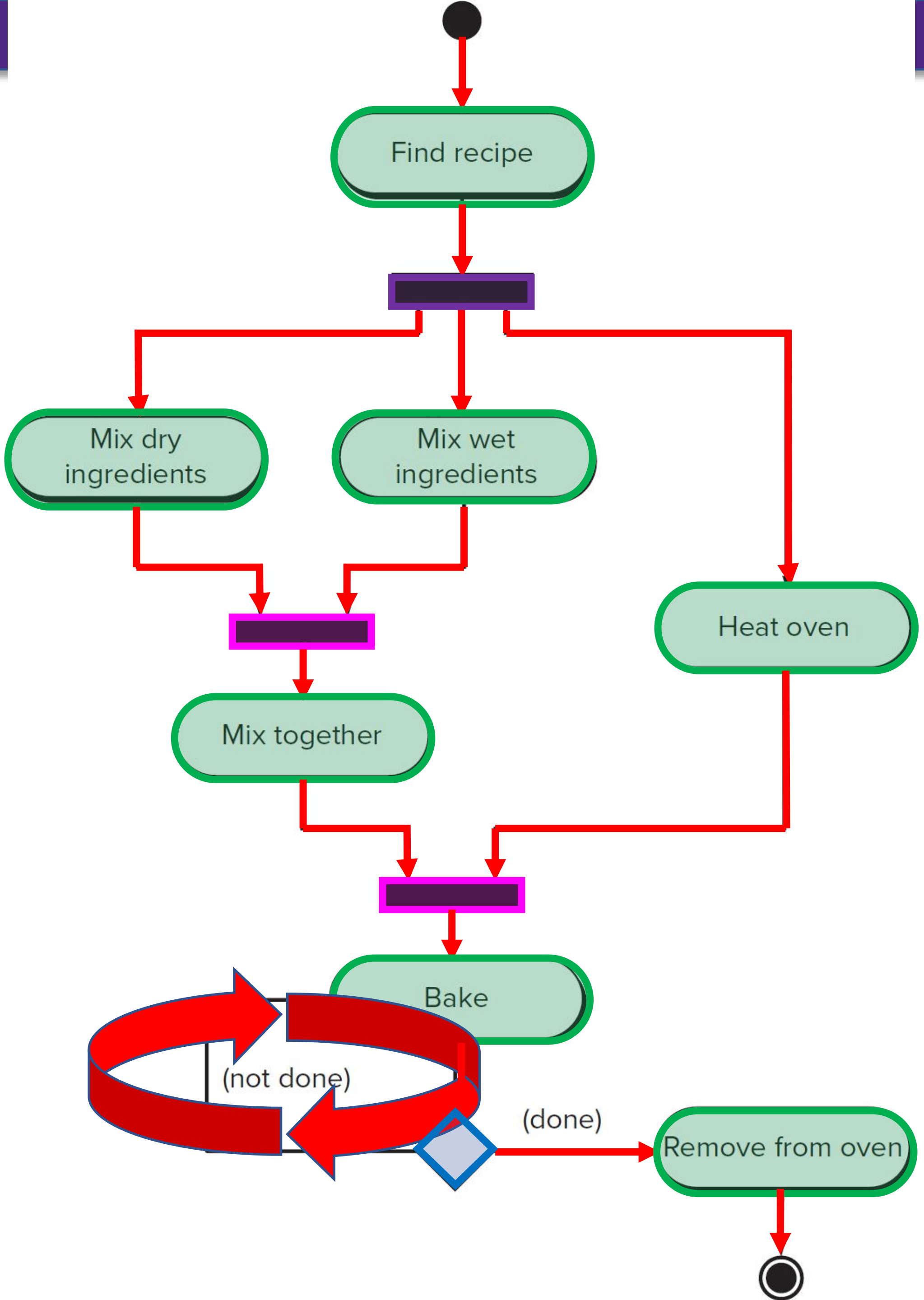


Activity Diagrams

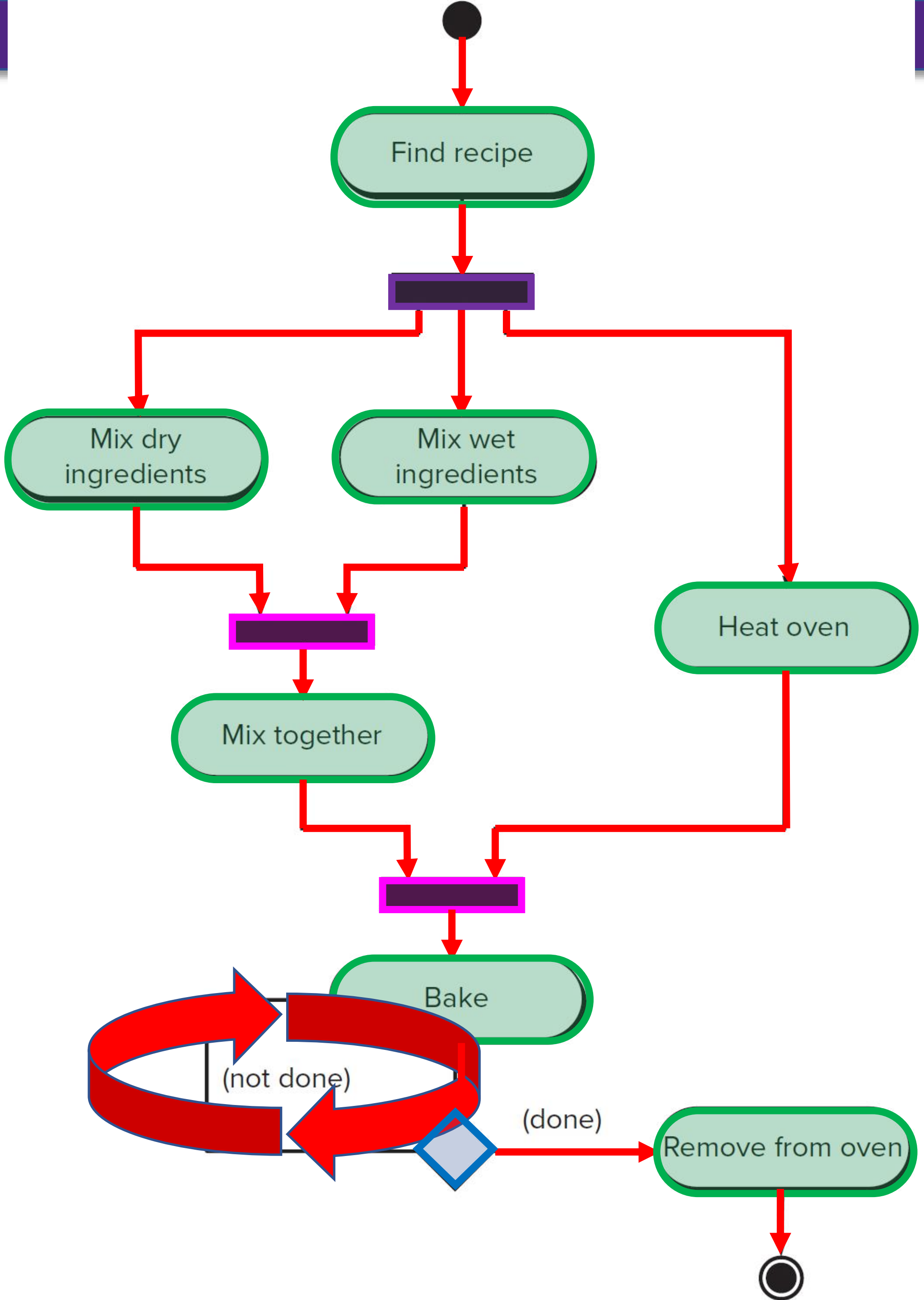
- Initial Node
- Action Node
- Final Node
- Decision Node
- Flow
- Fork
- **Join** wait for both task finish before go to the next



Activity Diagrams Example 2



Activity Diagrams Example 2

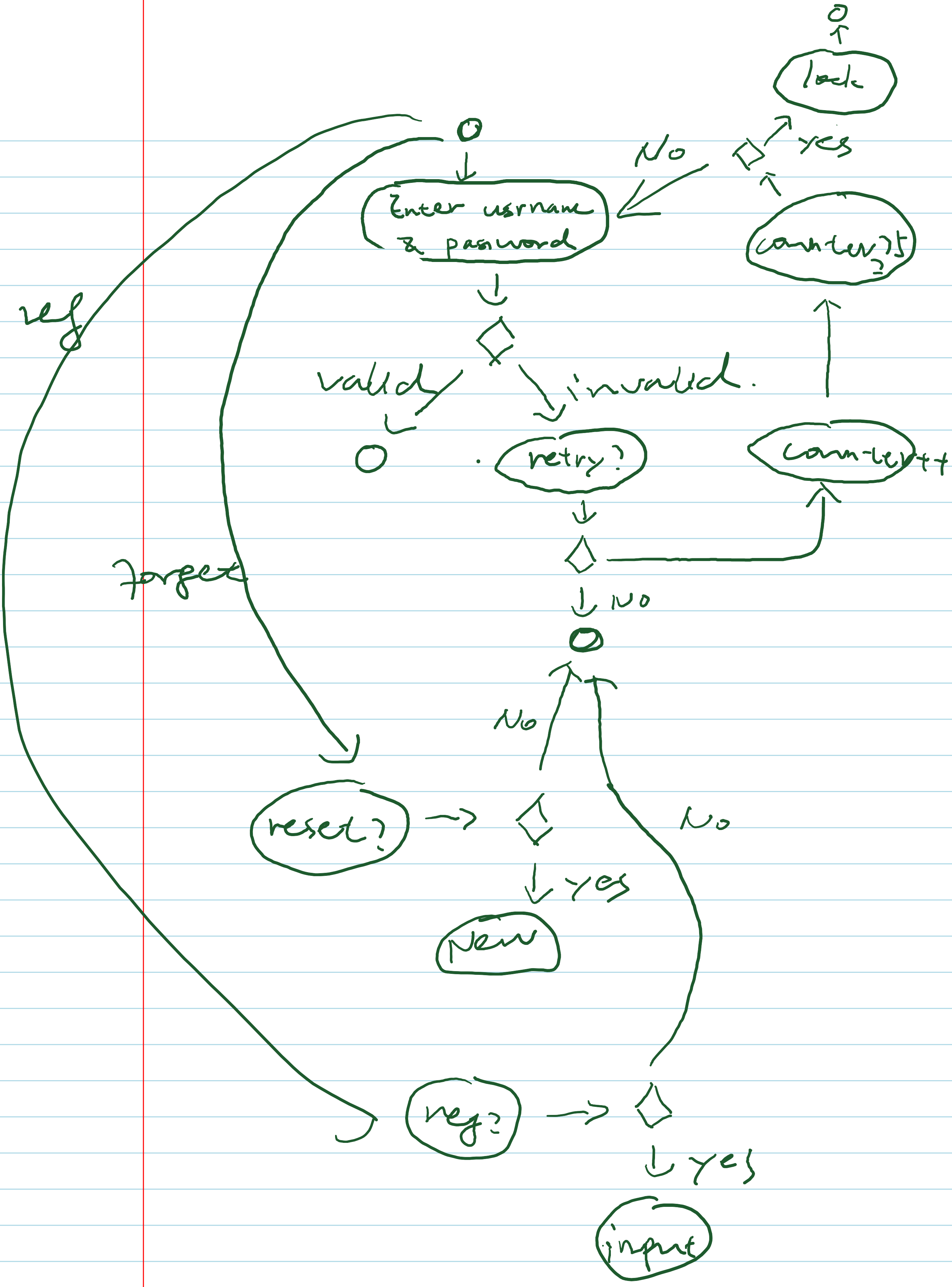


Activity Diagrams Activity 1

Create an Activity Diagram for this scenario.

Fill in any extra details needed.

- Create an activity diagram for a login system that allows users to login to a website by providing a username and password.
- If a username and password is input incorrect 5 times the account is locked.
- If a user does not yet have an account they can register for one by providing a username, password, and email.
- If the user has forgotten their password they can reset it using the forgot password feature and a password reset e-mail is sent (you can represent this by one action).
- You can assume that no actions happen concurrently (don’t need to use forks and joins).



Enter usrname & password
valid
invalid.
retry?
No
counter++
counter>5 ?
yes
No
lock
ref
forget
reset?
No
yes
New
No
reg?
yes
input

Activity Diagrams for Code

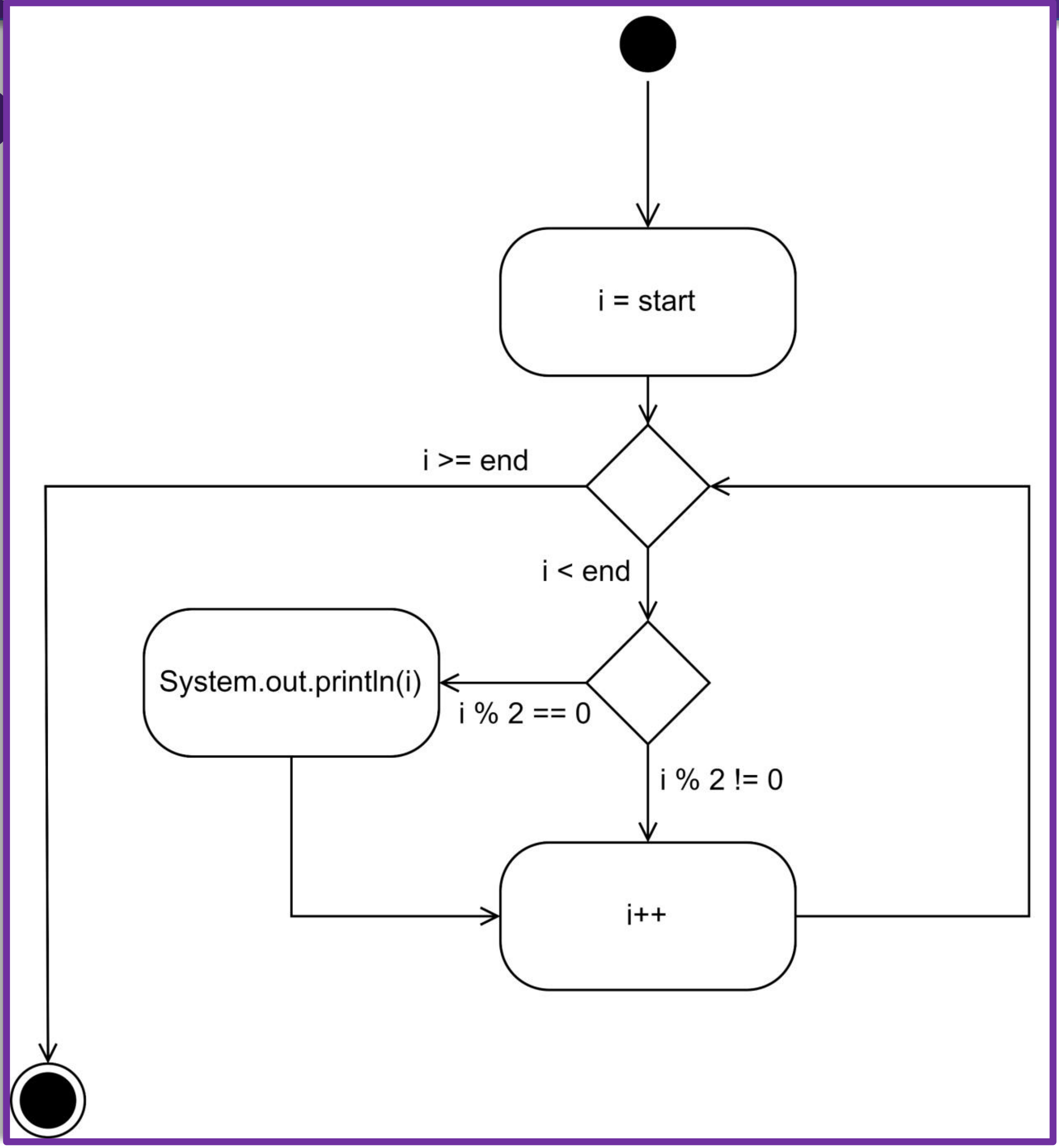
- Like flowcharts activity diagrams can also be used to describe an algorithm or a segment of code.
- **Example:**

```
void printEvenNumbers(int start, int end) {
    for (int i = start; i <= end; i++) {
        if (i % 2 == 0) {
            System.out.println(i);
        }
    }
}
```

Activity Diagrams fo

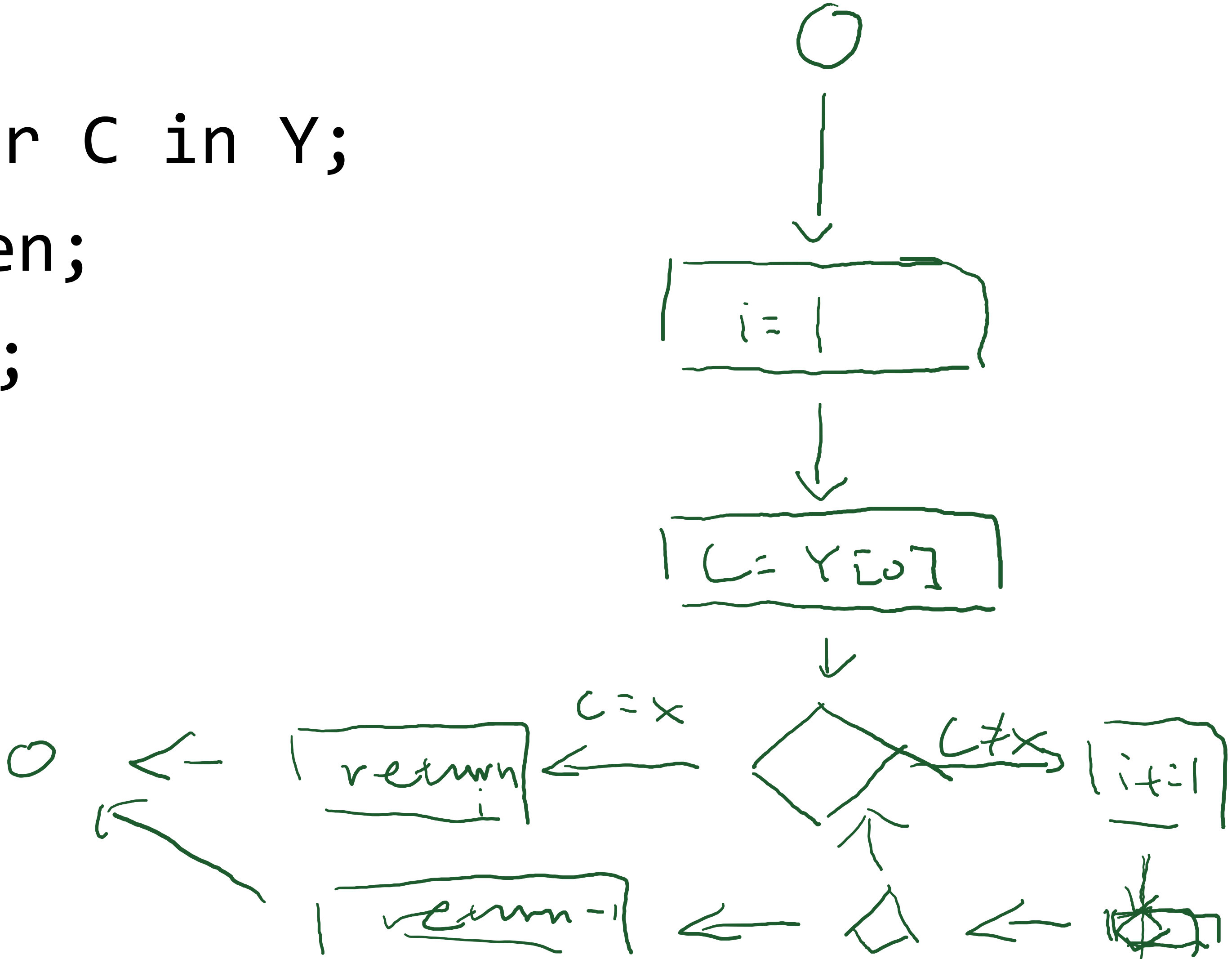
- Like flowcharts activity diagrams can algorithm or a segment of code.
- **Example:**

```
void printEvenNumbers(int start, int end) {
    for (int i = start; i <= end; i++) {
        if (i % 2 == 0) {
            System.out.println(i);
        }
    }
}
```



Activity Diagrams Activity 2: Activity Diagram For Code

```
Procedure find_character(char X, string Y): integer;
    i = 1;
    for each character C in Y;
        if C == X then;
            return i;
        endif;
        i += 1;
    endfor;
    return -1;
end find_character;
```



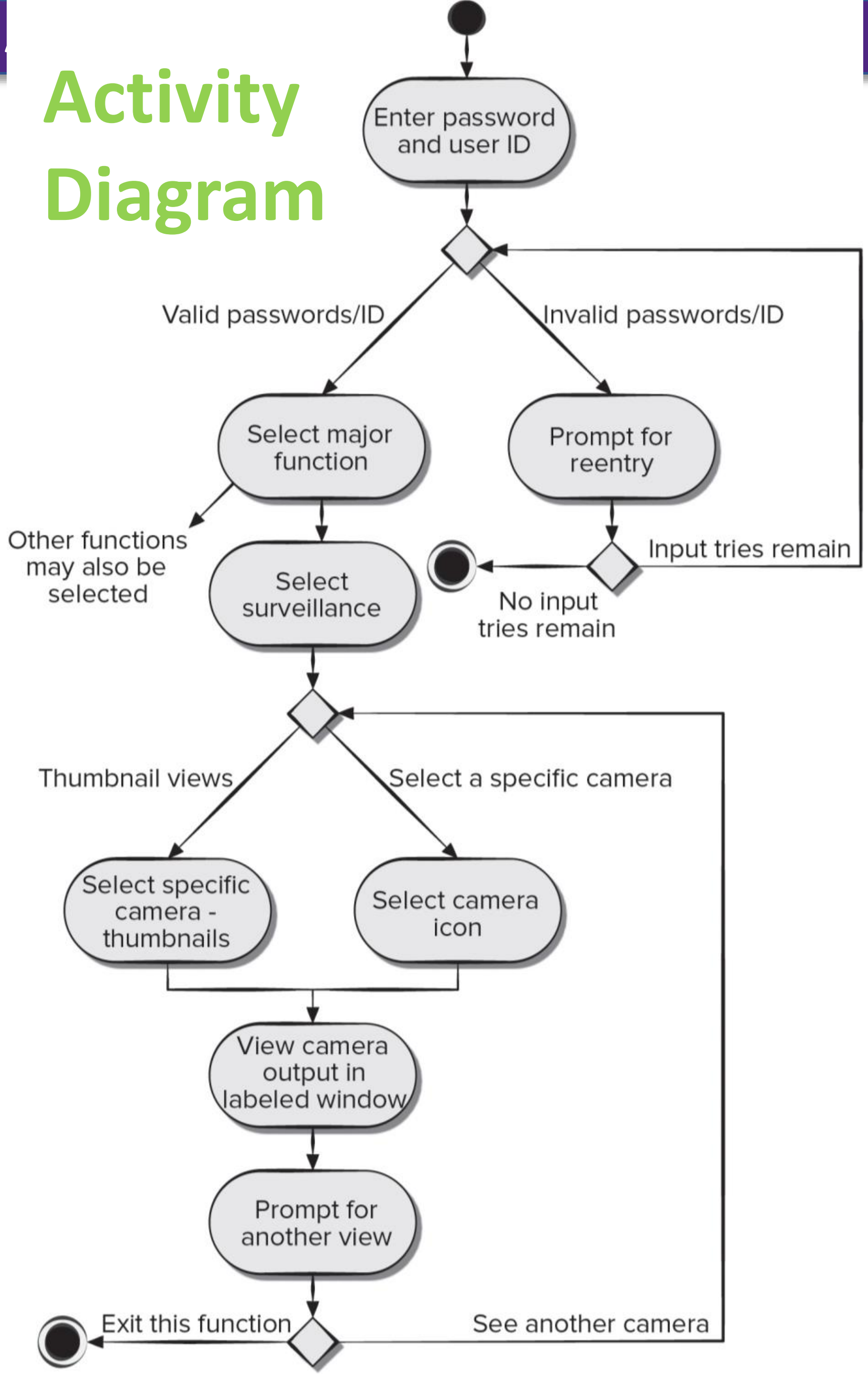
Swimlane Diagrams

- The **Swimlane Diagram** is a useful variation of the **Activity Diagram** that allows you to **represent the flow of activities.**

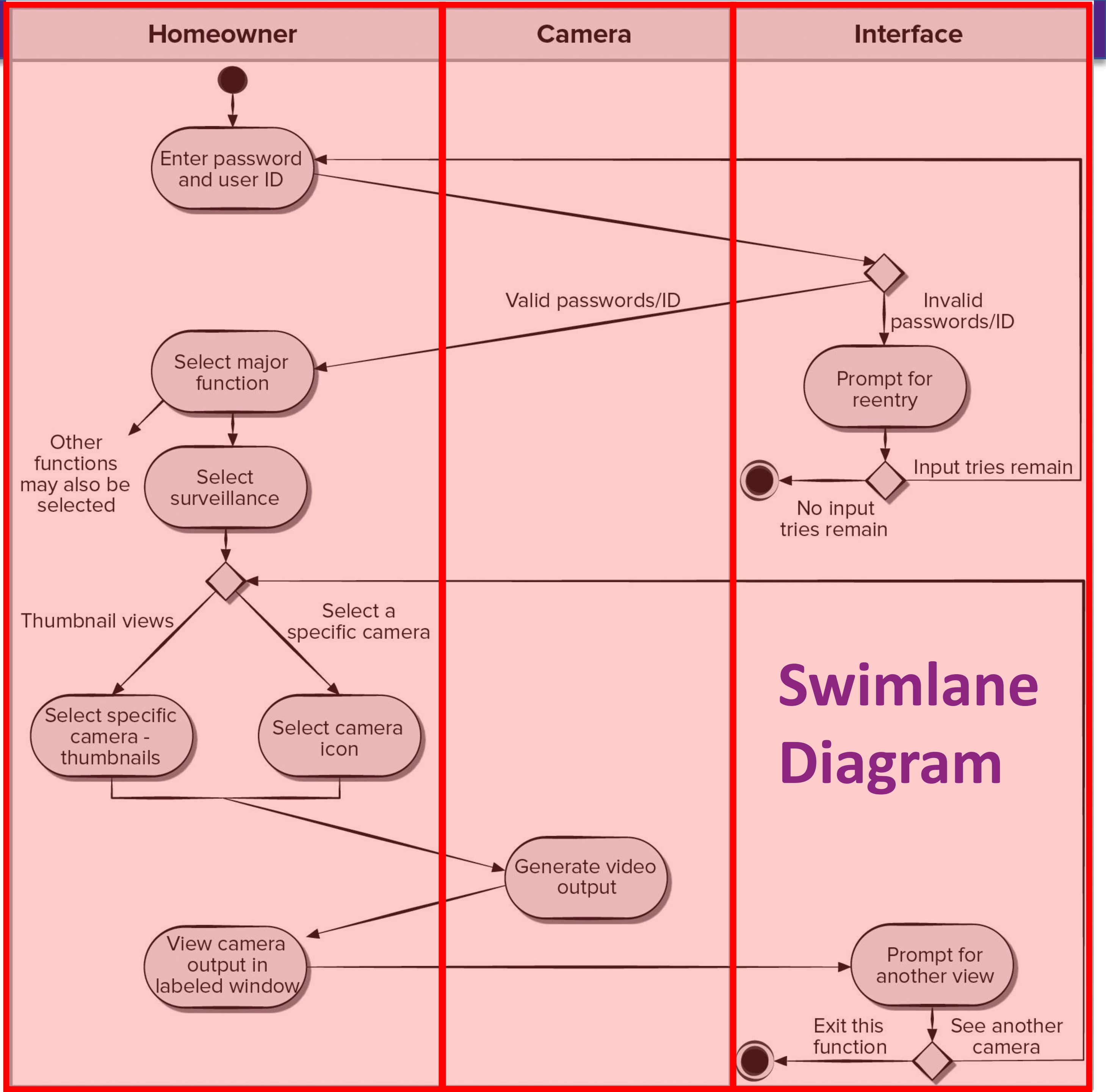
- **Swimlane Diagrams** indicate which **actor** *(if there are multiple actors involved in a specific use case)* or **analysis class** has **responsibility for the action** described by an action node.

- **Responsibilities** are **represented as parallel segments** that divide the diagram vertically, like the lanes in a swimming pool.

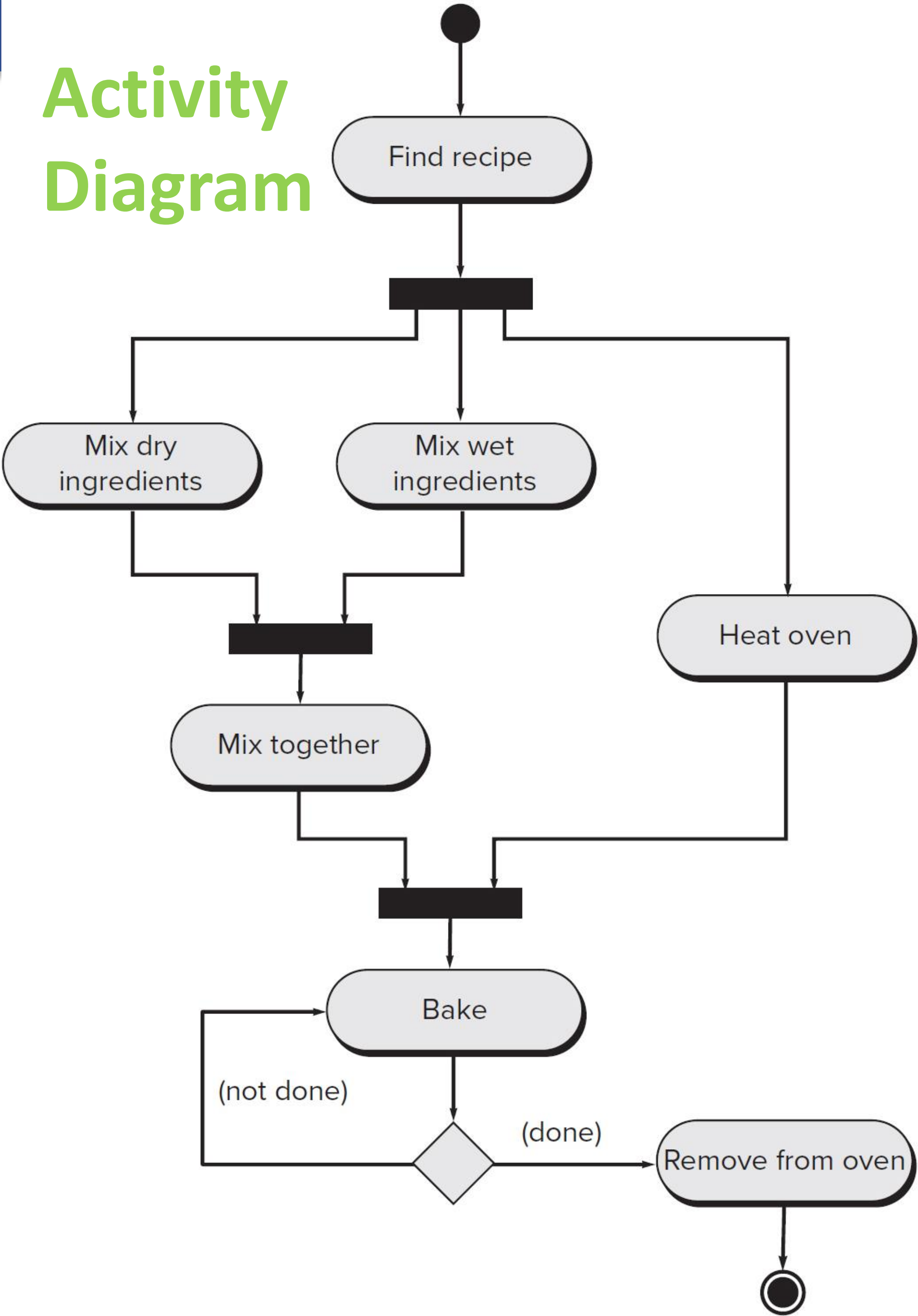
Activity Diagram



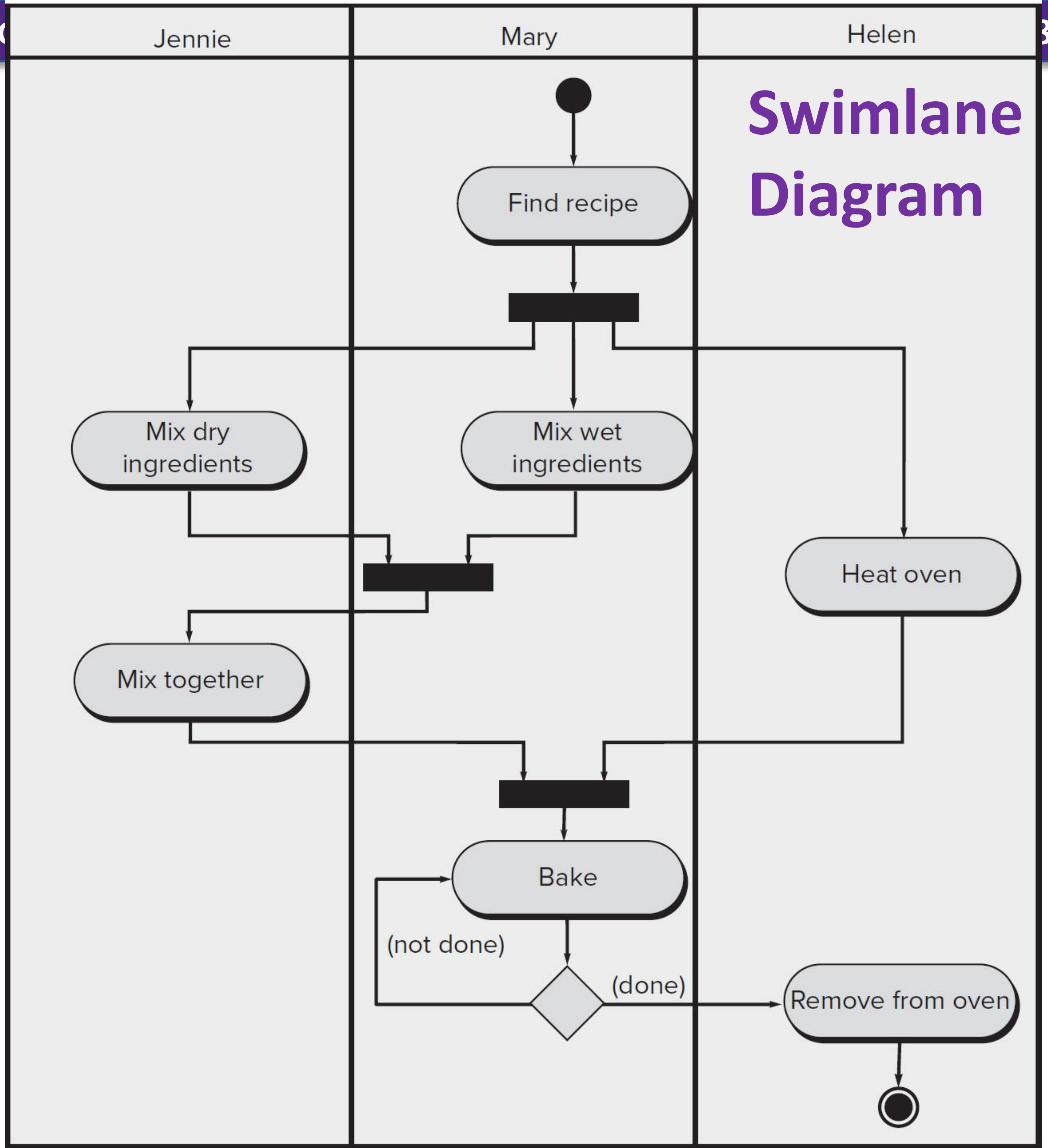
Swimlane Diagram



Activity Diagram



Swimlane Diagram



Week 2

CS2212

Introduction to Software Engineering

UML: Use Case Diagrams

Ask Questions Live

cs1.ca/ask

Some images and content from Software Engineering: A Practitioner's Approach 9th Edition By Roger Pressman and Bruce Maxim

Use Cases

Use Cases:

- A technique for capturing the **functional requirements** of a system.
- Describe typical interactions between the users of a system and the system it's self.
- Expressed in a **narrative form** (often using a fixed template).
- Users are referred to as **actors**, as are external systems.
- **Actors** are a role that the user (or external system) plays in the **use case**.

Example Use Cases: SafeHome

Scenario

Security system where homeowner uses control panel, tablet or cell phone to control and monitor their home security system.

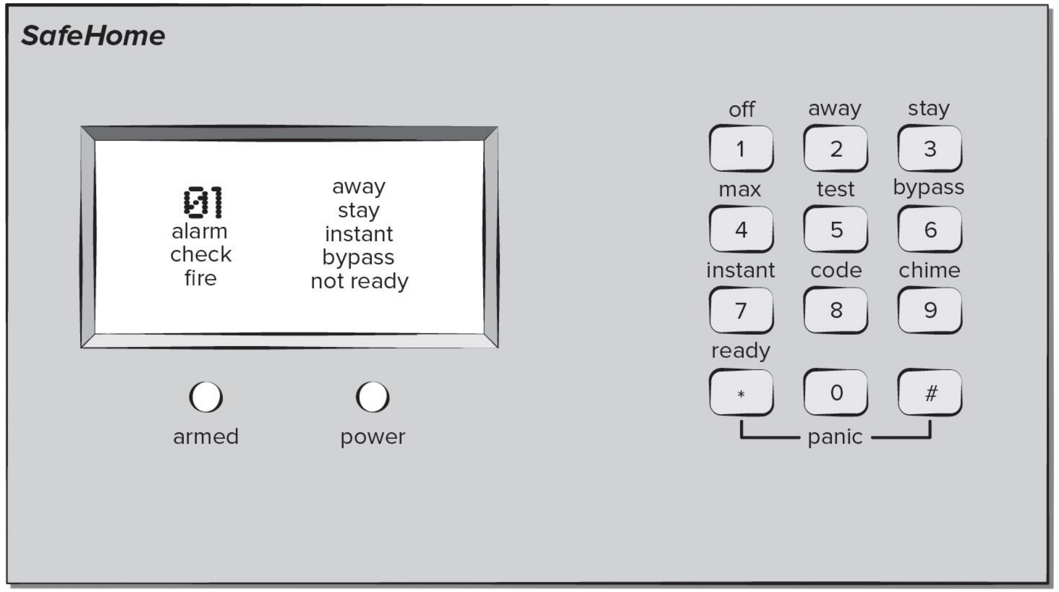
The homeowner:

The homeowner is an actor.

1. Arms/disarms the system with a pin.
2. Access the system remotely via the internet.
3. Responds to alarm events.
4. May encounter and deal with error conditions.

Each of these is it's own Use Case.

Control Panel



Example Use Cases: SafeHome

Scenario

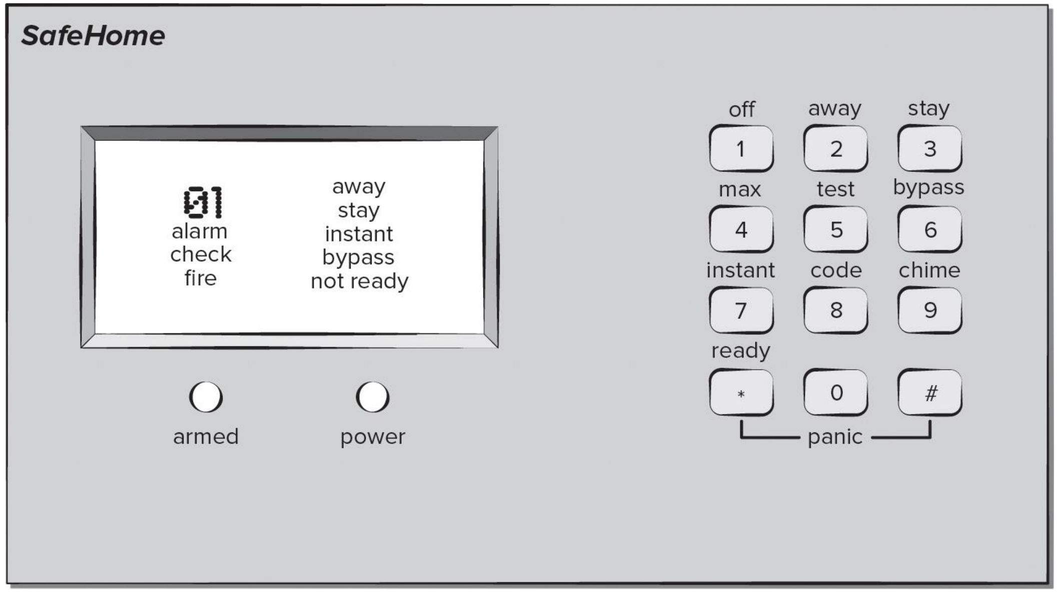
Security system where homeowner uses control panel, tablet or cell phone to control and monitor their home security system.

The system administrator:

1. Reconfigures sensors and related functions.

2. May also be a homeowner and can do all things a homeowner can.

Control Panel



Example Use Cases: SafeHome

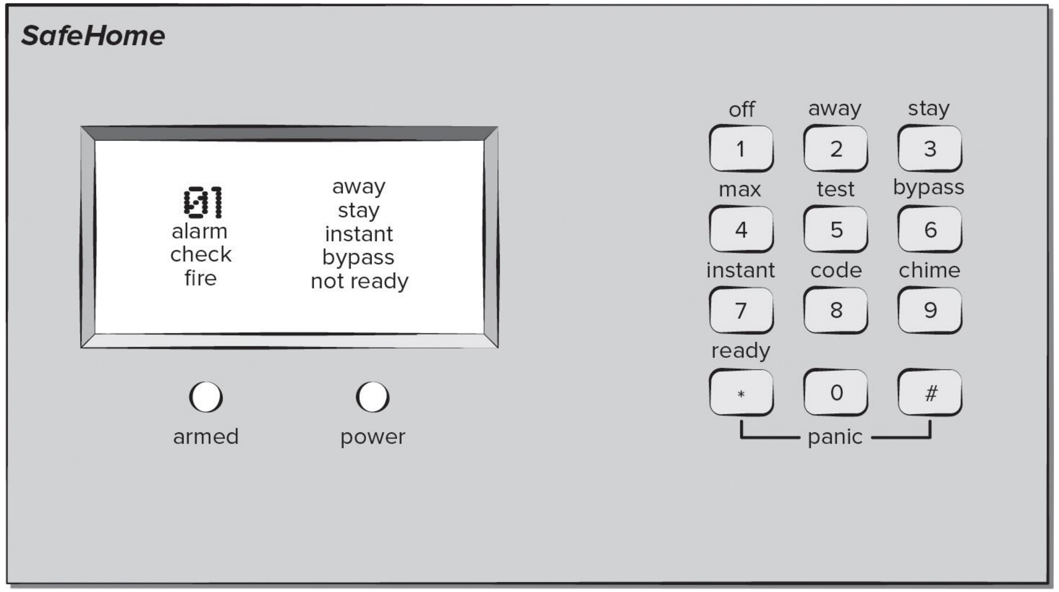
Scenario

Security system where homeowner uses control panel, tablet or cell phone to control and monitor their home security system.

Sensors:

1. Respond to alarm events.
2. May encounter and deal with error conditions.
3. Are configured and reconfigured by a system administrator.

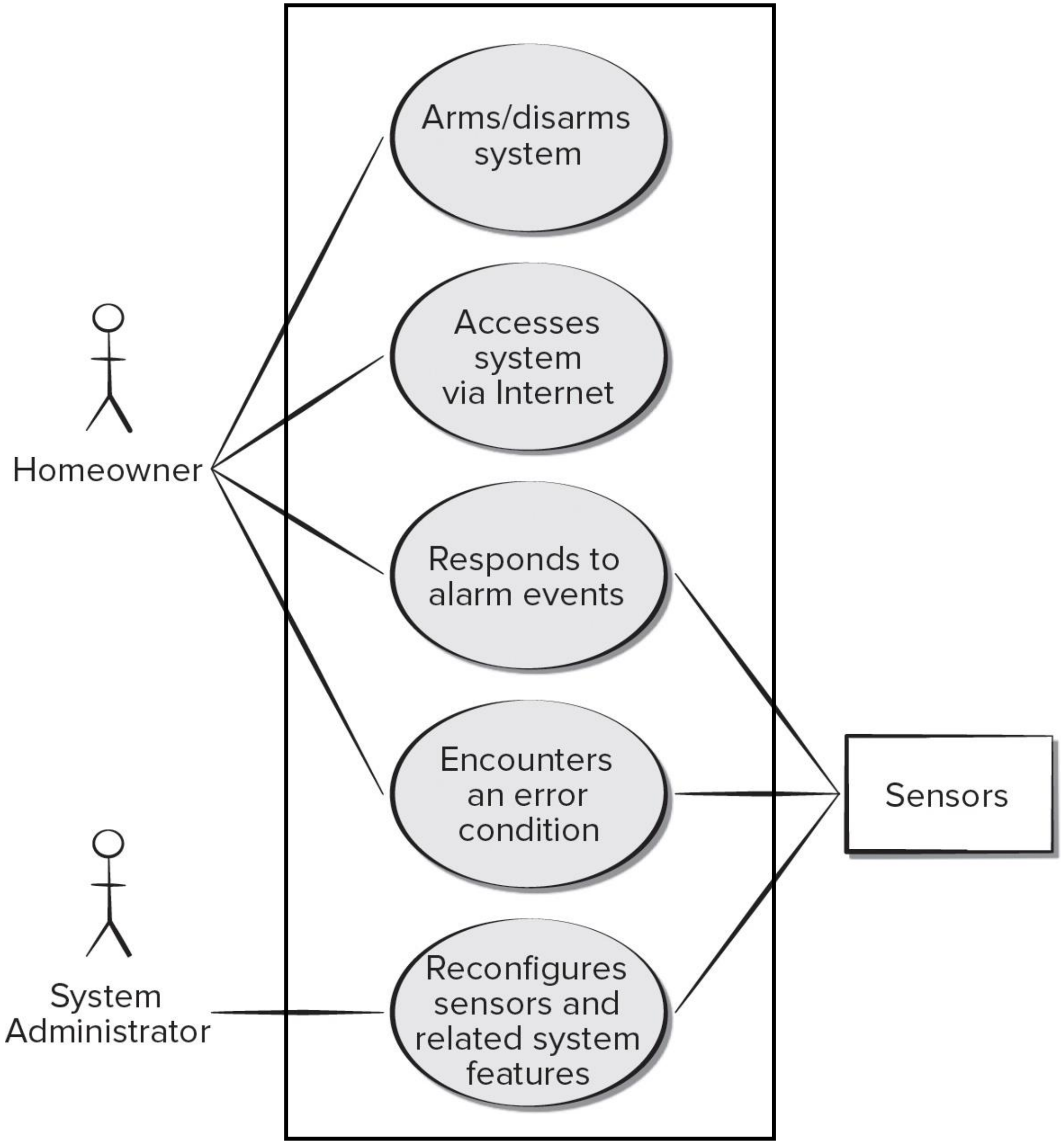
Control Panel



Use Case Diagrams

- Visually depict the relationships between **use cases** and **actors**.
- Act as a graphical table of contents for the **use case** set.
- Show system boundary and connections with outside world.
- Denotes:
  - Which actors carry out which use cases.
  - Which use cases include other use cases.

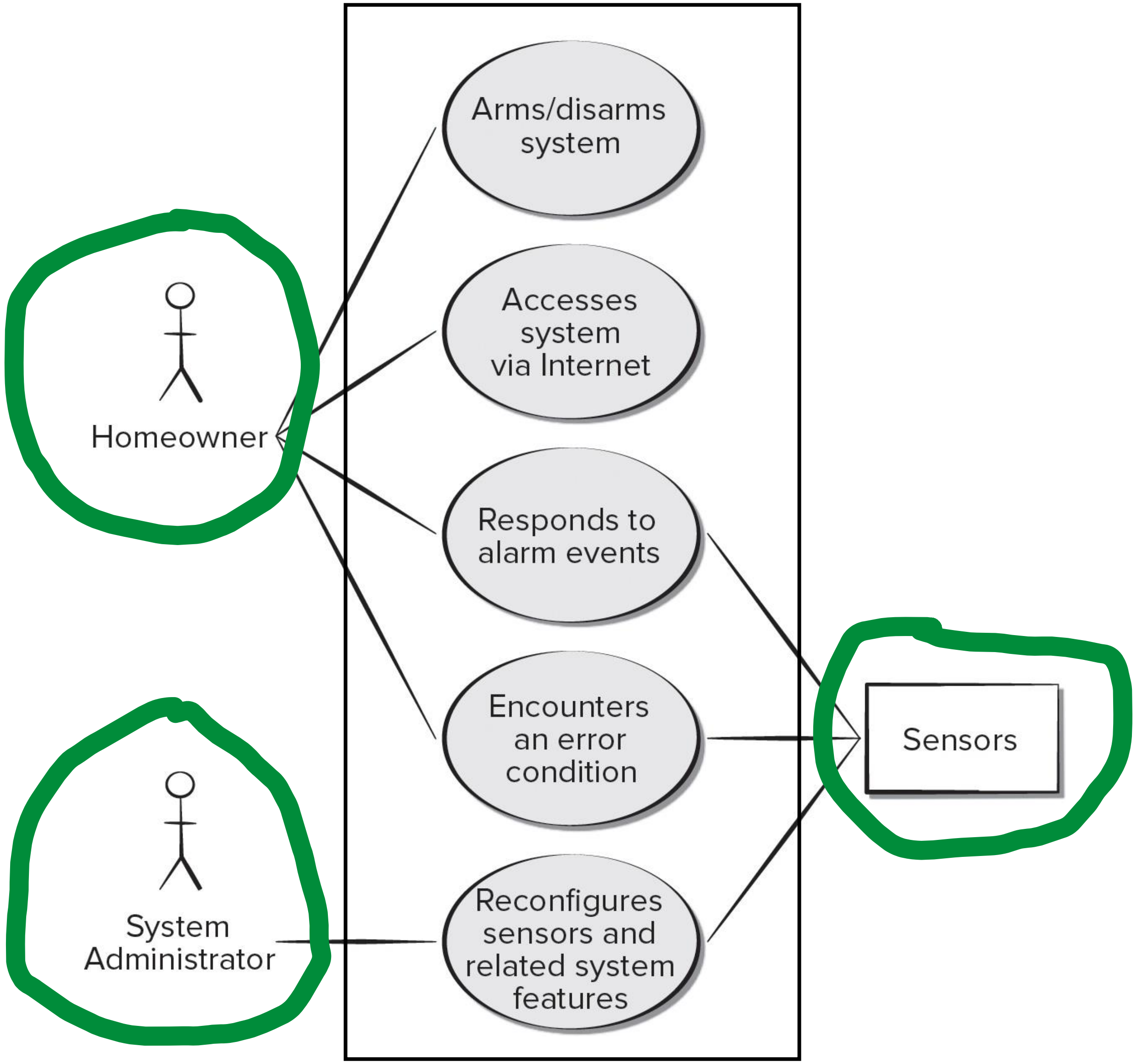
Example Use Case Diagram: Safe Home


Arms/disarms system
Accesses system via Internet
Homeowner
Responds to alarm events
Encounters an error condition
Sensors
System Administrator
Reconfigures sensors and related system features

Example Use Case Diagram: Safe Home

Actors

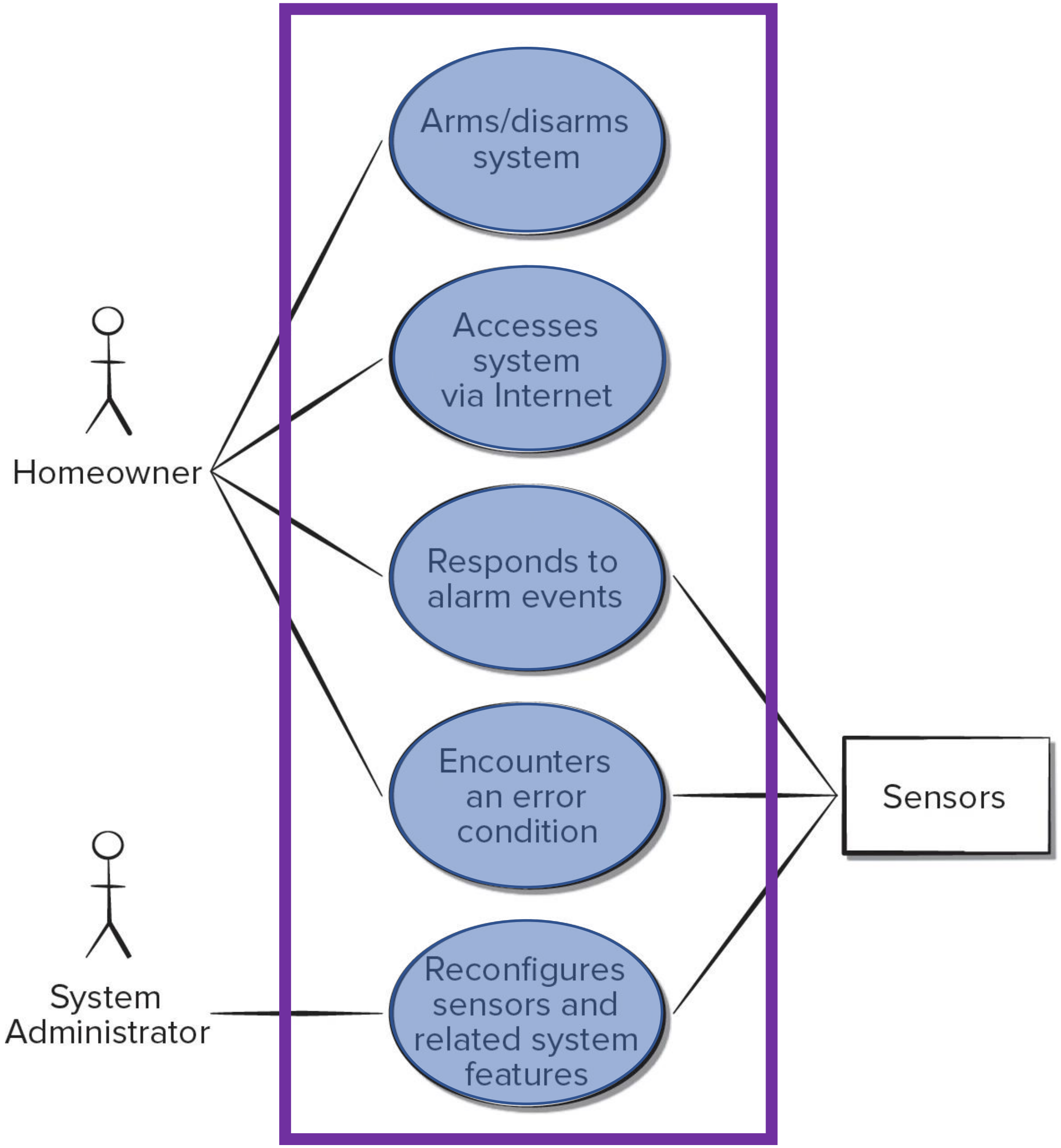
- Stick figure in a use case diagram represents an actor that is associated with one category/role of user or other element that interacts with the system.
- Complex systems may have many actors.
- In some cases, non-person actors are represented by different symbols (e.g., sensor is shown in a box). This is nonnormative but common.


Homeowner
System Administrator
Arms/disarms system
Accesses system via Internet
Responds to alarm events
Encounters an error condition
Reconfigures sensors and related system features
Sensors

Example Use Case Diagram: Safe Home

Use Cases

- Use cases are displayed as ovals.
- The actors are connected by lines to the use cases that they carry out.
- Use cases are placed in a rectangle, but the actors are not; the rectangle represents the **boundaries of the system**.


Arms/disarms system
Accesses system via Internet
Responds to alarm events
Encounters an error condition
Reconfigures sensors and related system features
Homeowner
System Administrator
Sensors

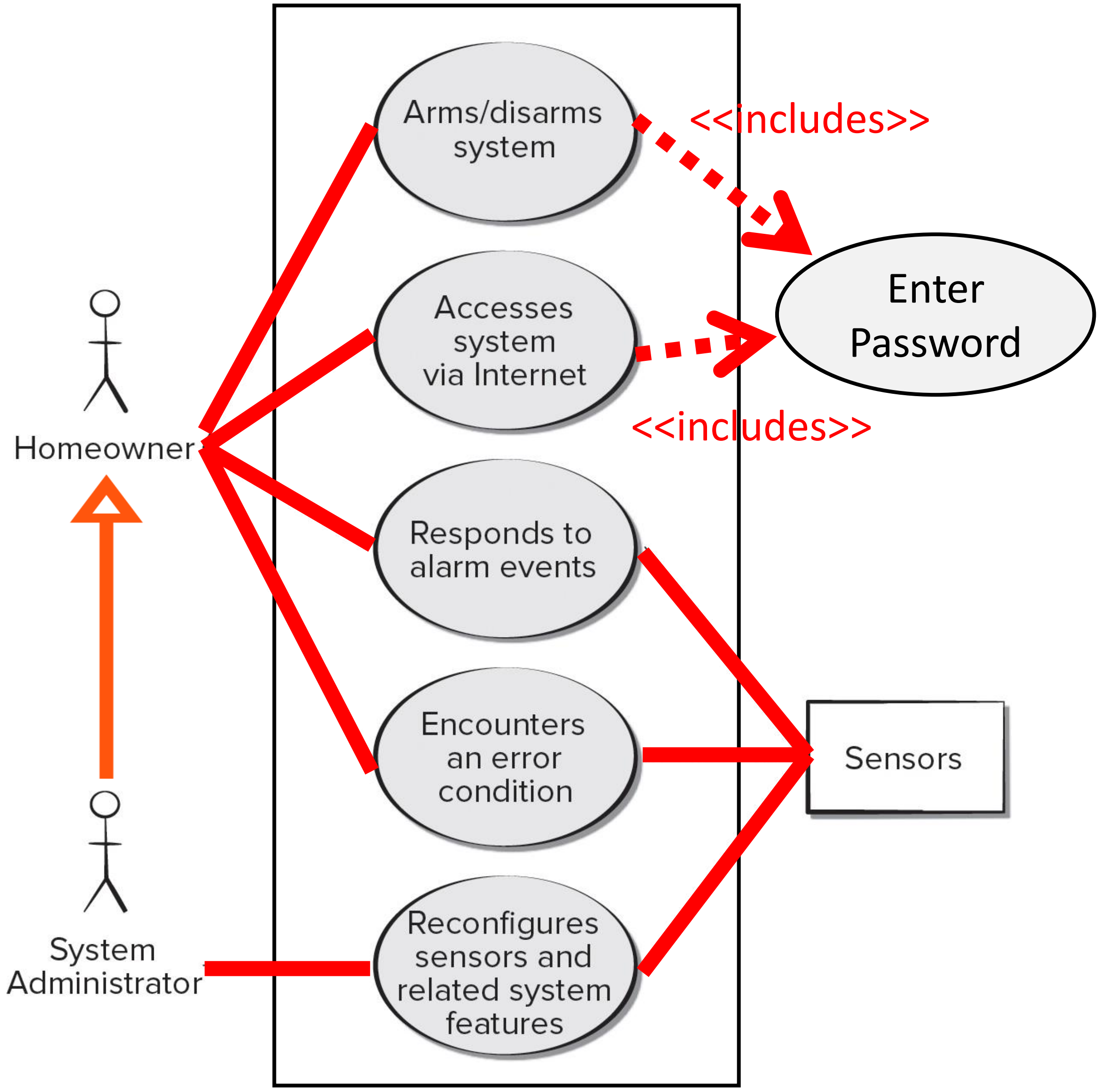
Example Use Case Diagram: Safe Home

Relationships

Communicates

Includes

Generalizes



Final Thoughts on Use Case Diagrams

- Use case diagrams are helpful in ensuring that you have covered all of the functionality of the system as you get to see the system as a whole.

- Note that **none of the details of use cases are included in the diagrams**, however, and such details need to be stored separately.

- These details are still important to the software development process, and are often considered to be more important than the overall use case diagram.

- Important not to over complicate a use case diagram. <<*includes*>> should also be added only if they make the diagram simpler, not more complex.

Use Case Diagrams Activity

Create an Use Case Diagram for this scenario.

Fill in any extra details needed.

- Create a Use Case Diagram for the CS1 ASK Tool.
- This is the tool we use in class to ask/answer questions live in-class, enter groupwork codes, enter participation tickets, and view our participation stats.
- Your instructor uses this tool to generate groupwork and participation keys, view participation statistics for a given student, view class participation statistics as a whole, and view incoming questions/answers during class.
- Assume that the CS1 ASK Tool communicates with an external system (OWL) when validating codes and viewing participation statistics.
- Brainstorm what actors and use cases are involved in this scenario and then create a Use Case Diagram.
- You do not have to detail or explain the Use Cases, just name them.