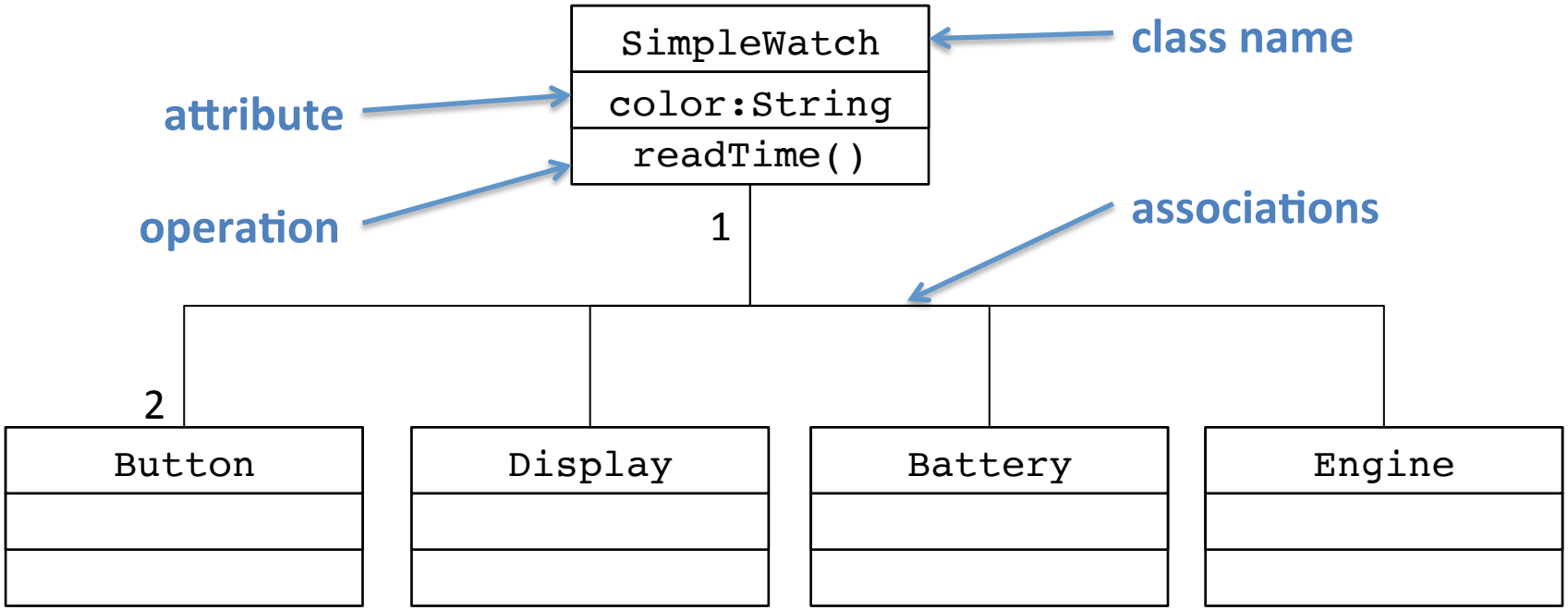
Object-Oriented Analysis and Design

Isara Anantavrasilp

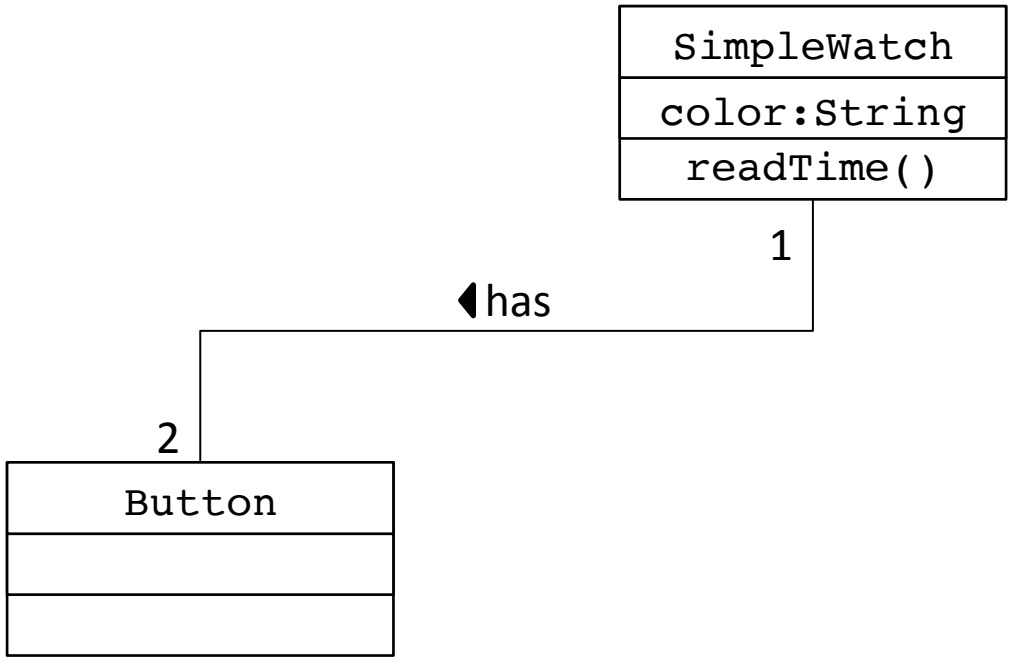
Lecture 8: Modeling with Classes (2)

Class Diagram Revision

- Describes the structure of the system in terms of classes and objects



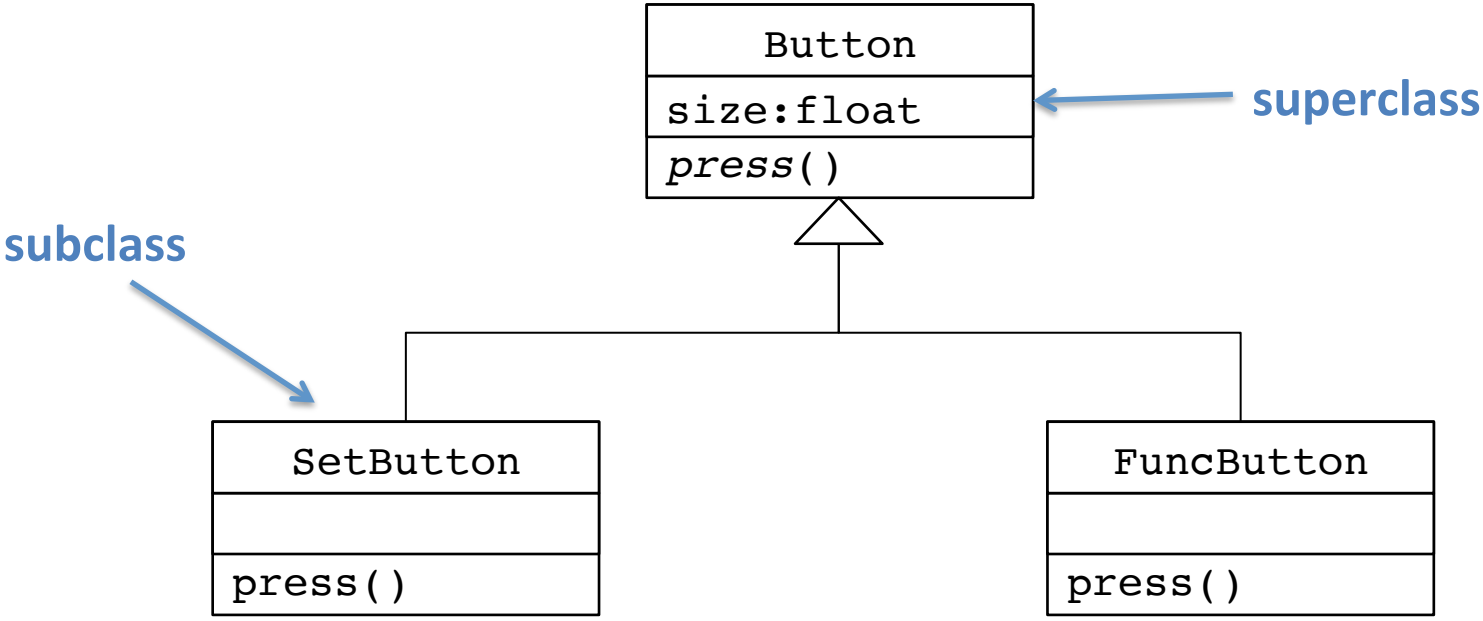
Associations



- Can specify multiplicity
- Can have directions

Inheritance

- "is-a" relationship
- A **subclass** assumes attributes and operations of the **superclass**

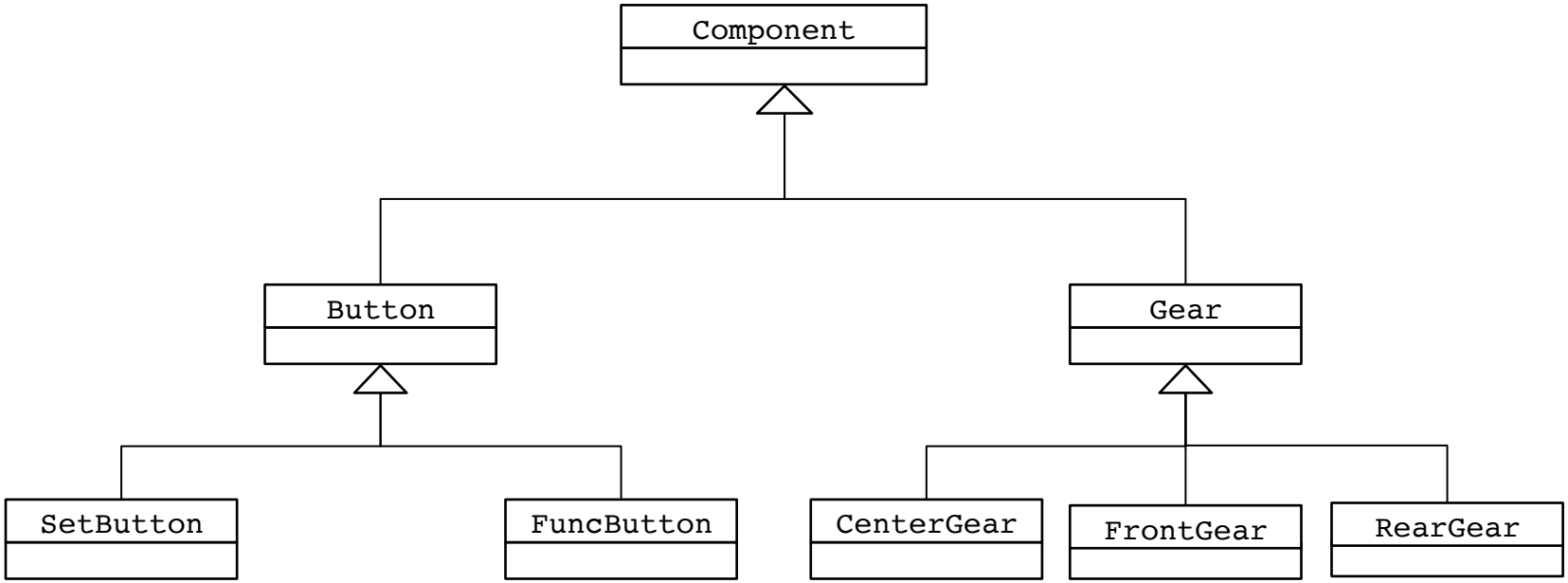


Using Inheritance

- Inheritance is used to achieve two different goals
  - **Description of Taxonomies**
  - **Interface Specification**
- **Description of taxonomies**
  - Used during requirements analysis
  - **Activity**: identify application domain objects that are hierarchically related
  - **Goal**: make the analysis model more understandable
- **Interface specification**
  - Used during detailed design ("object design")
  - **Activity**: identify the signatures of all identified objects
  - **Goal**: increase reusability, enhance modifiability and extensibility.

Describing Taxonomy

- Inheritance is typically used to identify application domain objects that are **hierarchically** related



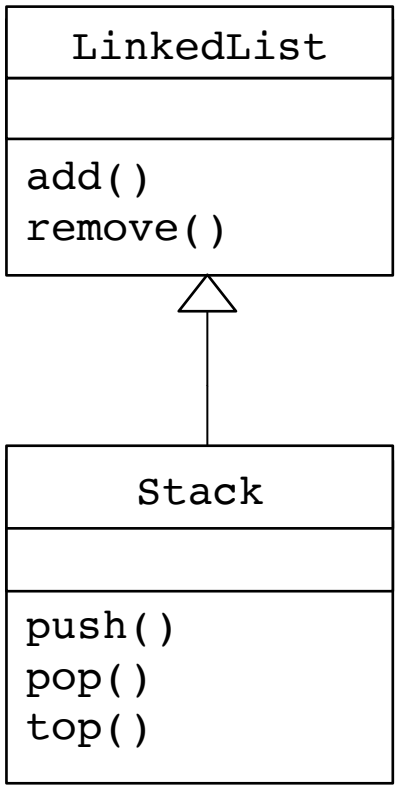
Specifying Interface

- The interface of the superclass is completely inherited
- **Implementation inheritance**: The combination of inheritance and implementation
  - Implementations of methods in the superclass are inherited by any subclass
  - Reference implementations
- **Specification inheritance**: The combination of inheritance and specification
  - Implementations of the superclass (if there are any) are **not** inherited.

Implementation Inheritance

- Subclass is inheriting the superclass' methods implementations
- What the superclass can do, the subclass can do exactly the same too

- I have a LinkedList class
- I want a Stack class
- So, I subclass Stack from LinkedList class and implement push(), pop() and top() using the existing implementations of add() and remove()



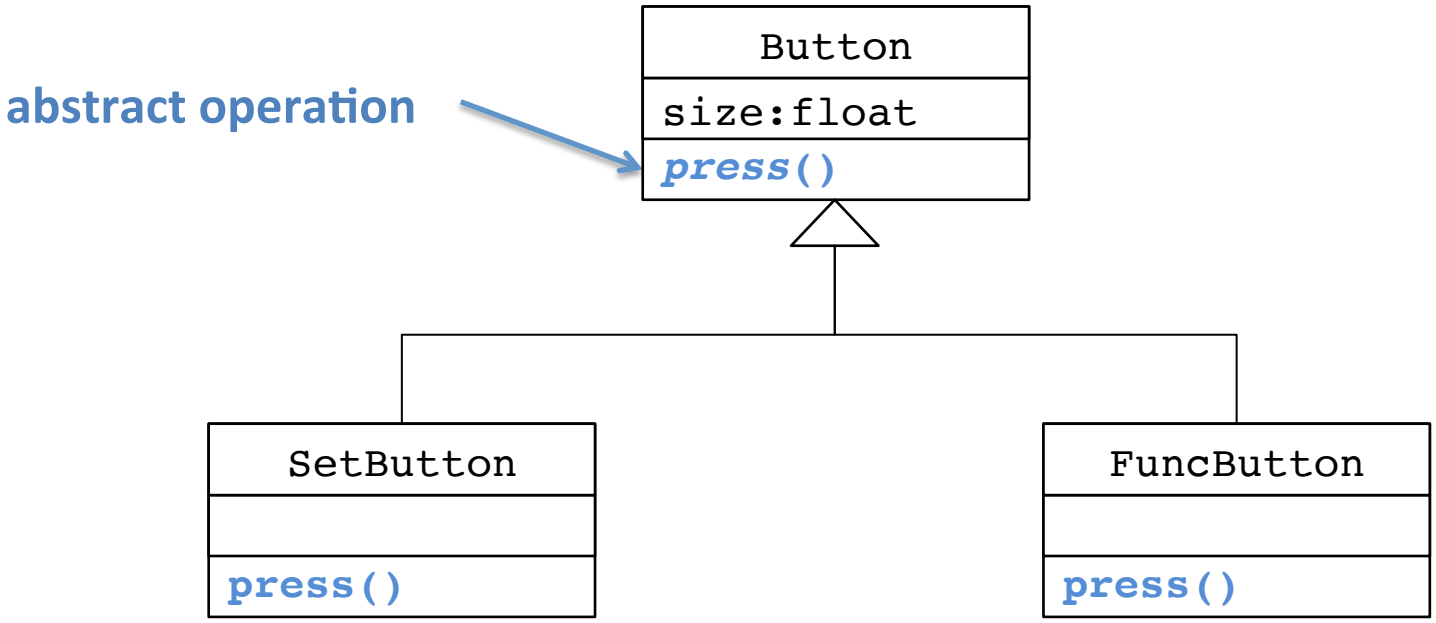
Problem:
- Undesired behavior
- Example: user can call remove() instead of pop()

Specification Inheritance

- Inherit only the signatures of the class
- Implementations are not inherited
  - **Abstract operation**: A method with a signature but **without an implementation. Also called abstract method**
  - **Abstract class**: A class which contains at least one abstract operation
  - **UML Interface**: An abstract class which has only abstract operations
    - The interface is primarily used for the specification of the system or subsystem
    - The implementation is provided by subclassing or by other mechanisms.

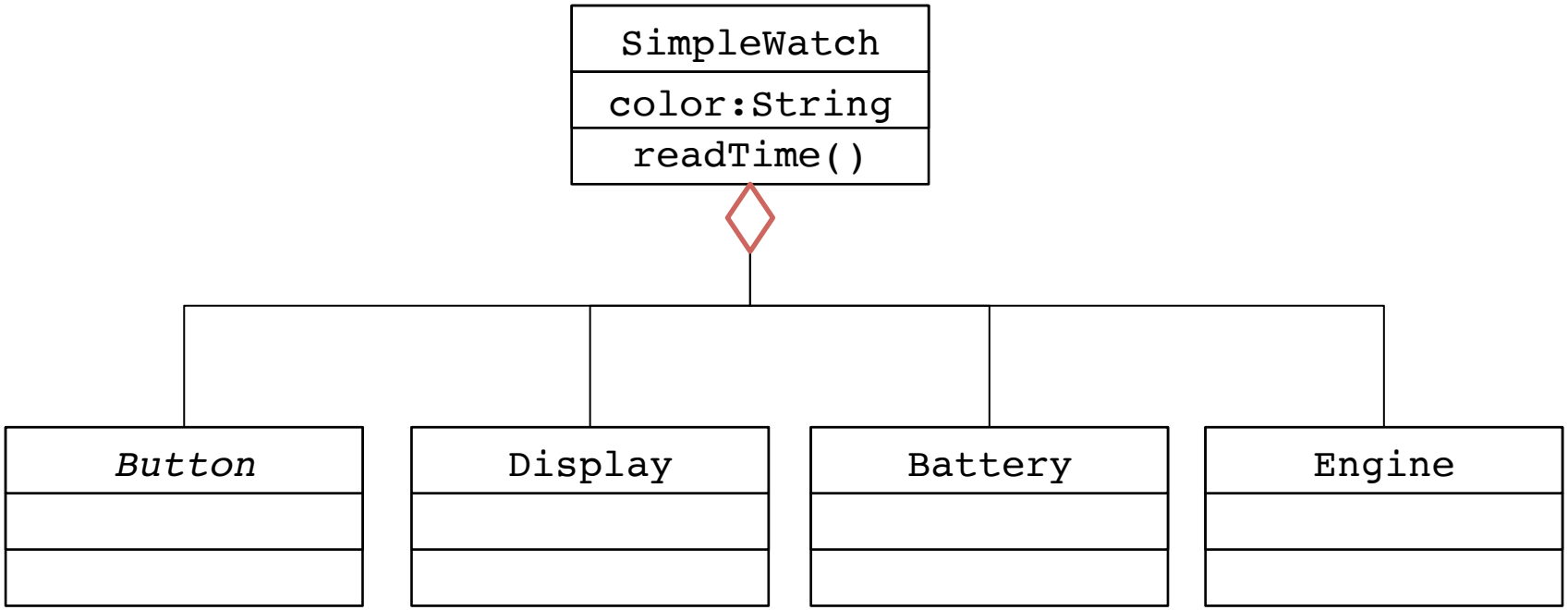
Abstract Operation

- Abstract operation
  - Denoted with Italics in UML
  - It must be implemented in each subclass



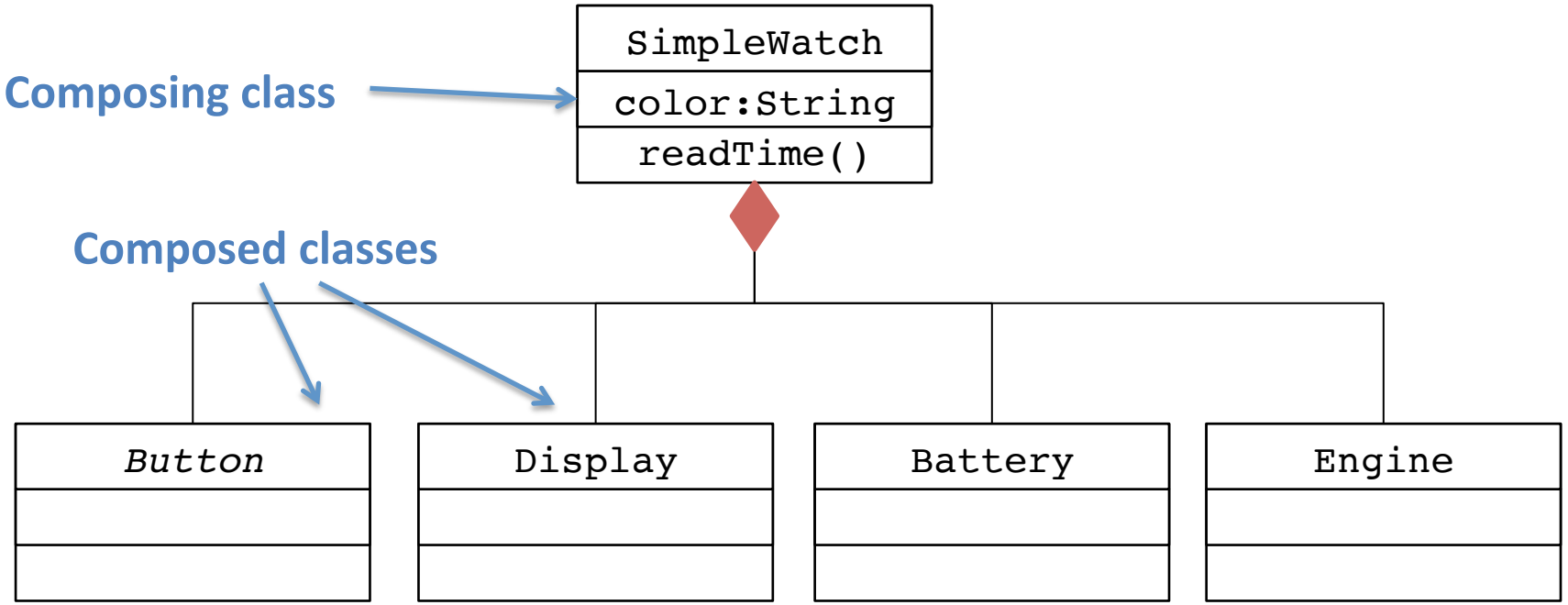
Aggregation

- A type of association
- “belongs to” relationship
- Aggregation indicates structural inclusion of objects of one class by another class



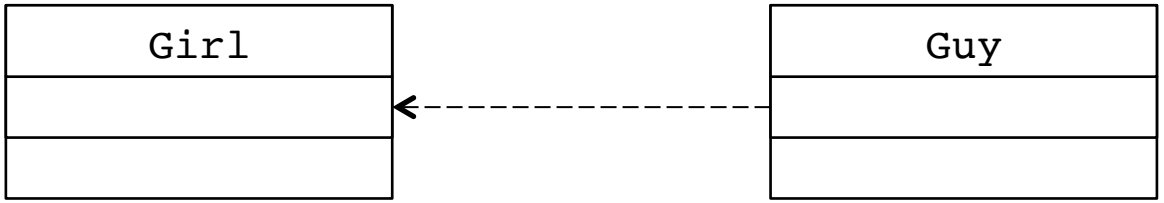
Composition

- Stronger form of aggregation
- “part-of” relationship
- Composed object is created/destroyed when composing object is created/destroyed
- Composed instance is not shared by other classes



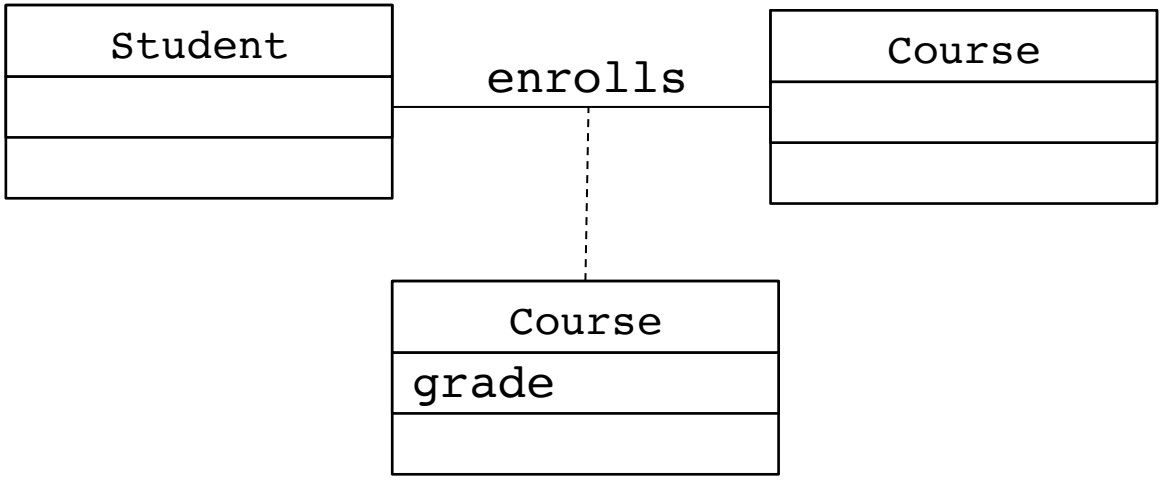
Dependency

- Class A depends upon class B means that if class B is changed, class A must also be affected
- Not vice-versa: B does not care about A
- A guy “depends on” a girl
  - When she changes her mind, he has to change to
  - When he changes his mind, she does not care



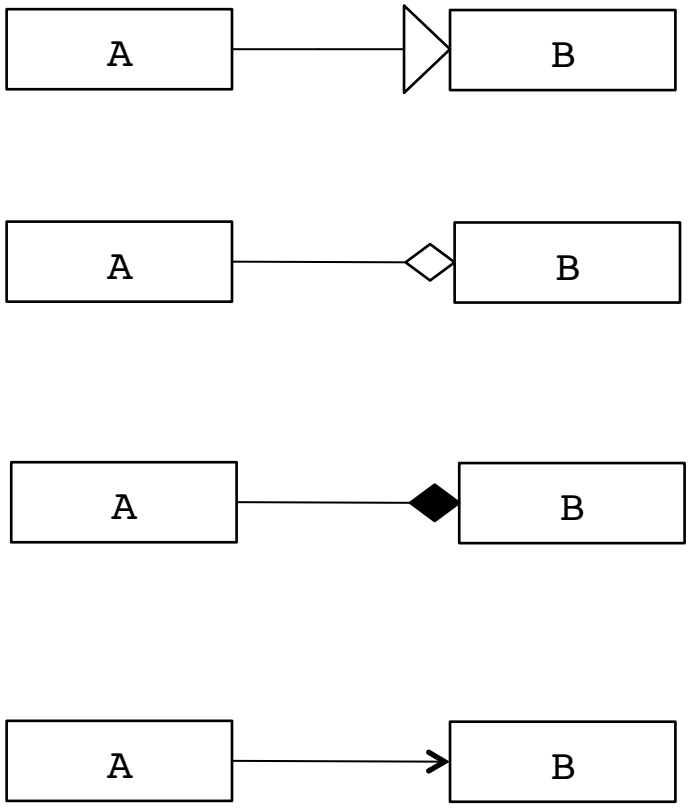
Association Class

- You can also model the "association" between classes
  - To provide additional information (behavior and state) about the relationship
  - Example: `Student-`***`enrolls`***`-Course`
  - What if we need to store grade of each student in each course?
- **Association class**: Class that represents to association between classes
  - In other word, this class is the association

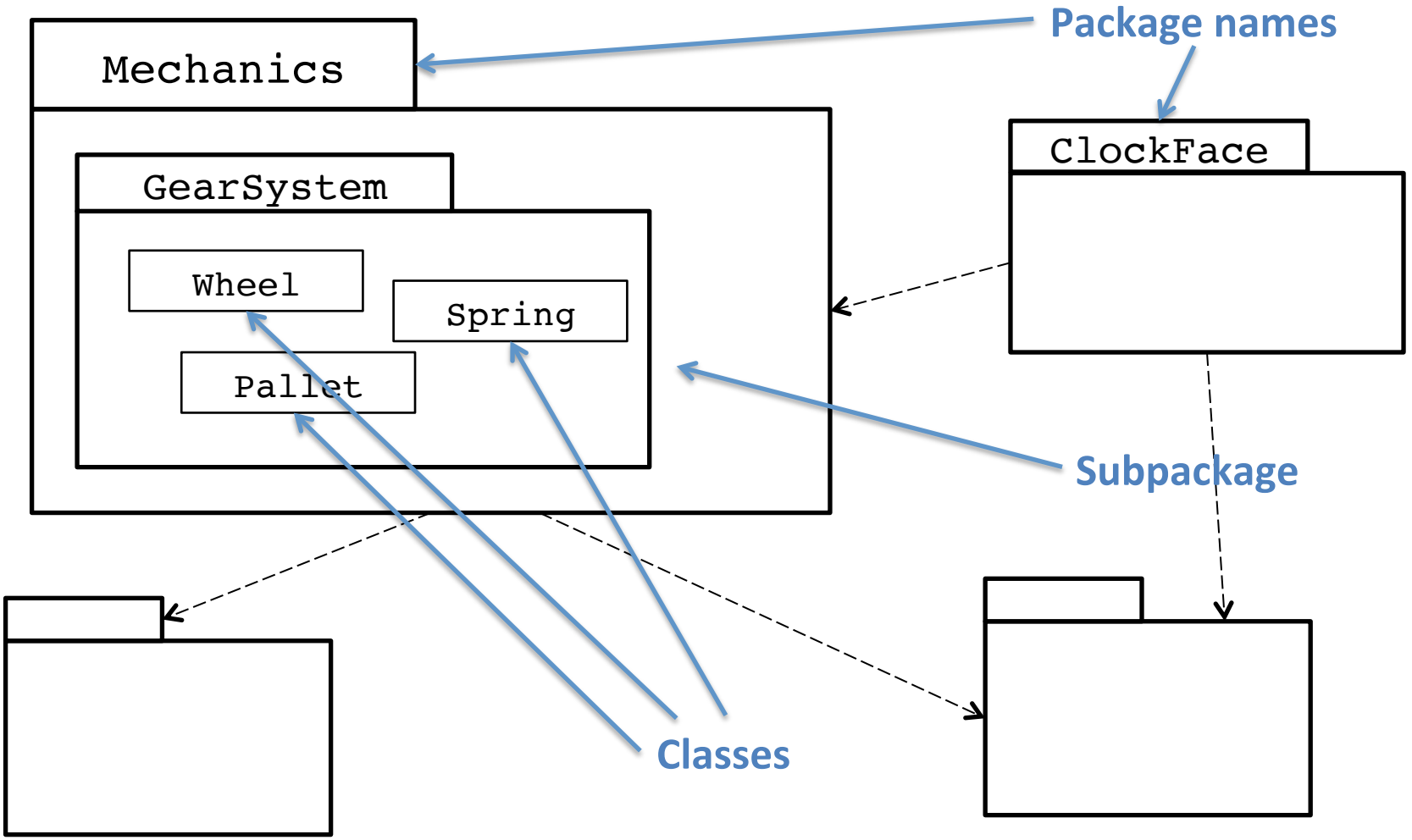


Associations Summary

- **is-a**
  - Class A is a class B
- **belongs-to**
  - Class A belongs to class B
- **part-of**
  - Class A is part of class B
- **uses**
  - Class A uses class B



Package Diagram



Package Diagram (2)

- Package diagram is a good way to present **subsystem decomposition**
- It groups the related classes and subpackages together
- Package dependencies can also be specified
- Remember, we always want **high cohesion / low coupling**

Coupling and Coherence of Subsystems

Good Design

- Goal: Reduce system complexity while allowing change
- **Coherence** measures dependency among classes
  - **High coherence:** The classes in the subsystem perform similar tasks and are related to each other via associations
  - **Low coherence:** Lots of miscellaneous and auxiliary classes, (almost) no associations
- **Coupling** measures dependency among subsystems
  - **High coupling:** Changes to one subsystem will have high impact on the other subsystem
  - **Low coupling:** A change in one subsystem does not affect any other subsystem

How to achieve high Coherence

- **High coherence** can be achieved if most of the interaction is within subsystems, rather than across subsystem boundaries
- Questions to ask:
  - Does one subsystem always call another one for a specific service?
    - Yes: Consider moving them together into the same subystem.
  - Which of the subsystems call each other for services?
    - Can this be avoided by restructuring the subsystems or changing the subsystem interface?
  - Can the subsystems even be hierarchically ordered (in layers)?

How to achieve Low Coupling

- **Low coupling** can be achieved if a calling class does not need to know anything about the internals of the called class (**Principle of information hiding**)
- Questions to ask:
  - Does the calling class really have to know any attributes of classes in the called class?
  - Is it possible that the calling class calls only operations of the called classes?

CRC Cards

- **CRC Cards**: Classes, Responsibilities, Collaboration Cards
  - It is a tool to design classes and collaboration of classes
  - Used in **brainstorming session**. It is not intended to be used alone.
  - It allows the designer to think out of the system
    - Not how the system works, but how the actual objects should behave
    - Let the designers focus on the properties and operations of the classes without getting into details
- When you design your system and do not know where to start, CRC might help.

CRC Card Components

- A CRC card :
  - Is a small paper index card (around 3”x5”)
  - represents one class
  - It is intended to be small so that you will focus only on important information
- It is composed of
  - **Class Name**: The name of the class
  - **Responsibilities**: Every thing that this class knows or does
  - **Collaborators**: Other classes that must work with this class in order to achieve a task (i.e., a use case)

Class Name	
Responsibilities	Collaborators

CRC Card Example

- A CRC card represents the class of a customer of an online store

Customer	
Update name Update address Request purchase history Process sale Make payment	Sale Payment

Class Responsibilities

- Responsibilities of a class: What does the class **knows** or **does**
  - Attributes of the class
  - Operations of the class
- A class can only change the values of what it knows, it cannot change values of what it does not know
  - It has need to tell other classes to do
- Example: A customer knows his name. He can also make payments (Or can he?)

Classes Collaboration

- One class cannot do or know everything.
  - Super-large class that does too many things is called a **Blob** and you should avoid.
- A class must use or interact with other classes to achieve a task
  - In this context, we call is **collaboration**
- Classes collaboration:
  - Request for information
  - Request to do something
- Example: The customer may not know the payment ID or payment process
  - In this case **Customer** and **Payment** are collaborating with each other to achieve **make payment** task

Creating CRC Cards

- **Find user story**: CRC process is to design a single user story or use case. So, pick one story before you start.
- **Find classes**: Study your requirement documentation and come up with classes
- **Find responsibilities**: Determine what the class does and the information that the class should maintain
  - Sometimes the class must do things for other classes too
  - Different designers may play the role of each class, acting out the use case by verbally sending messages to each other demonstrating responsibilities
- **Define collaborators**: For each responsibility, determine whether the class can do it alone, otherwise which classes need has to help

Creating CRC Cards (2)

- **Move the cards around**:
  - The cards should be placed on the table (or a board) in a meaningful manner
  - The cards that are collaborating with each other should be placed together
  - The system will be easier to understand when related cards are placed together
  - The cards maybe moved anywhere anytime as the team discuss the roles and collaborations of the classes

CRC Model

- At the end, you will have a model of the collaboration

Sales Handler	
Handle new sale	Customer Sale SaleItem

Customer	
Update name Update address Request purchase history Process sale Make payment	SaleTransaction

Sale	
Update information Request shipping Update status Cancel sale Add items to sale Take payment	SaleItem Transaction

SaleItem	
Update information Cancel item Request back order	PromoOffering Product InventoryItem

SaleTransaction	
Process payment	Customer sale

PromoOffering	
Provide price	

ProductItem	
Provide description	

InventoryItem	
Provide quantity Update quantity Order new supply	

Case Study

- Let us try to define a system