MC322 - Object Oriented Programming
Lesson 7.1
Mastering Inheritance and Composition

Prof. Marcos M. Raimundo
Instituto de Computação - UNICAMP

Inheritance and Composition in Object-Oriented Design

- Inheritance and composition play major roles in the design of object-oriented (OO).
- Many difficult and interesting design decisions revolve around choosing between inheritance and composition.
- Both inheritance and composition are mechanisms for reuse.
  - Inheritance involves inheriting attributes and behaviors from other classes.
  - It establishes a true parent/child relationship where the child (or subclass) inherits directly from the parent (or superclass).
- Composition involves building objects by using other objects.
- This chapter will explore the differences between inheritance and composition, considering appropriate times to use one or the other.

Reusing Objects

Composition in Object-Oriented Design

- Composition involves using other classes to build more complex classes—a sort of assembly.
- There is no parent/child relationship in this case.
- Basically, complex objects are composed of other objects.
- Composition represents a has-a relationship.
- For example, a car has an engine.
  - Both the engine and the car are separate, potentially standalone objects.
  - However, the car is a complex object that contains (has an) engine object.
- A child object might itself be composed of other objects.
  - For example, the engine might include cylinders.
  - In this case, an engine has a cylinder, actually several.

Evolution of Inheritance in Object-Oriented Design

- When OO technologies first entered the mainstream, inheritance was highly favored.
  - Designing a class once and then inheriting functionality from it was considered the foremost advantage.
  - Reuse was the primary goal, and inheritance was seen as the ultimate expression of reuse.
- However, over time, the enthusiasm for inheritance has diminished.
  - Some discussions even question the use of inheritance itself.
  - In their book ”Java Design,” Peter Coad and Mark Mayfield dedicate a chapter titled ”Design with Composition Rather Than Inheritance.”
  - Early object-based platforms often lacked true inheritance support.
  - For example, early versions of Visual Basic .NET relied on interface inheritance rather than strict inheritance capabilities.
- The debate between inheritance and composition has led to a more balanced perspective.
  - Discussions have progressed towards a middle ground.
  - While there are passionate arguments on both sides, a more sensible understanding of technology utilization has emerged.

Inheritance vs. Composition: Valid Design Techniques

- Later in this chapter, we will explore why some advocate for avoiding inheritance and favoring composition as the primary design method.
  - The argument is complex and subtle.
- Both inheritance and composition are valid class design techniques.
  - They each have their proper place in the OO developer's toolkit.
- Misuse and overuse of inheritance are often due to a lack of understanding of its true nature, rather than a fundamental flaw in its design strategy.
- Inheritance and composition are both important techniques in building OO systems.
  - Designers and developers should understand the strengths and weaknesses of both and use each appropriately.

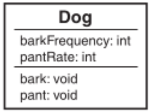
Inheritance

Understanding Inheritance in Object-Oriented Design

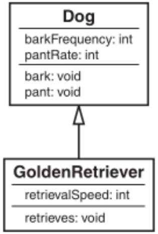
- **Definition and Concept:** Inheritance allows child classes to derive attributes and behaviors from a parent class. It forms the backbone of reusability and hierarchy in object-oriented design.
- **Is-a Relationship:** One of the fundamental rules of object-oriented design is that public inheritance should represent an "is-a" relationship. This relationship suggests that if Class B is a Class A, then it is suitable for inheritance.
- **Practical Example:** Consider the Dog class from Chapter 1:
  - **Behaviors:** A dog barks and pants—distinct behaviors that differentiate it from other mammals like cats.
  - **Attributes:** Attributes of the Dog class might include things like breed, age, or coat color (these specifics could be illustrated in Figure 7.1).
- **Implementation:** This setup suggests that Dog could inherit from a general Mammal class, leveraging shared attributes (like heart rate or body temperature) while specifying behaviors unique to dogs.

Inheritance in Practice: Extending the Dog Class

- **Building on Existing Classes:** Rather than starting from scratch, the ‘GoldenRetriever‘ class is derived from the ‘Dog‘ class. This is a practical application of the ”is-a” relationship— a Golden Retriever is a type of dog.
- **Inheritance Benefits:** By inheriting from ‘Dog‘, the ‘GoldenRetriever‘ class automatically gains all behaviors (like barking and panting) and attributes of its parent class.
- **Efficiency in Development:** This approach minimizes redundancy:
  - **Code Reuse:** Reduces the amount of code written, as common behaviors are coded once in the parent class.
  - **Reduced Testing and Maintenance:** Since the inherited methods are already tested in the ‘Dog‘ class, they require less testing in ‘GoldenRetriever‘, although retesting in the new context is advisable to ensure they work as intended.



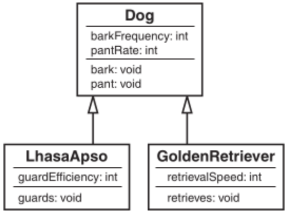
A class diagram for the Dog class.



The GoldenRetriever class inherits from the Dog class.

Expanding the Dog Class Hierarchy

- **Specialized Inheritance:** Continuing with the inheritance structure, we introduce the ‘LhasaApso‘ subclass. This breed is known for its role as a guard dog, utilizing its acute senses rather than aggression to alert owners by barking.
- **Contextual Differences:** Unlike the GoldenRetriever, which is bred for retrieving, the LhasaApso has a distinct role that might influence how inherited methods, such as barking, function under different conditions.



A class diagram for the Dog class.

Testing New Code

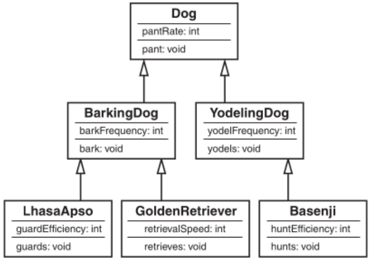
While inherited methods like bark and pant from the Dog class are already tested, introducing them into a new class like LhasaApso presents a new context. Each breed may exhibit unique traits affecting these behaviors, necessitating retesting to ensure reliability across different scenarios. Testing is crucial, even for inherited code, to verify its functionality in each specific subclass context.

Advantages and Pitfalls of Inheritance

- **Code Centralization:** Inheritance allows methods like `bark()` and `pant()` to be defined in a single place in the superclass (Dog). Changes made in these methods automatically propagate to all subclasses (e.g., LhasaApso and GoldenRetriever), simplifying maintenance and updates.
- **Inheritance Pitfalls:** While the inheritance model simplifies modifications and maintenance, it assumes that all subclasses will universally share or need behaviors defined in the superclass.
  - **Problematic Assumptions:** Not all dogs might exhibit the behaviors defined in the Dog class, such as barking or panting in specific ways that suit all dog types.
- **The Bird Example:** Drawing from Scott Meyers' example in *Effective C++*:
  - A class Bird includes a `fly()` method, which does not apply to non-flying birds like penguins or ostriches. Overriding `fly()` in a Penguin class to perform a waddle action, while semantically incorrect, highlights the limitation of using a broad inheritance for diverse behaviors.
- **Design Consideration:** This raises questions about the appropriate use of inheritance versus composition and whether methods like `fly()` should be universally inherited or selectively implemented.

Reevaluating Inheritance in the Dog Class Hierarchy

- **Inheritance Limitations:** Initially, our Dog class was designed with the assumption that all dogs can bark.
- **Case of Basenji:** Basenjis are known as "barkless" dogs. Instead of barking, they produce a unique sound known as a yodel.
- **Proposed Design Changes:**
  - Introduce a more flexible class structure where the `bark()` method can be overridden or replaced by other sound-making behaviors like `yodel()` for specific breeds.
  - Consider using interfaces or abstract classes to define behaviors that can vary significantly among subclasses, such as vocalizing.
- **Modeling the Correct Hierarchy:** Figure would ideally illustrate a revised hierarchy where the Dog class does not strictly enforce barking as a universal behavior. Instead, depending on the breed, it might show an interface for vocalization, with different implementations for barking, yodeling, etc.



The Dog class hierarchy.

Inheritance: Generalization and Specialization

Generalization-Specialization in Dog Class Hierarchy

- **Inheritance Concept:** The object-oriented design principle of generalization-specialization (or the "is-a" relationship) is foundational in structuring classes using inheritance. This approach organizes classes from the most general to the most specific.
- **Hierarchy Structure:**
  - **Top of the Tree:** The Dog class sits at the top of the hierarchy as the most general class. It encapsulates behaviors and attributes common to all dogs, such as eating, sleeping, or basic movements.
  - **Specific Breeds:** Subclasses like GoldenRetriever, LhasaApso, and Basenji represent more specialized instances. Each inherits common traits from Dog but also introduces unique behaviors or attributes specific to their breed, like the Basenji's yodel instead of a bark.
- **Design Benefit:** This design allows for efficient code reuse and maintenance. Common methods are written once in the Dog class and inherited by all breeds, reducing redundancy and simplifying updates. Specialized behaviors are then added to subclasses where necessary.
- **Modeling Strategy:** Effective use of this principle requires careful consideration to ensure that the general class does not enforce behaviors that are not universal (e.g., barking). This might involve abstract classes or interfaces for highly variable behaviors.

Refining the Dog Class Hierarchy for Diverse Behaviors

- **Evolving the Hierarchy:** All dog behaviors were modeled in a single Dog class. With the realization that not all dogs bark (e.g., Basenjis yodel), we adapted the inheritance structure:
  - **BarkingDog and YodelingDog Classes:** These subclasses were created to represent dogs that bark and those that yodel. Both inherit from the general Dog class, retaining panting.
- **Hierarchy Distribution:**
  - Dogs that bark (e.g., LhasaApso and GoldenRetriever) inherit from BarkingDog.
  - Dogs that yodel (e.g., Basenji) inherit from YodelingDog.
- **Alternative Approaches:**
  - **Single Class Approach:** One could choose not to differentiate behaviors at the class level and instead implement barking and yodeling directly within each breed's class. This might allow for more breed-specific behavior customization.
  - **Interface Implementation:** Implementing barking and yodeling as interfaces might offer a flexible and reusable approach, allowing the definition of specific behaviors without hierarchy constraints.
- **Design Considerations:** These decisions underscore the importance of careful design in object-oriented programming, balancing between inheritance for common behavior and flexibility for unique traits.

Inheritance: Design Decisions

Balancing Complexity and Accuracy in System Design

What Computers Are Not Good At

Computers excel at processing large quantities of data but struggle with abstract concepts. They can model real-world scenarios only to an extent. A computer model, no matter how detailed, is an approximation and might not fully capture the nuances of real-world behaviors.

Model Complexity

While the addition of two classes (BarkingDog and YodelingDog) might seem manageable, in larger systems, similar decisions can significantly increase complexity. Over time, this can make the system less manageable. It is often best to keep the system as simple as possible, especially in large-scale applications.

- **Design Conundrum:** While theoretically desirable, factoring out as much commonality as possible in a design can lead to increased complexity. This raises a key question: Should the design aim for higher realism at the cost of increased system complexity?
- **Practical Example:** Splitting the Dog class into BarkingDog and YodelingDog more accurately reflects real-world diversity among dogs, but introduces additional complexity to the model.
- **Decision Criteria:** Deciding between a more accurate or a simpler model depends on specific project requirements, such as the system's purpose, expected lifetime, maintenance resources, and how critical

Balancing Complexity and Functionality in Design

- **Practical Scenario:** Consider a dog breeder who needs a system to track all dogs. If the breeder does not deal with yodeling dogs, incorporating both BarkingDog and YodelingDog classes may unnecessarily complicate the system.
- **Simplifying the Model:** For the breeder, eliminating the YodelingDog class simplifies the system without sacrificing necessary functionality, perfectly suiting their specific needs.
- **Balancing Act:** Design decisions often involve weighing less complexity against the need for functionality. The aim is to create a system that remains manageable and cost-effective while fulfilling its intended purpose.
- **Consideration of Costs:** Both current and future costs impact these decisions. Adding features that do not significantly enhance the system's value can be a poor investment. For instance:
  - **Extended Functionality:** Expanding the Dog system to track other canines like hyenas and foxes might offer more completeness, but if such features are unlikely to be used, the extra complexity might not justify the costs.
- **Strategic Design:** Ultimately, each feature or class added should provide a clear benefit. Over-designing can lead to inflated costs and a cumbersome system that is difficult to maintain and expand.

An expanded canine mode

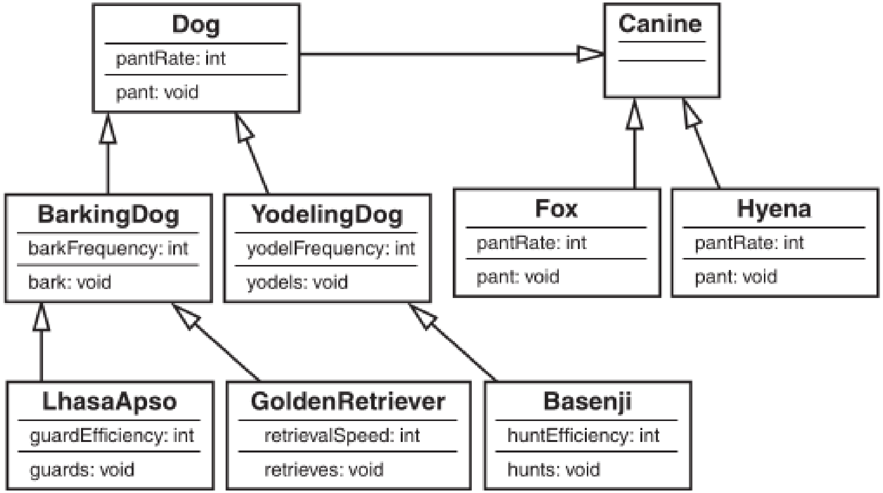


Figure 1: An expanded canine mode

Trade-offs in Design: Current Needs vs. Future Possibilities

- **Design Trade-offs:** When designing a system, especially one involving class hierarchies like Dog or Canine, it's essential to consider the specific context and needs. For instance, a zookeeper may require a more expansive Canine class than someone breeding only domesticated dogs.
- **Design for the Future:** While it might seem excessive to include capabilities like yodeling for a breeder who doesn't need them, future changes in business or scope could make such features relevant. Not designing for such possibilities now could lead to higher costs and complexities later.
- **Anticipating Change:** This "Never say never" approach encourages designing systems that are somewhat flexible and scalable without overly complicating the current application.

Making Design Decisions with the Future in Mind

Careful consideration is needed when deciding how much to invest in future-proofing a system. For example, incorporating the ability for a dog to yodel (by overriding the `bark()` method) may not be intuitive and could confuse users expecting typical behavior from a method named `bark()`. This emphasizes the need for clear, intuitive designs that can adapt over time without misleading users or

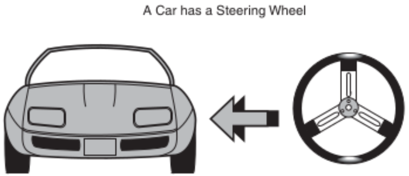
Composition

Object Composition in Design

- **Understanding Object Composition:** Object composition involves constructing complex objects by combining simpler, self-contained objects. This principle is a cornerstone of modular, maintainable design in both physical and software systems.
- **Real-World Examples:**
  - **Electronic Devices:** A television set comprises components like a tuner and video display. Similarly, a computer comprises video cards, keyboards, and drives.
  - **Computer Components:** The interchangeability of parts like a hard drive, which can be moved between computers, underscores the standalone nature and modularity of these components.
- **Automobiles as a Classic Example:**
  - The automobile is often cited as a classic example of object composition in educational resources. The concept of interchangeable parts, popularized by Henry Ford's assembly line, illustrates how individual components can be assembled to form a complete vehicle.
- **Application in Software Design:**
  - In object-oriented software systems, designing with object composition allows for parts of a system to be easily replaced or upgraded without affecting the integrity of the whole. This approach enhances flexibility, scalability, and ease of maintenance.

Understanding Composite Objects and Class Reuse

- **Composite Objects:** A composite object is formed when an object contains other objects as part of its structure, often represented as object fields within a class. This is common in everyday objects, such as cars, which contain engines, wheels, steering wheels, and stereos.
- **Definition and Examples:**
  - These objects, like a car, which consists of multiple smaller objects (engine, wheels, etc.), are known as compound, aggregate, or composite objects.
  - This structure allows for modular design where components can be easily maintained or replaced independently.



Aggregation, Association, and Composition

- **Class Reuse:** Class reuse in object-oriented programming is primarily achieved through two methods: inheritance and composition.
- **Composition Details:** Composition, including aggregation and association, is discussed further in Chapter 9. Although opinions vary, in this context, both aggregation and association are considered forms of composition:
  - **Aggregation** implies a relationship where the child can exist independently of the parent.
  - **Association** refers to a relationship where two objects are related but without aggregation's strong life cycle dependency.
- These relationships allow objects to be used and reused in different contexts, enhancing the flexibility and scalability of software designs.

Representing Composition with UML

- **UML for Composition:** A car containing a steering wheel is represented in UML by connecting the car and steering wheel with a line that typically ends with a filled diamond at the car end, indicating ownership and lifecycle dependency.
- **UML Notation:** The notation emphasizes that the steering wheel is a part of the car and cannot exist independently (in the context of the model).

Aggregation, Association, and UML

- **Aggregation in UML:** Represented by a line with an unfilled diamond at one end, showing a "part-of" relationship where components can exist independently of the whole. For example, an engine in a car is necessary but can exist outside of it.
- **Association in UML:** Denoted by just a line, this indicates a relationship between two connected entities that do not depend on each other's lifecycle. A keyboard connected to a computer does not affect the existence of the computer or vice versa.

Note that the line connecting the Car class to the SteeringWheel class has a diamond shape on the Car side of the line.This signifies that a Car contains (has-a) SteeringWheel.

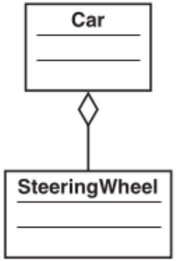


Figure 2: Representing composition in UML

Exploring Composition in Object Design

- **Composition Example:** Consider a car as an object composed entirely through composition, not inheritance. This means that each component of the car, such as the engine, stereo system, and door, is an object in its own right, linked solely by composition.
- **Levels of Composition:** In this example, a car integrates multiple objects:
  - **Engine:** Powers the car and is essential for its operation.
  - **Stereo System:** Provides entertainment, which is not essential for the car's operation.
  - **Door:** Provides access to the car's interior.
- **Simplifying for Illustration:** While a real car would typically have multiple doors and at least one stereo system, this example simplifies the scenario to a single door and a single stereo system to focus on illustrating basic composition principles.

How Many Doors and Stereos?

- **Doors:** Cars usually have two or four doors, hatchbacks might be considered to have a fifth door.
- **Stereos:** While most cars have one stereo system, some may have none or even multiple systems for enhanced audio experiences.

For our example, imagine a unique scenario, perhaps a special racing car, with only one door and one

Nested Composition in Car Design

- **Deep Composition:** In our car example, not only does the car consist of an engine, a stereo system, and a door, but each of these components is itself composed of smaller parts:
  - **Engine:** Contains pistons and spark plugs, each essential for its operation.
  - **Stereo System:** Includes a radio and a CD player, with each part contributing to the functionality of the stereo.
  - **Door:** Equipped with a handle, and potentially a lock within that handle, adding a level of detail to its composition.
- **Additional Levels:**
  - The radio itself contains a tuner, and further, a tuner could contain a dial.
  - A CD player might include buttons like fast forward, each representing another layer of composition.
- **Design Choices:** The depth of composition chosen by the designer depends on the requirements and the intended use of the model. More detailed compositions allow for greater specificity but can also add complexity.

Nested Composition in Car Design

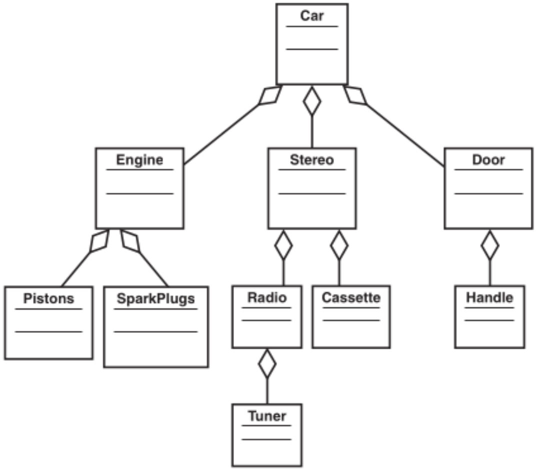


Figure 3: The Car class hierarchy

Model Complexity

While detailed composition provides a clear and specific representation of each component's functionality, it also increases the complexity of the overall model. Designers must find a balance where the model is detailed enough to be useful and expressive but not so complex that it becomes cumbersome to understand and maintain. This balance is crucial to effective software design and can vary significantly from one project to another.

Why Encapsulation Is Fundamental to OO

Encapsulation and Its Interplay with Inheritance

- **Defining Encapsulation:** Encapsulation is a core principle of OO design, focusing on dividing a class into a public interface and a private implementation. This division ensures that the internal workings of a class are hidden from the outside, exposing only what is necessary through a well-defined interface.
- **Encapsulation in Practice:**
  - It involves careful consideration of what aspects of a class should be accessible to the outside world and which should remain hidden, applying to both data (attributes) and behavior (methods).
  - Stephen Gilbert and Bill McCarty describe encapsulation as the process of packaging a program by separating each class into interface and implementation parts, a theme consistently emphasized throughout OO discussions.

Encapsulation and Its Interplay with Inheritance

- **Encapsulation and Inheritance:**
  - While encapsulation seeks to hide details and expose only necessary parts, inheritance allows a subclass to inherit the properties and methods of a parent class.
  - This relationship can potentially break encapsulation as it exposes the internal structure of a class to its subclasses, which may lead to dependencies that are hard to manage as systems evolve.
- **The OO Paradox:**
  - There seems to be an inherent paradox where inheritance, a pillar of OO, appears to contradict the principle of encapsulation by potentially exposing protected and internal class details beyond the intended scope.
  - Exploring whether these core concepts are indeed incompatible, or if they can be reconciled through careful design, is a crucial aspect of advanced OO programming.

Encapsulation vs. Inheritance: Balancing Class Visibility and Extensibility

- **Encapsulation Overview:** Encapsulation involves separating a class into a public interface and a private implementation, hiding internal details from other classes to ensure modularity and ease of maintenance.
- **Inheritance and Encapsulation:**
  - Peter Coad and Mark Mayfield highlight that while inheritance strengthens encapsulation from external classes, it weakens it within a class hierarchy. This is because subclasses have access to their superclass's protected members, potentially exposing more internal details than intended.
  - Inheritance allows subclasses to inherit and potentially alter implementations, which can lead to significant "rippling effects" where changes in the superclass affect all subclasses.

Encapsulation vs. Inheritance: Balancing Class Visibility and Extensibility

- **Implications for Design and Testing:**
  - This inherent conflict can complicate maintenance and testing. While encapsulation generally aids in isolating classes for testing, inheritance can complicate this by intertwining the functionalities of subclasses with their superclass.
  - For instance, modifying a superclass can unexpectedly impact subclasses, making testing challenging as changes are not confined to the modified class but also affect its descendants.
- **Design Conundrum:** The challenge lies in designing a system where the benefits of inheritance (reusability and extensibility) are balanced with the need for strong encapsulation to maintain clear and manageable class boundaries.

Impact of Superclass Changes on Subclasses

- **Inheritance in Practice:** When ‘PartTimeCabbie‘ inherits from ‘Cabbie‘, it adopts all public implementations from ‘Cabbie‘, including methods like ‘giveDirections()‘.
- **Changing Superclass Behavior:**
  - Changes to the ‘giveDirections()‘ method in ‘Cabbie‘ will automatically affect ‘PartTimeCabbie‘, since ‘PartTimeCabbie‘ inherits this method.
  - This scenario exemplifies how encapsulation can be compromised in an inheritance hierarchy. Although ‘Cabbie‘’s methods are intended to be reused by subclasses, any modification can inadvertently impact all subclasses that rely on these methods.
- **Design Considerations:**
  - This inheritance model necessitates careful consideration when modifying methods in a superclass, as these changes are propagated to all inheriting classes, potentially leading to unintended behaviors in subclasses.
  - It underscores the importance of robust design and testing strategies to ensure that changes in superclass methods do not adversely affect subclass functionalities.

Impact of Superclass Changes on Subclasses

- **UML Diagram Context:** The UML diagram in Figure illustrates the relationship between ‘Cabbie‘ and ‘PartTimeCabbie‘, visually representing the inheritance link and method sharing. This helps in understanding the structural dependencies within the class hierarchy.

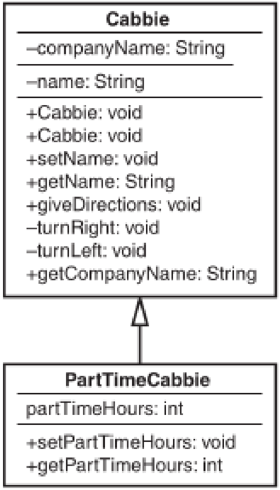


Figure 4: A UML diagram of the Cabbie-PartTimeCabbie classes

Proper Use of Inheritance and the ”Is-A” Relationship

- **Example of Correct Inheritance:**
  - A ‘Circle‘ class inheriting from a ‘Shape‘ class is appropriate because every circle is a shape, and any modifications to the ‘Shape‘ class that would break the ‘Circle‘ class suggest a flaw in the class design or the relationship.
- **Misuse of Inheritance:**
  - Creating a ‘Window‘ class as a subclass of a ‘Rectangle‘ class in a GUI system is a misuse of inheritance because a window is not strictly a type of rectangle but contains rectangles (e.g., menu bars, status bars, main view areas).
  - Correct Approach:

```java
public class Window {
    Rectangle menubar;
    Rectangle statusbar;
    Rectangle mainview;
}
```

This composition approach correctly models the relationship, as a window comprises multiple rectangular components rather than being a rectangle itself.

Understanding Polymorphism in OO Design

- **Concept of Polymorphism:** Polymorphism, one of the core principles of OO design, refers to the ability of different objects to respond to the same message (or method call) in different ways. It allows objects to be treated as instances of their parent class, but to behave differently according to their specific subclass implementation.
- **Polymorphism in Practice:**
    - Consider a class hierarchy where ‘Shape‘ is the superclass with a method called ‘Draw‘. This method, while defined in ‘Shape‘, is abstract—meaning it has no implementation.
    - Subclasses of ‘Shape‘, such as ‘Circle‘, ‘Square‘, and ‘Triangle‘, each provide their own specific implementation of the ‘Draw‘ method. This is a demonstration of polymorphism where the same method name behaves differently depending on the object’s class.
- **Method Overriding:**
    - Overriding is a key aspect of polymorphism. It allows a subclass to provide a specific implementation of a method that is already defined in its superclass.
    - Example: In the ‘Circle‘ subclass, the ‘Draw‘ method is overridden to implement drawing logic specific to circles, distinct from, say, squares or triangles.

Polymorphism and Object Responsibility in OO Design

- **Understanding Polymorphism:** Polymorphism allows objects to interact through a common interface with behavior that varies depending on the object's class. This principle is a cornerstone of effective inheritance and interface design in OO.
- **Shape Example:**
  - The 'Shape' class is an abstract class with an abstract method 'getArea()'. This method is designed without implementation because the calculation of area differs based on the shape type.
  - Concrete subclasses like 'Rectangle' and 'Circle' implement the 'getArea()' method to calculate their specific area based on their dimensions.
- **Polymorphism in Action:**
  - When the 'getArea' message is sent to any shape object, the response is determined by the object's type. For example, a 'Circle' will respond by calculating its area using πr^2, whereas a 'Rectangle' might use length $\times$ width.
  - This design ensures that each class is responsible for its own unique implementation of common methods, adhering to the principle that "an object should be responsible for itself."

Polymorphism in Java: Shape Example

- **Abstract and Concrete Classes:** An abstract class ‘Shape‘ defines the method ‘draw()‘ without implementation, leaving it to its subclasses to provide specific behaviors.

```
public abstract class Shape {
    public abstract void draw();
}
```

  Concrete classes ‘Circle‘, ‘Rectangle‘, and ‘Star‘ each implement the ‘draw()‘ method to output a message specific to their shape.

```
public class Circle extends Shape {
    public void draw() {
        System.out.println("I am drawing a Circle");
    }
}
public class Rectangle extends Shape {
    public void draw() {
        System.out.println("I am drawing a Rectangle");
    }
}
public class Star extends Shape {
    public void draw() {
        System.out.println("I am drawing a Star");
    }
}
```

Polymorphism in Java: Shape Example

- **Demonstration of Polymorphism:** The ‘TestShape‘ class demonstrates polymorphism by creating instances of ‘Circle‘, ‘Rectangle‘, and ‘Star‘, and invoking the ‘draw()‘ method on each.

```
public class TestShape {
    public static void main(String args[]) {
        Circle circle = new Circle();
        Rectangle rectangle = new Rectangle();
        Star star = new Star();
        circle.draw();
        rectangle.draw();
        star.draw();
    }
}
```

  Each object calls its own version of ‘draw()‘, showcasing how polymorphism allows objects to behave in ways particular to their specific class, despite all being instances of ‘Shape‘.

- **Key Takeaway:** This example illustrates the core principle of OO design where objects are responsible for themselves. Each class manages its own behavior independently, adhering to the defined contract by the superclass.

Extending Polymorphism with a New Shape

- **Adding a New Class:** To introduce a new shape, into our existing class hierarchy, simply define the new class and implement the required method from the 'Shape' abstract class.

  This addition demonstrates the power of polymorphism and abstraction in object-oriented design. The 'Shape' class remains unchanged, as do the other shapes.

- **Testing:** With the new 'Triangle' class, testing involves simply creating an instance and calling its 'draw()' method, along with the methods of other shape instances.

  This test setup confirms that the new 'Triangle' class integrates seamlessly with the existing system, adhering to the established polymorphic behavior.

- **Polymorphism at Work:** The 'Shape' class's design allows any new shapes to be added without modifying existing code. Each shape independently handles its drawing behavior, fulfilling the polymorphic principle that objects should act independently based on the same interface.

```
public class Triangle extends
     Shape {
    public void draw() {
        System.out.println("I
             am drawing a
             Triangle");
    }
}
```

```
public class TestShape {
    public static void
         main(String args[]) {
        Circle circle = new
             Circle();
        Rectangle rectangle =
             new Rectangle();
        Star star = new
             Star();
        Triangle triangle =
             new Triangle();
        circle.draw();
        rectangle.draw();
        star.draw();
        triangle.draw();
    }
}
```

Demonstrating Advanced Polymorphism in Java

- **Polymorphism Through Generalization:** The ‘drawMe‘ method illustrates the power of polymorphism by accepting any object that extends ‘Shape‘. This method can invoke the ‘draw()‘ method on any shape instance, regardless of its specific class.

```java
public static void drawMe(Shape s) {
    s.draw();
}
```

- **Flexible Method Invocation:** The flexibility of polymorphism is showcased by the ability to pass different shapes to ‘drawMe‘ without modifying its implementation:

```java
public class TestShape {
    public static void main(String args[]) {
        Circle circle = new Circle();
        Rectangle rectangle = new Rectangle();
        Star star = new Star();
        drawMe(circle);
        drawMe(rectangle);
        drawMe(star);
    }
}
```

Each call to ‘drawMe‘ results in the correct ‘draw‘ method being invoked for the shape passed, demonstrating the method’s ability to handle any kind of ‘Shape‘ object.

Demonstrating Advanced Polymorphism in Java

- **Adding New Shapes:**
  - If a new shape, like ‘Triangle‘, is added to the system, it can be immediately used with the ‘drawMe‘ method without any changes to existing code. This ability to seamlessly integrate new classes without altering existing functionality is a hallmark of well-designed polymorphic systems.
- **Implications for Software Design:**
  - This approach minimizes code dependencies and enhances modularity, making the system easier to expand, maintain, and debug.
  - It exemplifies how designing with polymorphism leads to more reusable and adaptable code.

Conclusion

Conclusion: Inheritance vs. Composition

- **Overview of Inheritance and Composition:**
  - This chapter outlined the fundamental concepts of inheritance and composition, highlighting their differences and applications within object-oriented (OO) design.
- **Common Guidance and Realities:**
  - While many experienced OO designers advocate for favoring composition over inheritance, it is essential to understand that this advice is not absolute. Composition is often more flexible and less prone to creating tightly coupled code, which is why it might be more suitable in many scenarios.
  - However, inheritance is not inherently problematic; it is a powerful tool for creating hierarchical class structures that should be used judiciously and in the right contexts.
- **Balanced Approach:**
  - The key is to use both composition and inheritance effectively, applying each where it fits naturally within the design. Understanding when to use one over the other comes from both experience and a clear grasp of the specific problems being solved.

MC322 - Object Oriented Programming
Lesson 7.1
Mastering Inheritance and Composition

Prof. Marcos M. Raimundo
Instituto de Computação - UNICAMP