MC322 - Object Oriented Programming
Lesson 8.1
Frameworks and Reuse:
Designing with Interfaces and Abstract Classes

Prof. Marcos M. Raimundo
Instituto de Computação - UNICAMP

Introduction to Java Interfaces and Abstract Classes

- **Expansion on OO Design Principles:** Building on the foundational principles of inheritance and composition, this class introduces Java interfaces and abstract classes as crucial tools for enhancing code reuse and structuring robust OO systems.
- **Role of Interfaces and Abstract Classes:** Java interfaces and abstract classes serve as mechanisms to define strict behavioral contracts within software designs. They enable developers to establish clear, enforceable guidelines for how system components should interact and function.
- **Concept of Contracts:** A contract in software design refers to a set of defined interfaces or abstract methods that a class agrees to implement. This ensures that all derived classes adhere to a predetermined behavior framework, promoting consistency and reliability in code.
- **Topics Covered:** This lesson discusses code reuse strategies, the creation and use of frameworks, and the implementation of contracts through Java interfaces and abstract classes.

Code: To Reuse or Not to Reuse?

Code Reuse Across Programming Paradigms

- **Historical Perspective on Code Reuse:** Code reuse is a foundational concept in software development, crucial for enhancing efficiency and reducing duplication.
- **Role of Object-Oriented Design:**
  - OO design is renowned for its support for code reuse, primarily through mechanisms like inheritance, polymorphism, and encapsulation.
  - However, the effectiveness of OO design in promoting code reuse significantly depends on the quality of the initial design and implementation.
- **Comparative Analysis with Non-OO Languages:** While OO languages provide structured approaches to code reuse, non-OO languages such as COBOL, C, and traditional VB also support the development of reusable code. This can be achieved through well-designed functions and modular programming techniques.
- **Conclusion:** A well-thought-out design, clear documentation, and a commitment to maintainable code standards are essential to creating reusable code in any programming environment. The key to effective reuse lies in how code is structured and implemented, regardless of the language or paradigm employed.

What Is a Framework?

Framework Standardization and Code Reuse in Software Design

- **Concept of Standardization and Reuse:** Standardization, or the "plug-and-play" principle, is integral to creating flexible and reusable software frameworks. It ensures that components can be easily integrated and reused across different parts of an application or different applications.
- **Example of an Office Suite:**
  - Office applications like Microsoft Word, PowerPoint, and Excel utilize a common framework that standardizes elements such as the menu bar and toolbar.
  - The uniformity across these applications in menu items (File, Edit, View, Insert, Format, Tools) and toolbar icons (New, Open, Save) exemplifies how standardization supports a consistent user experience and eases the learning curve for new software within the suite.
- **Benefits of a Common Framework:**
  - **Ease of Use:** Users benefit from a consistent interface and operation style across multiple applications, making transitioning between tools within the office suite easier.
  - **Development Efficiency:** Developers benefit from the ability to reuse code and design elements, reduce development time and costs, enhance maintainability, and ensure product consistency.

Standardization and Frameworks in Software Development

- **Standard Framework Components:**
  - In environments like Microsoft Windows, standard components such as title bars, close buttons, and menu options are pre-defined within the development framework.
  - These elements provide a consistent look and feel across applications, ensuring that when a window is created, it inherits behaviors like minimizing/maximizing on double-clicking the title bar or closing when the close button is clicked.
- **Practical Implementation:**
  - A word processing software framework, for example, might include standard operations for document management—like creating, opening, saving, editing, and searching documents.
  - Using this pre-established framework means that developers don't need to build these features from scratch. Instead, they focus on customizing and enhancing specific functionalities to meet particular needs.
- **Business Implications:**
  - The time saved by using a standard framework translates directly into cost savings and faster time-to-market—an essential factor in business settings where speed and efficiency are paramount.

Utilizing Framework Components via Documentation and APIs

- **Accessing Framework Features:**
  - Utilizing components like dialog boxes within a development framework involves understanding and following the guidelines provided by the framework's documentation.
  - Documentation typically includes detailed instructions on using the public interfaces of classes or class libraries within the framework.
- **Using API Documentation:**
  - For example, in Java development, a developer would refer to the API documentation for the 'JMenuBar' class to create a menu bar.
  - The API guides provide essential information on the methods available, their usage, parameter requirements, and the expected behavior of the components.
- **Practical Implementation:**
  - By adhering to the standards outlined in the API, developers ensure that their applications or applets comply with the expected norms and can operate effectively.
  - This process exemplifies how standardized APIs facilitate the reuse of sophisticated functionalities without needing to reinvent them, speeding up development.

Utilizing Framework Components via Documentation and APIs

- **Code Reuse in Frameworks:**
  - While earlier chapters discussed code reuse in terms of inheritance (one class inheriting behaviors from another), this chapter expands the concept to include using entire systems or subsystems provided by frameworks.
  - Frameworks offer a higher level of abstraction, allowing for the reuse of more complex and integrated functionalities which are well-tested and standardized.

Code Reuse Revisited

This chapter not only revisits the concept of code reuse but also extends it to encompass the reuse of whole or partial systems via frameworks, illustrating a broader application of reuse principles beyond simple inheritance.

What Is a Contract?

Understanding Contracts in Software Development

- **Definition of a Contract:**
  - In software development, a contract refers to the specifications developers must follow when using an Application Programming Interface (API) or a framework. These specifications include method names, parameter counts, and expected behaviors.
  - Similar to a legal contract, a software contract involves an enforceable agreement within the context of project management and industry standards.
- **Role of APIs as Frameworks:**
  - An API can often be seen as a framework because it provides a structured way to access a set of functionalities within a software system, dictating software components interaction.
  - Compliance with an API ensures that different software system components can work together effectively, promoting interoperability and reliability.

The Term Contract

While "contract" has various implications in business and law, in the context of software development discussed here, it specifically refers to the need for developers to adhere to the predefined standards and specifications of APIs and frameworks.

Understanding Contracts in Software Development

- **Importance of Compliance:**
  - By adhering to the defined contracts of an API, developers create software that is consistent with expected standards, which facilitates maintenance, scalability, and integration with other systems or components.
  - Contracts help maintain a high standard of development by ensuring that all participants in a project are clear about their roles and responsibilities concerning the software's architecture and functionality.
- **Clarification of Terms:**
  - It is essential to distinguish this use of the term "contract" from other potential meanings in software design, such as design by contract or other theoretical or conceptual frameworks.
- Enforcement is vital because it is always possible for a developer to break a contract. Without enforcement, a rogue developer could decide to reinvent the wheel and write her code rather than use the specifications provided by the framework.

Abstract Classes

Implementing Contracts with Abstract Classes

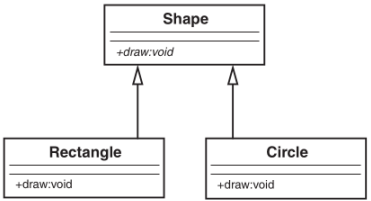
- **Concept of Abstract Classes:**
  - An abstract class is a class that cannot be instantiated on its own and typically contains one or more abstract methods. These methods are declared but not implemented within the abstract class itself—instead, they must be implemented by subclasses.
  - Example: An abstract class ‘Shape‘ might have an abstract method ‘draw()‘. The ‘Shape‘ class cannot be instantiated because it represents a general concept of a shape, not a specific one.
- **Abstract Classes as Contracts:**
  - In the context of a contract, an abstract class defines a set of methods that all subclasses agree to implement. This sets a standard or contract for what behaviors (methods) are compulsory for any concrete subclass.
  - By using an abstract class, you ensure that all specific shapes, such as circles or rectangles, adhere to the same interface, i.e., they all implement a ‘draw()‘ method, thus fulfilling the contract.

Implementing Contracts with Abstract Classes

- **Application in Drawing Shapes:**
  - To create an application that can draw various shapes, one would define an abstract class ‘Shape‘ with an abstract ‘draw()‘ method. Each specific shape class (like ‘Circle‘, ‘Rectangle‘) would then implement this method according to its particular form.
- **Advantages of Using Abstract Classes:**
  - **Flexibility:** New shapes can be added easily by simply creating new subclasses of ‘Shape‘. Each new class must implement the ‘draw()‘ method, ensuring it fits seamlessly into the existing framework.
  - **Consistency:** Abstract classes enforce a consistent interface in all subclasses. This makes it easier to handle a variety of shapes polymorphically, enhancing code reuse and modularity.

Standardization and Responsibility in Shape Drawing

- **Standardizing Method Invocation:** The abstract class ‘Shape‘ defines a standard method ‘draw()‘ that all subclasses must implement. This consistency allows developers to interact with different shape types more efficiently, as they can expect all shape objects to respond to the ‘draw()‘ method.
- **Encouraging Class Responsibility:** Each shape class, such as ‘Circle‘ or ‘Rectangle‘, provides its own specific implementation of the ‘draw()‘ method. This is in line with the principle that each class should manage its own behavior.
- **Impact on Development:** This methodological consistency not only aids in maintaining clean and understandable code but also enhances collaboration among developers who can easily understand and use each other’s code.



Constructing an object.

Implementing Polymorphism in a Shape Framework

Polymorphic Behavior through Abstract Classes:

- The ‘Shape‘ class is designed as an abstract class with an abstract method ‘draw()‘. This setup ensures that while ‘Shape‘ can outline what needs to be done, it does not dictate how it is to be done, leaving that responsibility to its subclasses.
- This approach allows the ‘Shape‘ class to be extended by various shapes, each with a unique implementation of the ‘draw()‘ method, embodying the polymorphic nature of the framework.

```
public abstract class Shape {
    public abstract void draw(); // no
        implementation
}
```

```
public class Circle extends Shape {
    public void draw() {
        System.out.println("Drawing a
            circle.");
    }
}
public class Rectangle extends Shape {
    public void draw() {
        System.out.println("Drawing a
            rectangle.");
    }
}
```

Conforming to the Shape Contract:

- Subclasses like ‘Circle‘ and ‘Rectangle‘ implement the ‘draw()‘ method to provide specific drawing behaviors that reflect their shapes.
- Each class’s implementation of ‘draw()‘ is a direct response to the method’s invocation, demonstrating polymorphism — the method ‘draw()‘ behaves differently depending on the object’s class.

Implementing Polymorphism in a Shape Framework

- **Practical Usage:**
  - In practice, this allows objects of type ‘Circle‘ or ‘Rectangle‘ to be used interchangeably where a ‘Shape‘ is expected. Developers can add new shapes to the system without altering existing code, simply by creating new subclasses of ‘Shape‘.
  - Example of using polymorphism:

```
Shape circle = new Circle();
Shape rectangle = new Rectangle();
circle.draw();     // Outputs: Drawing a circle.
rectangle.draw(); // Outputs: Drawing a rectangle.
```

- **Benefits of Polymorphism:**
  - This design minimizes code redundancy and enhances flexibility, enabling a system where new functionality can be added with minimal impact on existing code.
  - It promotes an intuitive understanding among developers, who can rely on the uniform interface of the ‘Shape‘ contract while appreciating the unique behaviors of its subclasses.

Fulfilling Contracts in Object-Oriented Design

- **Implementation of Contracts:**
  - Subclasses like ‘Circle‘ and ‘Rectangle‘ extend the abstract ‘Shape‘ class and are required to provide their own implementation of the ‘draw()‘ method.

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

  - This adherence to the abstract method requirement ensures that each subclass conforms to the expected functionality, satisfying the contract established by the ‘Shape‘ class.
- **Enforcement and Compliance:**
  - If a subclass like ‘Circle‘ fails to implement the ‘draw()‘ method, it will not compile, thus failing to fulfill its contract with ‘Shape‘. This enforcement guarantees that all shape objects behave as expected within the system.
  - Project managers can mandate that all shape classes in the application inherit from

- **Structured Analogy and OO Design Strengths:**
  - Incorporating all possible shapes' drawing logic within a single 'Shape' class using conditional statements (like a switch-case) would lead to cluttered and hard-to-maintain code.
  - Object-oriented design avoids this complexity by delegating the responsibility of implementing specific behaviors to the subclasses. This approach not only keeps the code cleaner but also enhances maintainability and scalability.

Structured Analogy

This scenario illustrates the strength of an object-oriented design in promoting clean, maintainable, and scalable code structures by leveraging polymorphism and inheritance effectively.

Abstract Classes: Combining Abstract and Concrete Methods

- **Functionality of Abstract Classes:**
  - Abstract classes can contain a mix of abstract and concrete methods. This flexibility allows abstract classes to define a partial implementation while leaving specific details to be defined by subclasses.
  - For example, a ‘Shape‘ class can provide a concrete implementation for setting the color of a shape, which is common across all shapes, while leaving the implementation of the ‘draw()‘ method abstract for customization by each shape.

```
public abstract class Shape {
    private String color;

    public void setColor(String color) {
        this.color = color;
    }

    public abstract void draw();
}
```

Abstract Classes: Combining Abstract and Concrete Methods

- **Inheritance and Implementation Requirements:**
  - Subclasses like ‘Circle‘ and ‘Rectangle‘ inherit the ‘setColor()‘ method and are required to implement the abstract ‘draw()‘ method. Failure to implement this method would result in the subclass being considered abstract itself.

```
public class Circle extends Shape {
    public void draw() {
        System.out.println("Drawing a Circle");
    }
}
```

- **Implications of Non-Implementation:**
  - If a subclass like ‘Circle‘ does not implement the ‘draw()‘ method, it remains abstract, and another subclass would need to extend ‘Circle‘ and provide an implementation for ‘draw()‘.

Abstract Classes: Combining Abstract and Concrete Methods

- **Understanding Contractual Obligations:**
  - The contract in this context is defined by the abstract methods which must be implemented by the concrete subclasses. This is distinct from interfaces which we will discuss next.

Circle

A failure by ‘Circle‘ to implement ‘draw()‘ means it cannot be instantiated and requires further subclassing to provide a concrete ‘draw()‘ implementation.

Caution

The relationship between ‘Shape‘, ‘Circle‘, and ‘Rectangle‘ is a classical inheritance scenario (is-a), unlike interfaces or composition (has-a), which define different types of contractual relationships.

Interfaces

Understanding Interfaces in Java and .NET

- **Interfaces vs. Abstract Classes:**
  - While C++ uses abstract classes to achieve the functionality of interfaces due to its support for multiple inheritance, Java and .NET introduced interfaces to circumvent the limitations imposed by single inheritance.
  - An interface in Java and .NET can be seen as a contract that any implementing class must fulfill, defining a set of methods without any implementation.
- **Need for Interfaces in Java and .NET:**
  - Since Java and .NET do not support multiple inheritance of classes, interfaces provide a way for a class to inherit from one class while simultaneously adhering to multiple contracts, or interfaces.
  - This allows for greater flexibility in designing systems and promotes loose coupling between components.

Understanding Interfaces in Java and .NET

- **Definition and Function of an Interface:**
  - An interface is strictly a syntactical construct specific to a programming language that specifies what methods a class must implement, without specifying how.
  - Interfaces ensure that certain methods are present in classes that implement them, thus guaranteeing a particular behavior or set of behaviors.
- **Clarification of Terms:**
  - It's crucial to differentiate between the general term "interface" referring to any public methods a class offers, and the specific programming construct "interface" that defines a formal, structured method list without implementations.

Interface Terms

In software terminology, "interface" can refer broadly to the public interface of a class (its publicly accessible methods) or more narrowly to a language-specific construct that classes can implement, distinct from class inheritance.

Interfaces as Contracts in Software Design

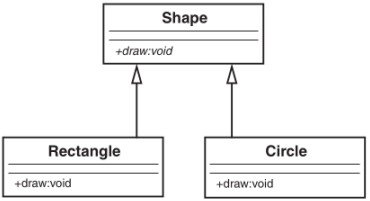
- **Interfaces as Contracts:**
  - Interfaces, much like abstract classes, provide a mechanism to enforce contracts within a framework. However, unlike abstract classes, interfaces do not offer any method implementations; they only specify method signatures that must be implemented by the classes that agree to the contract.
- **Example of an Interface:**
  - Consider an interface ‘Nameable‘ which specifies that any class implementing it must provide a mechanism to handle naming. The interface might include methods like ‘setName(String name)‘ and ‘getName()‘.

```
public interface Nameable {
    void setName(String name);
    String getName();
}
```

  - In UML, ‘Nameable‘ would be represented with its methods listed, typically with the stereotype ‘≪interface≫‘ to distinguish it from classes.

Interfaces as Contracts in Software Design

- **Misconceptions About Interfaces:**
  - While interfaces provide a way to implement polymorphism and can simulate aspects of multiple inheritance by allowing classes to implement multiple interfaces, they are not a direct substitute for multiple inheritance.
  - Interfaces should be seen as a design choice that offers flexibility and promotes loose coupling between components, rather than merely a workaround for the lack of multiple inheritance in languages like Java and .NET.



Constructing an object.

Circle

Interfaces are not just workarounds for lack of multiple inheritance but are fundamental to designing loosely coupled systems. They allow for the extension and integration of features without compromising the codebase's stability and maintainability.

Tying It All Together

Abstract Classes vs. Interfaces: Understanding Their Distinctions

- **Abstract Classes:**
  - **Definition:** Abstract classes are classes that can have both abstract methods (without implementation) and concrete methods (with implementation).
  - **Example:**

```
public abstract class Mammal {
    public void generateHeat() {
        System.out.println("Generate heat");
    }
    public abstract void makeNoise();
}
```

  - **Use Case:** Abstract classes are used when there are shared behaviors among subclasses with some common implementation. For instance, all mammals generate heat, so this method is implemented in the ‘Mammal‘ class.

Abstract Classes vs. Interfaces: Understanding Their Distinctions

- **Interfaces:**
  - **Definition:** Interfaces are contracts that specify what methods a class must implement, without providing any implementation details.
  - **Characteristic:** Interfaces contain only abstract method declarations.
  - **Implication:** Since interfaces only declare methods without defining any concrete behavior, they are used when different objects need to be treated in the same way but may have completely different implementations.
- **Comparison:**
  - While both abstract classes and interfaces can specify abstract methods, the key difference lies in the allowance for concrete methods. Abstract classes can provide some method implementations, useful for sharing code among closely related classes.
  - Interfaces, on the other hand, ensure total abstraction and are ideal for defining a set of actions that different classes can perform, but possibly in very different ways.
- **Design Consideration:** Choosing between an abstract class and an interface depends on the design requirement. If you need to capture some common behavior among all subclasses: abstract class. If you need to ensure a certain interface for unrelated classes: interface.

Object Building in Java and .NET

In a nutshell, Java and .NET build objects in three ways: inheritance, interfaces, and composition. Note the dashed line in Figure that represents the interface. This example illustrates when you should use each of these constructs.

- **When do you choose an abstract class?**
- **When do you choose an interface?**
- **When do you choose composition?**

Let's explore further. You should be familiar with the following concepts:

- **Inheritance:** Dog is a Mammal, so the relationship is inheritance.
- **Interface:** Dog implements Nameable, so the relationship is an interface.
- **Composition:** Dog has a Head, so the relationship is composition.

The question is: where does the interface fit in?

Object Building in Java and .NET: Example of Dog Class

To answer this question and tie everything together, let's create a class called `Dog` that is a subclass of `Mammal`, implements `Nameable`, and has a `Head` object (see Figur).

In a nutshell, Java and .NET build objects in three ways.

- Inheritance: If Dog is a Mammal, then inheritance.
- Interface: If Dog implements Nameable, then interface.
- Composition: If Dog has a Head, then composition.

Note the dashed line in Figure that represents the interface. This example illustrates when you should use each of these constructs:

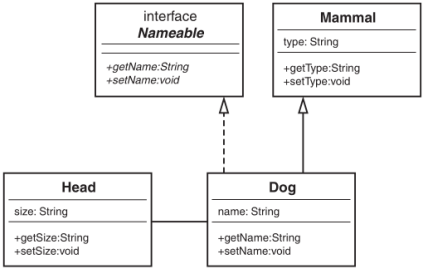


Figure 1: Caption

- When do you choose an abstract class?
- When do you choose an interface?
- When do you choose composition?

Let's explore further.

Example Code:

```
public class Dog extends Mammal implements Nameable {
    String name;
    Head head;
    public void makeNoise() { System.out.println("Bark"); }
    public void setName(String aName) { name = aName; }
    public String getName() { return (name); }
}
```

Discussion: After looking at the UML diagram, you might come up with an obvious question: *"Even though the dashed line from Dog to Nameable represents an interface, isn't it still inheritance?"* At first glance, the answer is not simple. Although interfaces are a special type of inheritance, it is important to know what *special* means. Understanding these special differences are key to a strong object-oriented design.

Understanding Inheritance and Interfaces

Inheritance:

- A dog is a mammal.
- A reptile is not a mammal.
- Thus, a Reptile class could not inherit from the Mammal class.

Interfaces:

- A dog is nameable.
- A lizard is nameable.
- Interfaces transcend the various classes, providing a way to use common functionality among unrelated classes.

Key Differences:

- Classes in a strict inheritance relationship must be directly related, e.g., the Dog class is directly related to the Mammal class.
- Interfaces can be used for classes that are not related. You can name a dog just as well as you can name a lizard.

Abstract Class vs Interface:

- **Abstract Class:** Represents some sort of implementation. For example, Mammal might provide a concrete method called `generateHeat()`, which all mammals do.
- **Interface:** Models only behavior. It never provides any type of implementation but specifies behavior that is the same across classes that conceivably have no connection, such as dogs, cars, and planets being nameable.

Making a Contract

The Importance of Standardized Coding Conventions - Non-Standardized Examp

```java
public class Planet {
    String planetName;
    public void getplanetName() { return planetName; };
}
```

```java
public class Car {
    String carName;
    public String getCarName() { return carName; };
}
```

```java
public class Dog {
    String dogName;
    public String getDogName() { return dogName; };
}
```

Discussion: Each class, Planet, Car, and Dog, implements code to name the entity. However, because they are all implemented separately, each class has different syntax to retrieve the name. This can lead to inconsistencies and confusion in a larger codebase. Standardizing coding conventions, such as through a consistent interface for naming, can greatly improve clarity and maintainability.

Standardizing Naming Conventions with the Nameable Interface

Problem of Non-Standardized Naming Conventions:

- Different classes (Planet, Car, Dog) have different methods to retrieve the name of an object.
- Users need to consult documentation to understand how to access names in each class—a cumbersome process.

Introducing the Nameable Interface:

```
public interface Nameable {
    String getName();
}
```

Standardizing Naming Conventions with the Nameable Interface

Implementation Example:

- Each class implements the `Nameable` interface, providing a uniform `getName()` method.
- This standardizes the method to access names across all classes, enhancing code maintainability and usability.

Advantages:

- Reduces the need to consult documentation for basic functionalities.
- Facilitates easier and more intuitive interaction with the classes across different projects or within a company.

Conclusion: By adopting a standardized interface like `Nameable`, all classes share the same naming syntax, simplifying the user experience and improving code organization.

Implementing the Nameable Interface

```
public interface Nameable {
    public String getName();
    public void setName(String aName);
}
```

Implementing Nameable in Classes:

```
public class Planet implements Nameable {
    String planetName;
    public String getName() { return planetName; }
    public void setName(String myName) { planetName = myName; }
}
```

```
public class Car implements Nameable {
    String carName;
    public String getName() { return carName; }
    public void setName(String myName) { carName = myName; }
}
```

```
public class Dog implements Nameable {
    String dogName;
    public String getName() { return dogName; }
    public void setName(String myName) { dogName = myName; }
}
```

Benefits: By standardizing the interface for naming entities, all classes provide a uniform and intuitive way to access and set names.

Adhering to Contracts in Programming - Challenges and Enforcement

The Importance of Contracts: Contracts in programming, like interfaces, ensure a standard way of coding which is crucial for maintaining code quality and reliability. However, contracts rely on all parties following the agreed standards.

What if someone breaks the contract?

- While there is nothing inherently stopping a rogue programmer from deviating from the standards, doing so can lead to serious consequences.
- On a team level, a project manager can enforce standards—similar to variable naming conventions or using a common configuration management system.

Consequences of Non-Compliance:

- Non-compliance might lead to reprimands or more severe penalties like termination from the project.
- Breaking a programming contract, such as not implementing a required method in an interface, can result in unusable code.

Example: Java Runnable Interface

Consequences of not implementing `run()`

Java applets implement the `Runnable` interface, which requires the `run()` method. If this method is absent, the browser expecting to call `run()` will find nothing, leading to failures.

Conclusion: Enforcing adherence to contracts is crucial for the stability and functionality of software, especially in environments where multiple components interact dynamically.

System Plug-in-Points: Using Contracts in Code

Role of Contracts: Contracts serve as "plug-in points" allowing system design abstraction. By using a contract, systems are not tightly coupled to objects of specific classes but can connect to any object that implements the contract.

Utilization of Contracts: Contracts are beneficial, enabling flexibility and extensibility in code architecture. Examples like the `Nameable` interface demonstrate practical applications of contracts in defining common features across different system components.

Trade-offs of Using Contracts:

- While contracts can facilitate code reuse and promote modular design, they also introduce a level of complexity.
- The complexity arises from managing the interfaces and ensuring that all implementing objects adhere to the agreed specifications.

Conclusion: Identifying the right balance in the use of contracts is crucial. They can significantly enhance system design by offering flexible plug-in points, but careful consideration is needed to avoid overuse and unnecessary complexity.

Conclusion

Conclusion: Building Objects with Inheritance, Interfaces, and Composition

Key Points Recap:

- Understanding how objects are related is crucial in object-oriented design.
- This chapter has covered the primary mechanisms for building objects: inheritance, interfaces, and composition.
- A focus has been placed on designing reusable code through the strategic use of contracts.

Looking Ahead:

- In Chapter 9, *"Building Objects,"* we will complete our object-oriented (OO) journey.
- The next chapter explores how objects that may seem totally unrelated can interact with each other effectively.

Conclusion: The concepts discussed provide a foundation for creating robust and maintainable software architectures, enabling efficient and flexible interactions among components.

MC322 - Object Oriented Programming
Lesson 8.1
Frameworks and Reuse:
Designing with Interfaces and Abstract Classes

Prof. Marcos M. Raimundo
Instituto de Computação - UNICAMP