[Slide 1: Introduction]

Presenter: Good morning/afternoon everyone. Today, I will be discussing two fundamental design patterns in software engineering: the Strategy Pattern and the Template Method Pattern. These patterns are essential for creating flexible and maintainable code.

[Slide 2: Overview of Behavioral Patterns]

Presenter: Both the Strategy Pattern and the Template Method Pattern are behavioral patterns. They help separate algorithms from the specific details of how they are implemented. This allows us to extend the behavior of a system more easily. The key difference is that the Strategy Pattern decides the behavior at runtime using composition, while the Template Method Pattern defines the skeleton of an algorithm at compile time using inheritance.

[Slide 3: Strategy Pattern]

Presenter: Let's start with the Strategy Pattern. The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. The behavior of the context is defined at runtime by delegating it to one of these algorithms. This is achieved through dynamic polymorphism.

[Slide 4: Example of Strategy Pattern - Burger]

Presenter: Consider a burger restaurant where you can choose different types of burgers at runtime. Each burger type, such as chicken, beef, or veggie, is an algorithm that can be selected dynamically. This demonstrates the power of the Strategy Pattern in providing flexibility.

[Slide 5: Template Method Pattern]

Presenter: Next, the Template Method Pattern. This pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses. It allows subclasses to redefine certain steps of an algorithm without changing its overall structure. This is achieved through static polymorphism and inheritance.

[Slide 6: Example of Template Method Pattern - Meal Recipe]

Presenter: An example of the Template Method Pattern is a meal recipe where the general steps of making a meal are defined, but the specific steps, such as cooking methods or ingredients, are left to the subclasses to define. This ensures consistency in the overall process while allowing flexibility in the details.

[Slide 7: Comparison of Strategy and Template Method Patterns]

Presenter: Let's compare the two patterns. The Template Method Pattern uses inheritance, where a subclass inherits and modifies methods from an abstract class. In contrast, the Strategy Pattern uses composition, where an object delegates an action to another object. The Template Method ensures consistency by using a common structure in the base class, while the Strategy Pattern makes it easy to switch strategies at runtime.

[Slide 8: SOLID Principles]

Presenter: Both patterns adhere to the SOLID principles. They support the Single Responsibility Principle (SRP) by having a single responsibility per class or interface. They support the Open-Closed Principle (OCP) by allowing the addition of more classes that implement the interfaces without altering existing code. They follow the Liskov Substitution Principle (LSP) by ensuring that subclasses or implementations can be used interchangeably. The Template Method Pattern provides

a form of interface segregation with abstract classes, and both patterns depend on abstractions rather than concrete implementations, adhering to the Dependency Inversion Principle (DIP).

[Slide 9: Practical Considerations]

Presenter: In practice, choosing between these patterns depends on your specific needs. If you need to switch between different behaviors dynamically, the Strategy Pattern is more suitable. If you need to ensure a consistent algorithm structure while allowing certain steps to vary, the Template Method Pattern is a better choice.

[Slide 10: Conclusion]

Presenter: In conclusion, both the Strategy Pattern and the Template Method Pattern provide powerful ways to make your code more flexible and maintainable. By understanding and applying these patterns, you can create software that is easier to extend and adapt to changing requirements.

Strategy Pattern

Definition:

The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. The behavior of the context is defined at runtime by delegating it to one of the algorithms.

Key Points:

- **Encapsulation:** Each algorithm is encapsulated in a separate class.
- **Interchangeability:** Algorithms can be easily swapped.
- **Dynamic Behavior:** The behavior is determined at runtime.

Example - Burger Selection:

Imagine we are creating a system where a customer can choose different types of burgers. We will use the Strategy Pattern to encapsulate the algorithms for different burger preparations.

Template Method Pattern

Definition:

The Template Method Pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses. This allows subclasses to redefine certain steps of the algorithm without changing its structure.

Key Points:

- **Skeleton:** The main algorithm is defined in a base class.
- **Override Steps:** Subclasses override specific steps of the algorithm.
- **Static Behavior:** The structure is determined at compile time.

Example - Meal Recipe:

Consider a scenario where we have a base recipe class and different types of meals that inherit from it. We will use the Template Method Pattern to define the recipe structure.

Comparison: Strategy vs. Template Method

- **Strategy Pattern:**
  - Based on composition.
  - Behavior defined at runtime.
  - Easy to switch strategies dynamically.
- **Template Method Pattern:**
  - Based on inheritance.
  - Structure defined at compile time.
  - Consistent algorithm structure with customizable steps.