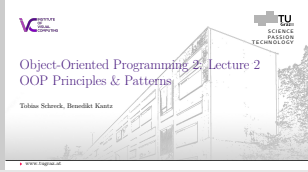
INSTITUTE OF VISUAL COMPUTING

TU Graz
SCIENCE
PASSION
TECHNOLOGY

Object-Oriented Programming 2: Lecture 2 OOP Principles & Patterns

Tobias Schreck, Benedikt Kantz



Which OOP principles do you know?

Which OOP principles do you know?

- Ask students, write down relevant answers on blackboard

Why should we use OOP principles? [1]

Why should we use OOP principles? [1]

- we can tolerate changes
- our team can communicate more effectively $\implies$ easy to understand, concise
- we develop a common “language” - we can understand *any* code!

Why should we use OOP principles? [1]

- resilience
- effective communication
- common “language”

Why should we use OOP principles? [1]
- resilience
- effective communication
- common “language”

- we can tolerate changes
- our team can communicate more effectively $\implies$ easy to understand, concise
- we develop a common “language” - we can understand *any* code!

Why should we use OOP principles? [1]

- resilience
- effective communication
- common “language”

Why should we use OOP principles? [1]
- resilience
- effective communication
- common “language”

- we can tolerate changes
- our team can communicate more effectively $\implies$ easy to understand, concise
- we develop a common “language” - we can understand *any* code!

Why should we use OOP principles? [1]

- resilience
- effective communication
- common “language”

Why should we use OOP principles? [1]
- resilience
- effective communication
- common “language”

- we can tolerate changes
- our team can communicate more effectively $\implies$ easy to understand, concise
- we develop a common “language” - we can understand *any* code!

One *very* popular principle (collection)

Book: [2] R. C. Martin, **Agile principles, patterns, and practices in c,** eng, 2007

One *very* popular principle (collection)
Book: [2] R. C. Martin, **Agile principles, patterns, and practices in c,** eng, 2007

One *very* popular principle (collection)

Book: [2] R. C. Martin, **Agile principles, patterns, and practices in c,** eng, 2007

- **S**ingle-Responsibility
- **O**pen-Closed Principle
- **L**iskov-Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

One *very* popular principle (collection)
Book: [2] R. C. Martin, **Agile principles, patterns, and practices in c,** eng, 2007
- **S**ingle-Responsibility
- **O**pen-Closed Principle
- **L**iskov-Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

One *very* popular principle (collection)

Book: [2] R. C. Martin, **Agile principles, patterns, and practices in c,** eng, 2007

- **S**ingle-Responsibility
- **O**pen-Closed Principle
- **L**iskov-Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

One *very* popular principle (collection)
Book: [2] R. C. Martin, **Agile principles, patterns, and practices in c,** eng, 2007
- **S**ingle-Responsibility
- **O**pen-Closed Principle
- **L**iskov-Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

One *very* popular principle (collection)

Book: [2] R. C. Martin, **Agile principles, patterns, and practices in c,** eng, 2007

- **S**ingle-Responsibility
- **O**pen-Closed Principle
- **L**iskov-Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

One *very* popular principle (collection)
Book: [2] R. C. Martin, **Agile principles, patterns, and practices in c,** eng, 2007
- **S**ingle-Responsibility
- **O**pen-Closed Principle
- **L**iskov-Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

One *very* popular principle (collection)

Book: [2] R. C. Martin, **Agile principles, patterns, and practices in c,** eng, 2007

- **S**ingle-Responsibility
- **O**pen-Closed Principle
- **L**iskov-Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

One *very* popular principle (collection)
Book: [2] R. C. Martin, **Agile principles, patterns, and practices in c,** eng, 2007
- **S**ingle-Responsibility
- **O**pen-Closed Principle
- **L**iskov-Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

One *very* popular principle (collection)

Book: [2] R. C. Martin, **Agile principles, patterns, and practices in c,** eng, 2007

- **S**ingle-Responsibility
- **O**pen-Closed Principle
- **L**iskov-Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

One *very* popular principle (collection)
Book: [2] R. C. Martin, **Agile principles, patterns, and practices in c,** eng, 2007
- **S**ingle-Responsibility
- **O**pen-Closed Principle
- **L**iskov-Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

Single Responsibility Principle (SRP)
Definition

Single Responsibility Principle

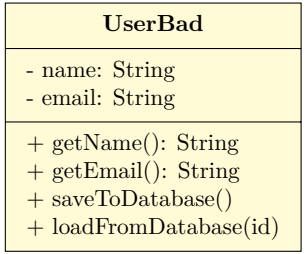
A class should have one, and only one, reason to change.

Single Responsibility Principle (SRP)
Definition
Single Responsibility Principle
A class should have one, and only one, reason to change.

- This means a class should encapsulate a single, well-defined responsibility or concern.
- "Reason to change" often relates to a specific actor or stakeholder group that would request changes related to that responsibility.
- It's about cohesion - keeping related things together and unrelated things separate.

Single Responsibility Principle (SRP)
Example

Violation:



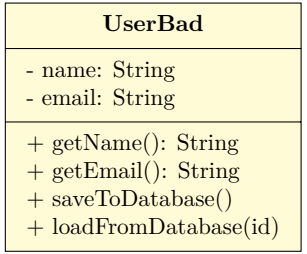
Single Responsibility Principle (SRP)
Example
Violation:

Problem: The `User` class handles both user data *and* data storage logic. Reasons to change: user attributes change OR database technology changes.

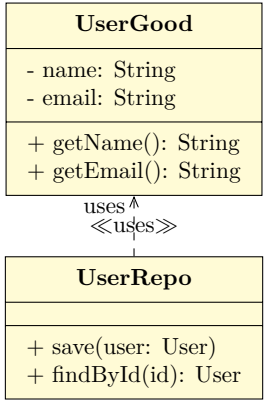
Solution: Separate concerns. `User` holds data. `UserRepository` handles persistence, depending on `User`.

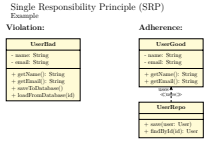
Single Responsibility Principle (SRP)
Example

Violation:



Adherence:





Problem: The `User` class handles both user data *and* data storage logic. Reasons to change: user attributes change OR database technology changes.

Solution: Separate concerns. `User` holds data. `UserRepository` handles persistence, depending on `User`.

Single Responsibility Principle (SRP)
Why?

- **Improved Readability & Understandability:** Classes are smaller and focused on one task.
- **Easier Maintenance:** Changes related to one responsibility are isolated to a single class, reducing the risk of unintended side effects.
- **Enhanced Testability:** Smaller classes with single responsibilities are easier to unit test in isolation.
- **Reduced Coupling:** Classes are less dependent on unrelated concerns.
- **Increased Reusability:** A class focused on one task is more likely to be reusable elsewhere.

Single Responsibility Principle (SRP)
Why?

- **Improved Readability & Understandability**
- Easier Maintenance
- Enhanced Testability
- Reduced Coupling
- Increased Reusability

Single Responsibility Principle (SRP)
Why?
- Improved Readability & Understandability
- Easier Maintenance
- Enhanced Testability
- Reduced Coupling
- Increased Reusability

- **Improved Readability & Understandability:** Classes are smaller and focused on one task.
- **Easier Maintenance:** Changes related to one responsibility are isolated to a single class, reducing the risk of unintended side effects.
- **Enhanced Testability:** Smaller classes with single responsibilities are easier to unit test in isolation.
- **Reduced Coupling:** Classes are less dependent on unrelated concerns.
- **Increased Reusability:** A class focused on one task is more likely to be reusable elsewhere.

Single Responsibility Principle (SRP)
Why?

- **Improved Readability & Understandability**
- **Easier Maintenance**
- Enhanced Testability
- Reduced Coupling
- Increased Reusability

Single Responsibility Principle (SRP)
Why?
- Improved Readability & Understandability
- Easier Maintenance
- Enhanced Testability
- Reduced Coupling
- Increased Reusability

- **Improved Readability & Understandability:** Classes are smaller and focused on one task.
- **Easier Maintenance:** Changes related to one responsibility are isolated to a single class, reducing the risk of unintended side effects.
- **Enhanced Testability:** Smaller classes with single responsibilities are easier to unit test in isolation.
- **Reduced Coupling:** Classes are less dependent on unrelated concerns.
- **Increased Reusability:** A class focused on one task is more likely to be reusable elsewhere.

Single Responsibility Principle (SRP)
Why?

- **Improved Readability & Understandability**
- **Easier Maintenance**
- **Enhanced Testability**
- Reduced Coupling
- Increased Reusability

Single Responsibility Principle (SRP)
Why?
- Improved Readability & Understandability
- Easier Maintenance
- Enhanced Testability
- Reduced Coupling
- Increased Reusability

- **Improved Readability & Understandability:** Classes are smaller and focused on one task.
- **Easier Maintenance:** Changes related to one responsibility are isolated to a single class, reducing the risk of unintended side effects.
- **Enhanced Testability:** Smaller classes with single responsibilities are easier to unit test in isolation.
- **Reduced Coupling:** Classes are less dependent on unrelated concerns.
- **Increased Reusability:** A class focused on one task is more likely to be reusable elsewhere.

Single Responsibility Principle (SRP)
Why?

- **Improved Readability & Understandability**
- **Easier Maintenance**
- **Enhanced Testability**
- **Reduced Coupling**
- Increased Reusability

Single Responsibility Principle (SRP)
Why?
- Improved Readability & Understandability
- Easier Maintenance
- Enhanced Testability
- Reduced Coupling
- Increased Reusability

- **Improved Readability & Understandability:** Classes are smaller and focused on one task.
- **Easier Maintenance:** Changes related to one responsibility are isolated to a single class, reducing the risk of unintended side effects.
- **Enhanced Testability:** Smaller classes with single responsibilities are easier to unit test in isolation.
- **Reduced Coupling:** Classes are less dependent on unrelated concerns.
- **Increased Reusability:** A class focused on one task is more likely to be reusable elsewhere.

Single Responsibility Principle (SRP)
Why?

- **Improved Readability & Understandability**
- **Easier Maintenance**
- **Enhanced Testability**
- **Reduced Coupling**
- **Increased Reusability**

Single Responsibility Principle (SRP)
Why?
- Improved Readability & Understandability
- Easier Maintenance
- Enhanced Testability
- Reduced Coupling
- Increased Reusability

- **Improved Readability & Understandability:** Classes are smaller and focused on one task.
- **Easier Maintenance:** Changes related to one responsibility are isolated to a single class, reducing the risk of unintended side effects.
- **Enhanced Testability:** Smaller classes with single responsibilities are easier to unit test in isolation.
- **Reduced Coupling:** Classes are less dependent on unrelated concerns.
- **Increased Reusability:** A class focused on one task is more likely to be reusable elsewhere.

Open/Closed Principle (OCP)
Definition

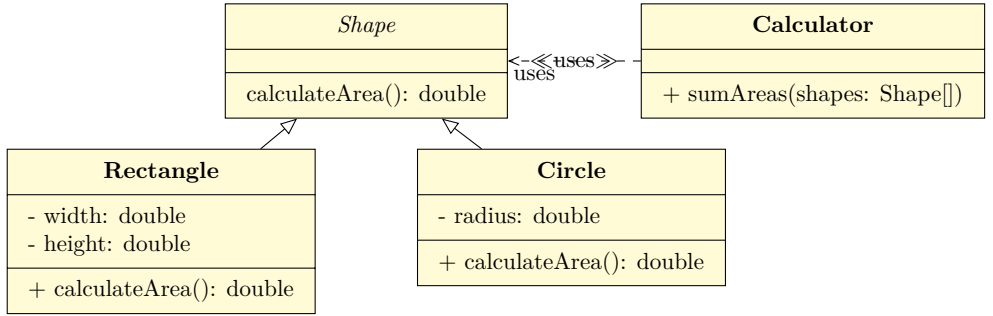
Open/Closed Principle

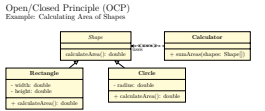
Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

Open/Closed Principle (OCP)
Definition
Open/Closed Principle
Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

- You should be able to add new functionality without changing existing, working code.
- Often achieved through abstraction (abstract classes, interfaces) and polymorphism.
- Existing code (closed for modification) relies on abstractions, while new functionality (open for extension) is added by creating new concrete implementations of those abstractions.

Open/Closed Principle (OCP)
Example: Calculating Area of Shapes





Solution: The `Calculator` depends on the `Shape` abstraction. To add a new shape (e.g., `Triangle`), create a new class inheriting from `Shape` and implement `calculateArea()`. The `Calculator` does **not** need to be modified.

Open/Closed Principle (OCP)
Why?

- **Stability**
- Maintainability
- Flexibility
- Reduced Testing Effort

Open/Closed Principle (OCP)
Why?
- Stability
- Maintainability
- Flexibility
- Reduced Testing Effort

- **Stability:** Reduces the risk of introducing bugs into existing, tested code when adding features.
- **Maintainability:** Changes are localized to new code, not spread across existing modules.
- **Flexibility:** Easier to adapt the system to new requirements.
- **Reduced Testing Effort:** Only the new extensions need rigorous testing; existing code (ideally) doesn't need full re-testing.

Open/Closed Principle (OCP)
Why?

- **Stability**
- **Maintainability**
- Flexibility
- Reduced Testing Effort

Open/Closed Principle (OCP)
Why?
- Stability
- Maintainability
- Flexibility
- Reduced Testing Effort

- **Stability:** Reduces the risk of introducing bugs into existing, tested code when adding features.
- **Maintainability:** Changes are localized to new code, not spread across existing modules.
- **Flexibility:** Easier to adapt the system to new requirements.
- **Reduced Testing Effort:** Only the new extensions need rigorous testing; existing code (ideally) doesn't need full re-testing.

Open/Closed Principle (OCP)
Why?

- **Stability**
- **Maintainability**
- **Flexibility**
- **Reduced Testing Effort**

Open/Closed Principle (OCP)
Why?
- Stability
- Maintainability
- Flexibility
- Reduced Testing Effort

- **Stability:** Reduces the risk of introducing bugs into existing, tested code when adding features.
- **Maintainability:** Changes are localized to new code, not spread across existing modules.
- **Flexibility:** Easier to adapt the system to new requirements.
- **Reduced Testing Effort:** Only the new extensions need rigorous testing; existing code (ideally) doesn't need full re-testing.

Open/Closed Principle (OCP)
Why?

- **Stability**
- **Maintainability**
- **Flexibility**
- **Reduced Testing Effort**

Open/Closed Principle (OCP)
Why?
- Stability
- Maintainability
- Flexibility
- Reduced Testing Effort

- **Stability:** Reduces the risk of introducing bugs into existing, tested code when adding features.
- **Maintainability:** Changes are localized to new code, not spread across existing modules.
- **Flexibility:** Easier to adapt the system to new requirements.
- **Reduced Testing Effort:** Only the new extensions need rigorous testing; existing code (ideally) doesn't need full re-testing.

Liskov Substitution Principle (LSP)
Definition

Liskov Substitution Principle

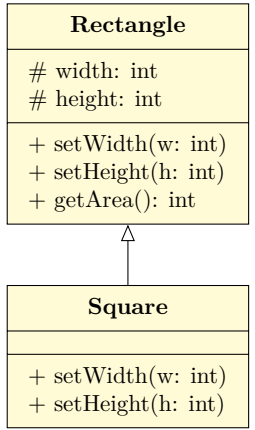
Subtypes must be substitutable for their base types without altering the correctness of the program.

Liskov Substitution Principle (LSP)
Definition
Liskov Substitution Principle
Subtypes must be substitutable for their base types without altering the correctness of the program.

- If class S is a subtype of class T, then objects of type T in a program may be replaced with objects of type S without affecting the desired properties (correctness, task performed, etc.) of that program.
- Focuses on the "is-a" relationship based on *behavior*, not just structure or inheritance syntax.
- Subclasses should fulfill the contract (preconditions, postconditions, invariants) defined by their superclass.

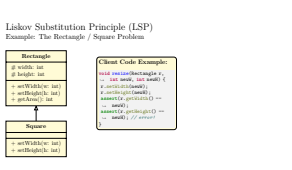
Liskov Substitution Principle (LSP)

Example: The Rectangle / Square Problem



Client Code Example:

```
void resize(Rectangle r,
 ↪  int newW, int newH) {
 r.setWidth(newW);
 r.setHeight(newH);
 assert(r.getWidth() ==
  ↪  newW);
 assert(r.getHeight() ==
  ↪  newH); // error!
}
```

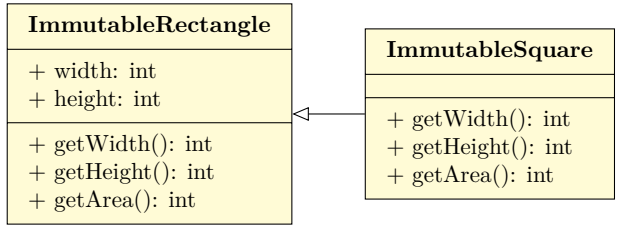



Violation: A `Square` "is-a" `Rectangle` geometrically, but behaviorally it violates the implied contract. A client expecting to set width and height independently on a `Rectangle` will break if given a `Square`. `Square` is not substitutable for `Rectangle` here.

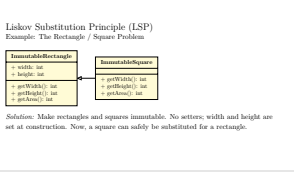
- What could one do to mitigate this issue?
- Solution: introduce a Geometry class above, or reduce "promised" functionality (i.e. width/height immutable!)

Liskov Substitution Principle (LSP)

Example: The Rectangle / Square Problem



Solution: Make rectangles and squares immutable. No setters; width and height are set at construction. Now, a square can safely be substituted for a rectangle.



Violation: A `Square` "is-a" `Rectangle` geometrically, but behaviorally it violates the implied contract. A client expecting to set width and height independently on a `Rectangle` will break if given a `Square`. `Square` is not substitutable for `Rectangle` here.

- What could one do to mitigate this issue?
- Solution: introduce a Geometry class above, or reduce "promised" functionality (i.e. width/height immutable!)

Liskov Substitution Principle (LSP)
Why?

- Reliable Polymorphism
- Correct Inheritance Hierarchies
- Reduced Runtime Errors
- Simplified Client Code

Liskov Substitution Principle (LSP)
Why?
- Reliable Polymorphism
- Correct Inheritance Hierarchies
- Reduced Runtime Errors
- Simplified Client Code

- **Reliable Polymorphism:** Ensures that using subclasses through base class references works as expected.
- **Correct Inheritance Hierarchies:** Helps model true "is-a" behavioral relationships.
- **Reduced Runtime Errors:** Avoids unexpected behavior when substituting subtypes.
- **Simplified Client Code:** Clients can trust that all subtypes behave according to the superclass contract, reducing the need for type checking (‘instanceof‘) or conditional logic.

Liskov Substitution Principle (LSP)
Why?

- **Reliable Polymorphism**
- Correct Inheritance Hierarchies
- Reduced Runtime Errors
- Simplified Client Code

Liskov Substitution Principle (LSP)
Why?
- Reliable Polymorphism
- Correct Inheritance Hierarchies
- Reduced Runtime Errors
- Simplified Client Code

- **Reliable Polymorphism:** Ensures that using subclasses through base class references works as expected.
- **Correct Inheritance Hierarchies:** Helps model true "is-a" behavioral relationships.
- **Reduced Runtime Errors:** Avoids unexpected behavior when substituting subtypes.
- **Simplified Client Code:** Clients can trust that all subtypes behave according to the superclass contract, reducing the need for type checking (‘instanceof‘) or conditional logic.

Liskov Substitution Principle (LSP)
Why?

- **Reliable Polymorphism**
- **Correct Inheritance Hierarchies**
- Reduced Runtime Errors
- Simplified Client Code

Liskov Substitution Principle (LSP)
Why?
- Reliable Polymorphism
- Correct Inheritance Hierarchies
- Reduced Runtime Errors
- Simplified Client Code

- **Reliable Polymorphism:** Ensures that using subclasses through base class references works as expected.
- **Correct Inheritance Hierarchies:** Helps model true "is-a" behavioral relationships.
- **Reduced Runtime Errors:** Avoids unexpected behavior when substituting subtypes.
- **Simplified Client Code:** Clients can trust that all subtypes behave according to the superclass contract, reducing the need for type checking (‘instanceof‘) or conditional logic.

Liskov Substitution Principle (LSP)
Why?

- **Reliable Polymorphism**
- **Correct Inheritance Hierarchies**
- **Reduced Runtime Errors**
- Simplified Client Code

Liskov Substitution Principle (LSP)
Why?
- Reliable Polymorphism
- Correct Inheritance Hierarchies
- Reduced Runtime Errors
- Simplified Client Code

- **Reliable Polymorphism:** Ensures that using subclasses through base class references works as expected.
- **Correct Inheritance Hierarchies:** Helps model true "is-a" behavioral relationships.
- **Reduced Runtime Errors:** Avoids unexpected behavior when substituting subtypes.
- **Simplified Client Code:** Clients can trust that all subtypes behave according to the superclass contract, reducing the need for type checking (‘instanceof‘) or conditional logic.

Liskov Substitution Principle (LSP)
Why?

- **Reliable Polymorphism**
- **Correct Inheritance Hierarchies**
- **Reduced Runtime Errors**
- **Simplified Client Code**

Liskov Substitution Principle (LSP)
Why?
- Reliable Polymorphism
- Correct Inheritance Hierarchies
- Reduced Runtime Errors
- Simplified Client Code

- **Reliable Polymorphism:** Ensures that using subclasses through base class references works as expected.
- **Correct Inheritance Hierarchies:** Helps model true "is-a" behavioral relationships.
- **Reduced Runtime Errors:** Avoids unexpected behavior when substituting subtypes.
- **Simplified Client Code:** Clients can trust that all subtypes behave according to the superclass contract, reducing the need for type checking (‘instanceof‘) or conditional logic.

Interface Segregation Principle (ISP)
Definition

Interface Segregation Principle

Clients should not be forced to depend on methods they do not use.

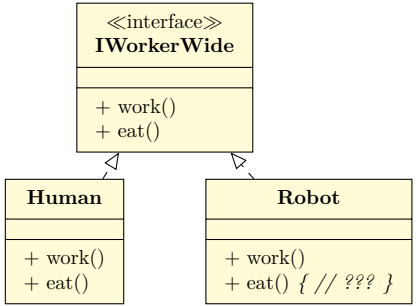
Interface Segregation Principle (ISP)
Definition
Interface Segregation Principle
Clients should not be forced to depend on methods they do not use.

- Prefer many small, client-specific interfaces over one large, general-purpose interface.
- Classes should only need to implement methods that are relevant to them.
- If a class implements an interface with methods it doesn't need, it leads to empty implementations, exceptions, or unnecessary dependencies.

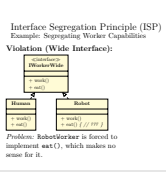
Interface Segregation Principle (ISP)

Example: Segregating Worker Capabilities

Violation (Wide Interface):



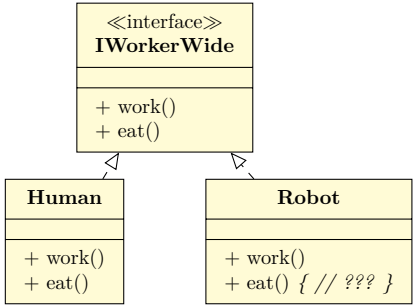
Problem: `RobotWorker` is forced to implement `eat()`, which makes no sense for it.



Interface Segregation Principle (ISP)

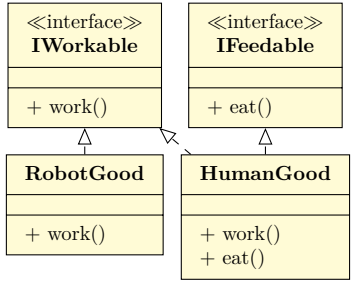
Example: Segregating Worker Capabilities

Violation (Wide Interface):

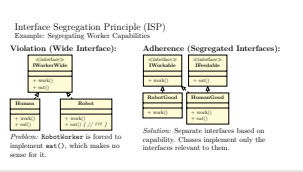


Problem: `RobotWorker` is forced to implement `eat()`, which makes no sense for it.

Adherence (Segregated Interfaces):



Solution: Separate interfaces based on capability. Classes implement only the interfaces relevant to them.



Interface Segregation Principle (ISP)
Why?

- **Reduced Coupling**
- **Improved Cohesion**
- **Increased Flexibility & Reusability**
- **Easier Implementation**
- **Better Design Clarity**

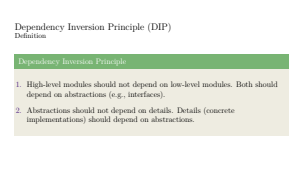
Interface Segregation Principle (ISP)
Why?
- Reduced Coupling
- Improved Cohesion
- Increased Flexibility & Reusability
- Easier Implementation
- Better Design Clarity

- **Reduced Coupling:** Clients only depend on the methods they actually call. Changes to unused methods in an interface won't affect unrelated clients.
- **Improved Cohesion:** Interfaces are more focused and represent specific roles or capabilities.
- **Increased Flexibility & Reusability:** Smaller interfaces are easier to implement and combine.
- **Easier Implementation:** Classes don't need to provide dummy implementations for irrelevant methods.
- **Better Design Clarity:** The roles and capabilities in the system are

Dependency Inversion Principle (DIP)
Definition

Dependency Inversion Principle

1. High-level modules should not depend on low-level modules. Both should depend on abstractions (e.g., interfaces).
2. Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.

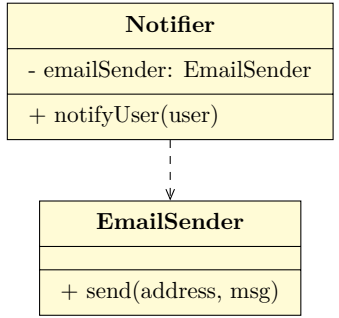


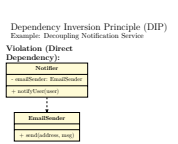
- "Inversion" refers to inverting the traditional dependency direction (High-level → Low-level) to (High-level → Abstraction ← Low-level).
- Decouples policy (high-level logic) from implementation details (low-level specifics).
- Often implemented using Dependency Injection (DI) patterns.

Dependency Inversion Principle (DIP)

Example: Decoupling Notification Service

Violation (Direct Dependency):



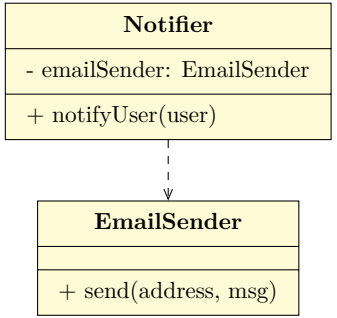


Problem: `NotificationService` is tightly coupled to the concrete `EmailSender`. Changing to SMS requires modifying `NotificationService`.

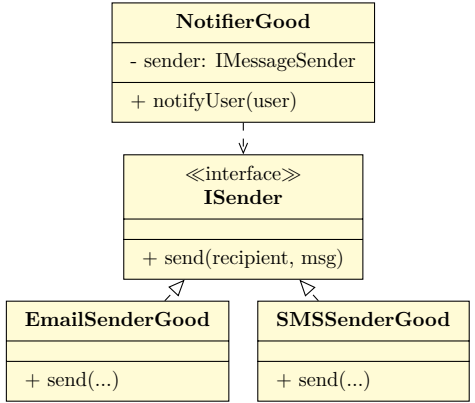
Dependency Inversion Principle (DIP)

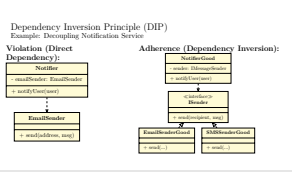
Example: Decoupling Notification Service

Violation (Direct Dependency):



Adherence (Dependency Inversion):





Problem: `NotificationService` is tightly coupled to the concrete `EmailSender`. Changing to SMS requires modifying `NotificationService`.

Solution: Both high-level and low-level depend on the abstraction.

- High-Level: `NotifierGood`
- Low-Level: `EmailSender`, `SMSSender`
- Intermediate: `ISender`

Dependency Inversion Principle (DIP)
Why?

- Decoupling
- Flexibility & Extensibility
- Testability
- Reusability
- Parallel Development

Dependency Inversion Principle (DIP)
Why?
- Decoupling
- Flexibility & Extensibility
- Testability
- Reusability
- Parallel Development

- **Decoupling:** High-level modules are isolated from changes in low-level implementation details.
- **Flexibility & Extensibility:** Easy to swap concrete implementations (e.g., change database, logging framework) without modifying high-level code.
- **Testability:** Allows substituting real dependencies with mock objects or test doubles for unit testing high-level modules in isolation.
- **Reusability:** Both high-level and low-level modules (implementing abstractions) become more reusable.
- **Parallel Development:** Different teams can work on high-level logic

Dependency Inversion Principle (DIP)
Why?

- **Decoupling**
- Flexibility & Extensibility
- Testability
- Reusability
- Parallel Development

Dependency Inversion Principle (DIP)
Why?
- Decoupling
- Flexibility & Extensibility
- Testability
- Reusability
- Parallel Development

- **Decoupling:** High-level modules are isolated from changes in low-level implementation details.
- **Flexibility & Extensibility:** Easy to swap concrete implementations (e.g., change database, logging framework) without modifying high-level code.
- **Testability:** Allows substituting real dependencies with mock objects or test doubles for unit testing high-level modules in isolation.
- **Reusability:** Both high-level and low-level modules (implementing abstractions) become more reusable.
- **Parallel Development:** Different teams can work on high-level logic

Dependency Inversion Principle (DIP)
Why?

- **Decoupling**
- **Flexibility & Extensibility**
- Testability
- Reusability
- Parallel Development

Dependency Inversion Principle (DIP)
Why?
- Decoupling
- Flexibility & Extensibility
- Testability
- Reusability
- Parallel Development

- **Decoupling:** High-level modules are isolated from changes in low-level implementation details.
- **Flexibility & Extensibility:** Easy to swap concrete implementations (e.g., change database, logging framework) without modifying high-level code.
- **Testability:** Allows substituting real dependencies with mock objects or test doubles for unit testing high-level modules in isolation.
- **Reusability:** Both high-level and low-level modules (implementing abstractions) become more reusable.
- **Parallel Development:** Different teams can work on high-level logic

Dependency Inversion Principle (DIP)
Why?

- **Decoupling**
- **Flexibility & Extensibility**
- **Testability**
- Reusability
- Parallel Development

Dependency Inversion Principle (DIP)
Why?
- Decoupling
- Flexibility & Extensibility
- Testability
- Reusability
- Parallel Development

- **Decoupling:** High-level modules are isolated from changes in low-level implementation details.
- **Flexibility & Extensibility:** Easy to swap concrete implementations (e.g., change database, logging framework) without modifying high-level code.
- **Testability:** Allows substituting real dependencies with mock objects or test doubles for unit testing high-level modules in isolation.
- **Reusability:** Both high-level and low-level modules (implementing abstractions) become more reusable.
- **Parallel Development:** Different teams can work on high-level logic

Dependency Inversion Principle (DIP)
Why?

- **Decoupling**
- **Flexibility & Extensibility**
- **Testability**
- **Reusability**
- Parallel Development

Dependency Inversion Principle (DIP)
Why?
- Decoupling
- Flexibility & Extensibility
- Testability
- Reusability
- Parallel Development

- **Decoupling:** High-level modules are isolated from changes in low-level implementation details.
- **Flexibility & Extensibility:** Easy to swap concrete implementations (e.g., change database, logging framework) without modifying high-level code.
- **Testability:** Allows substituting real dependencies with mock objects or test doubles for unit testing high-level modules in isolation.
- **Reusability:** Both high-level and low-level modules (implementing abstractions) become more reusable.
- **Parallel Development:** Different teams can work on high-level logic and low-level implementations concurrently, relying on the agreed upon

Dependency Inversion Principle (DIP)
Why?

- **Decoupling**
- **Flexibility & Extensibility**
- **Testability**
- **Reusability**
- **Parallel Development**

Dependency Inversion Principle (DIP)
Why?
- Decoupling
- Flexibility & Extensibility
- Testability
- Reusability
- Parallel Development

- **Decoupling:** High-level modules are isolated from changes in low-level implementation details.
- **Flexibility & Extensibility:** Easy to swap concrete implementations (e.g., change database, logging framework) without modifying high-level code.
- **Testability:** Allows substituting real dependencies with mock objects or test doubles for unit testing high-level modules in isolation.
- **Reusability:** Both high-level and low-level modules (implementing abstractions) become more reusable.
- **Parallel Development:** Different teams can work on high-level logic and low-level implementations concurrently, relying on the agreed upon

Summary

- **SRP:** One reason to change per class. (Cohesion)
- **OCP:** Open for extension, closed for modification. (Abstraction, Polymorphism)
- **LSP:** Subtypes must be substitutable for base types. (Behavioral Contract)
- **ISP:** Small, specific interfaces are better than large ones. (Decoupling Clients)
- **DIP:** Depend on abstractions, not concretions. (Decoupling Layers)

Summary

- **SRP:** One reason to change per class. (Cohesion)
- **OCP:** Open for extension, closed for modification. (Abstraction, Polymorphism)
- **LSP:** Subtypes must be substitutable for base types. (Behavioral Contract)
- **ISP:** Small, specific interfaces are better than large ones. (Decoupling Clients)
- **DIP:** Depend on abstractions, not concretions. (Decoupling Layers)

What are the effects?

- Maintainability
- Flexibility
- Testability
- Robustness

What are the effects?
- Maintainability
- Flexibility
- Testability
- Robustness

Applying SOLID principles leads to software that is:

- Easier to maintain and understand.
- More flexible and extensible.
- Simpler to test.
- Less prone to bugs when changes are made.

What are the effects?

- Maintainability
- Flexibility
- Testability
- Robustness

What are the effects?
- Maintainability
- Flexibility
- Testability
- Robustness

Applying SOLID principles leads to software that is:

- Easier to maintain and understand.
- More flexible and extensible.
- Simpler to test.
- Less prone to bugs when changes are made.

What are the effects?

- Maintainability
- Flexibility
- Testability
- Robustness

What are the effects?
- Maintainability
- Flexibility
- Testability
- Robustness

Applying SOLID principles leads to software that is:

- Easier to maintain and understand.
- More flexible and extensible.
- Simpler to test.
- Less prone to bugs when changes are made.

What are the effects?

- Maintainability
- Flexibility
- Testability
- Robustness

What are the effects?
- Maintainability
- Flexibility
- Testability
- Robustness

Applying SOLID principles leads to software that is:

- Easier to maintain and understand.
- More flexible and extensible.
- Simpler to test.
- Less prone to bugs when changes are made.

What are the effects?

- Maintainability
- Flexibility
- Testability
- Robustness

What are the effects?
- Maintainability
- Flexibility
- Testability
- Robustness

Applying SOLID principles leads to software that is:

- Easier to maintain and understand.
- More flexible and extensible.
- Simpler to test.
- Less prone to bugs when changes are made.

What are the effects?

- Maintainability
- Flexibility
- Testability
- Robustness

Note

These are principles, not strict rules. Apply them judiciously where they provide clear benefits. Over-engineering can also be a problem.

What are the effects?

- Maintainability
- Flexibility
- Testability
- Robustness

Note

These are principles, not strict rules. Apply them judiciously where they provide clear benefits. Over-engineering can also be a problem.

Applying SOLID principles leads to software that is:

- Easier to maintain and understand.
- More flexible and extensible.
- Simpler to test.
- Less prone to bugs when changes are made.

Exercise: KI Tool Use for SOLID I

- Use a KI tool to get a code example of one SOLID principle.
- Instructions:
  - Groups of 3-4
  - Select a tool of your choice (ChatGPT, Mistral,...)

  ... or use the Discord Bot!
  - What are your findings?

Exercise: KI Tool Use for SOLID I
- Use a KI tool to get a code example of one SOLID principle.
- Instructions:
  - Groups of 3-4
  - Select a tool of your choice (ChatGPT, Mistral,...)

  ... or use the Discord Bot!
  - What are your findings?

Exercise: KI Tool Use for SOLID II

- Are you satisfied with your example?
- How do you think this will affect the practice (of writing code)?
- Do you spot any drawbacks (of the LLM/principle?)?

Exercise: KI Tool Use for SOLID II
- Are you satisfied with your example?
- How do you think this will affect the practice (of writing code)?
- Do you spot any drawbacks (of the LLM/principle?)?

Any applications for your work?

Any applications for your work?

Composition over Inheritance (CoI)
Definition

Principle

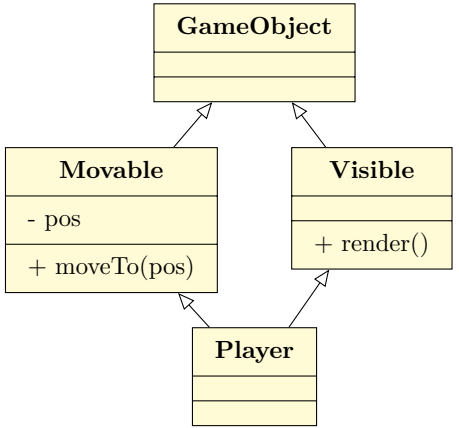
Favor composing objects (HAS-A relationship) over extending classes (IS-A relationship) to achieve polymorphic behavior and code reuse.

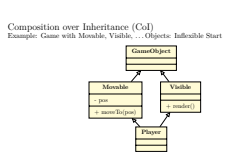
Composition over Inheritance (CoI)
Definition
Principle
Favor composing objects (HAS-A relationship) over extending classes (IS-A relationship) to achieve polymorphic behavior and code reuse.

- Inheritance is a powerful tool, but can lead to rigid hierarchies and the "fragile base class" problem.
- Combines DIP and ISP
- Composition allows for more flexible designs where behavior can be changed at runtime by composing with different objects.
- Often used in conjunction with Dependency Inversion (depending on interfaces of composed objects).

Composition over Inheritance (CoI)

Example: Game with Movable, Visible, . . . Objects: Inflexible Start



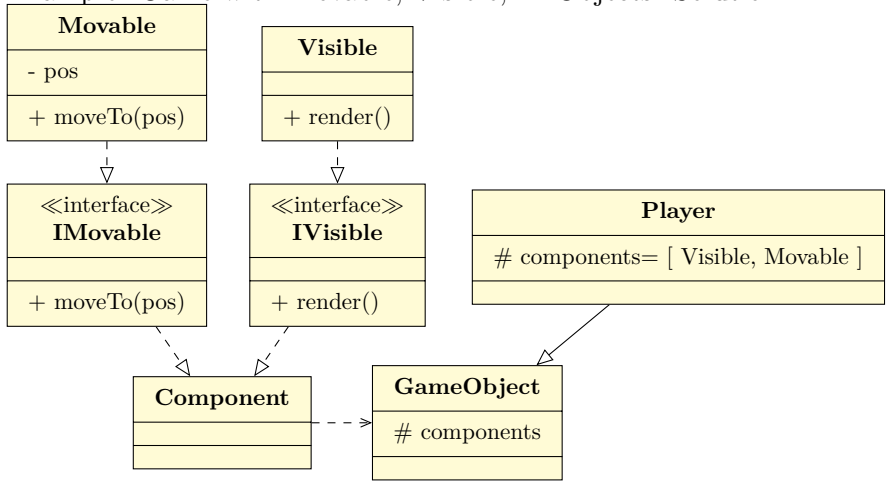


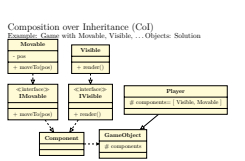
Problem:

Attributes tied to base class $\implies$ hard to compose Visible and Movable Objects.

Composition over Inheritance (CoI)

Example: Game with Movable, Visible, . . . Objects: Solution





Solution:

Composable attributes $\implies$ easy to combine Visible and Movable & expand!

Composition over Inheritance (CoI)
Why?

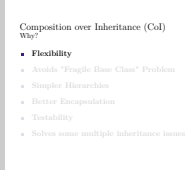
- Flexibility
- Avoids "Fragile Base Class" Problem
- Simpler Hierarchies
- Better Encapsulation
- Testability
- Solves some multiple inheritance issues

Composition over Inheritance (CoI)
Why?
- Flexibility
- Avoids "Fragile Base Class" Problem
- Simpler Hierarchies
- Better Encapsulation
- Testability
- Solves some multiple inheritance issues

- **Flexibility:** Behavior can be defined at runtime by composing with different objects, rather than being fixed at compile time by inheritance.
- **Avoids "Fragile Base Class" Problem:** Changes to a base class can unexpectedly break derived classes. Composition isolates components.
- **Simpler Hierarchies:** Prevents deep and wide inheritance trees which can become hard to manage.
- **Better Encapsulation:** Interactions happen through well-defined interfaces of composed objects, rather than relying on protected members of a base class.
- **Testability:** Individual components can be tested in isolation more easily.

Composition over Inheritance (CoI)
Why?

- **Flexibility**
- Avoids "Fragile Base Class" Problem
- Simpler Hierarchies
- Better Encapsulation
- Testability
- Solves some multiple inheritance issues



- **Flexibility:** Behavior can be defined at runtime by composing with different objects, rather than being fixed at compile time by inheritance.
- **Avoids "Fragile Base Class" Problem:** Changes to a base class can unexpectedly break derived classes. Composition isolates components.
- **Simpler Hierarchies:** Prevents deep and wide inheritance trees which can become hard to manage.
- **Better Encapsulation:** Interactions happen through well-defined interfaces of composed objects, rather than relying on protected members of a base class.
- **Testability:** Individual components can be tested in isolation more easily.

Composition over Inheritance (CoI)

Why?

- **Flexibility**
- **Avoids "Fragile Base Class" Problem**
- Simpler Hierarchies
- Better Encapsulation
- Testability
- Solves some multiple inheritance issues

Composition over Inheritance (CoI)
Why?
- Flexibility
- Avoids "Fragile Base Class" Problem
- Simpler Hierarchies
- Better Encapsulation
- Testability
- Solves some multiple inheritance issues

- **Flexibility:** Behavior can be defined at runtime by composing with different objects, rather than being fixed at compile time by inheritance.
- **Avoids "Fragile Base Class" Problem:** Changes to a base class can unexpectedly break derived classes. Composition isolates components.
- **Simpler Hierarchies:** Prevents deep and wide inheritance trees which can become hard to manage.
- **Better Encapsulation:** Interactions happen through well-defined interfaces of composed objects, rather than relying on protected members of a base class.
- **Testability:** Individual components can be tested in isolation more easily.

Composition over Inheritance (CoI)
Why?

- **Flexibility**
- **Avoids "Fragile Base Class" Problem**
- **Simpler Hierarchies**
- Better Encapsulation
- Testability
- Solves some multiple inheritance issues

Composition over Inheritance (CoI)
Why?
- Flexibility
- Avoids "Fragile Base Class" Problem
- Simpler Hierarchies
- Better Encapsulation
- Testability
- Solves some multiple inheritance issues

- **Flexibility:** Behavior can be defined at runtime by composing with different objects, rather than being fixed at compile time by inheritance.
- **Avoids "Fragile Base Class" Problem:** Changes to a base class can unexpectedly break derived classes. Composition isolates components.
- **Simpler Hierarchies:** Prevents deep and wide inheritance trees which can become hard to manage.
- **Better Encapsulation:** Interactions happen through well-defined interfaces of composed objects, rather than relying on protected members of a base class.
- **Testability:** Individual components can be tested in isolation more easily.

Composition over Inheritance (CoI)
Why?

- **Flexibility**
- **Avoids "Fragile Base Class" Problem**
- **Simpler Hierarchies**
- **Better Encapsulation**
- Testability
- Solves some multiple inheritance issues

Composition over Inheritance (CoI)
Why?
- Flexibility
- Avoids "Fragile Base Class" Problem
- Simpler Hierarchies
- Better Encapsulation
- Testability
- Solves some multiple inheritance issues

- **Flexibility:** Behavior can be defined at runtime by composing with different objects, rather than being fixed at compile time by inheritance.
- **Avoids "Fragile Base Class" Problem:** Changes to a base class can unexpectedly break derived classes. Composition isolates components.
- **Simpler Hierarchies:** Prevents deep and wide inheritance trees which can become hard to manage.
- **Better Encapsulation:** Interactions happen through well-defined interfaces of composed objects, rather than relying on protected members of a base class.
- **Testability:** Individual components can be tested in isolation more easily.

Composition over Inheritance (CoI)
Why?

- **Flexibility**
- **Avoids "Fragile Base Class" Problem**
- **Simpler Hierarchies**
- **Better Encapsulation**
- Testability
- Solves some multiple inheritance issues

Composition over Inheritance (CoI)
Why?
- Flexibility
- Avoids "Fragile Base Class" Problem
- Simpler Hierarchies
- Better Encapsulation
- Testability
- Solves some multiple inheritance issues

- **Flexibility:** Behavior can be defined at runtime by composing with different objects, rather than being fixed at compile time by inheritance.
- **Avoids "Fragile Base Class" Problem:** Changes to a base class can unexpectedly break derived classes. Composition isolates components.
- **Simpler Hierarchies:** Prevents deep and wide inheritance trees which can become hard to manage.
- **Better Encapsulation:** Interactions happen through well-defined interfaces of composed objects, rather than relying on protected members of a base class.
- **Testability:** Individual components can be tested in isolation more easily.

Composition over Inheritance (CoI)
Why?

- **Flexibility**
- **Avoids "Fragile Base Class" Problem**
- **Simpler Hierarchies**
- **Better Encapsulation**
- **Testability**
- Solves some multiple inheritance issues

Composition over Inheritance (CoI)
Why?
- Flexibility
- Avoids "Fragile Base Class" Problem
- Simpler Hierarchies
- Better Encapsulation
- Testability
- Solves some multiple inheritance issues

- **Flexibility:** Behavior can be defined at runtime by composing with different objects, rather than being fixed at compile time by inheritance.
- **Avoids "Fragile Base Class" Problem:** Changes to a base class can unexpectedly break derived classes. Composition isolates components.
- **Simpler Hierarchies:** Prevents deep and wide inheritance trees which can become hard to manage.
- **Better Encapsulation:** Interactions happen through well-defined interfaces of composed objects, rather than relying on protected members of a base class.
- **Testability:** Individual components can be tested in isolation more easily.

Composition over Inheritance (CoI)
Why?

- **Flexibility**
- **Avoids "Fragile Base Class" Problem**
- **Simpler Hierarchies**
- **Better Encapsulation**
- **Testability**
- **Solves some multiple inheritance issues**

Composition over Inheritance (CoI)
Why?
- Flexibility
- Avoids "Fragile Base Class" Problem
- Simpler Hierarchies
- Better Encapsulation
- Testability
- Solves some multiple inheritance issues

- **Flexibility:** Behavior can be defined at runtime by composing with different objects, rather than being fixed at compile time by inheritance.
- **Avoids "Fragile Base Class" Problem:** Changes to a base class can unexpectedly break derived classes. Composition isolates components.
- **Simpler Hierarchies:** Prevents deep and wide inheritance trees which can become hard to manage.
- **Better Encapsulation:** Interactions happen through well-defined interfaces of composed objects, rather than relying on protected members of a base class.
- **Testability:** Individual components can be tested in isolation more easily.

Patterns Overview
Why patterns?

Patterns Overview
Why patterns?

Patterns Overview
Why patterns?

- Similar to Principles!
- Common language
- Repeated use case, (one) of the best approaches!

Patterns Overview
Why patterns?
- Similar to Principles!
- Common language
- Repeated use case, (one) of the best approaches!

Patterns Overview

Which patterns?

Book: [3] S. Stelting, **Applied java patterns,** eng, 2002

- **Singleton**
- **Observer**
- **Builder**
- **Decorator**
- **Adapter**
- **Factories**

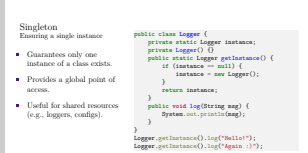
Patterns Overview
Which patterns?
Book: [3] S. Stelting, **Applied java patterns,** eng, 2002
- **Singleton**
- **Observer**
- **Builder**
- **Decorator**
- **Adapter**
- **Factories**

Singleton

Ensuring a single instance

- Guarantees only one instance of a class exists.
- Provides a global point of access.
- Useful for shared resources (e.g., loggers, configs).

```
public class Logger {
    private static Logger instance;
    private Logger() {}
    public static Logger getInstance() {
        if (instance == null) {
            instance = new Logger();
        }
        return instance;
    }
    public void log(String msg) {
        System.out.println(msg);
    }
}
Logger.getInstance().log("Hello!");
Logger.getInstance().log("Again :)");
```

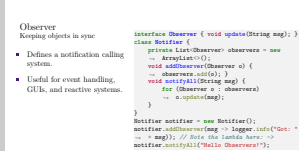



Observer

Keeping objects in sync

- Defines a notification calling system.
- Useful for event handling, GUIs, and reactive systems.

```
interface Observer { void update(String msg); }
class Notifier {
    private List<Observer> observers = new
    ↪  ArrayList<>();
    void addObserver(Observer o) {
    ↪  observers.add(o); }
    void notifyAll(String msg) {
        for (Observer o : observers)
        ↪  o.update(msg);
    }
}
Notifier notifier = new Notifier();
notifier.addObserver(msg -> logger.info("Got: "
↪  + msg)); // Note the lambda here: ->
notifier.notifyAll("Hello Observers!");
```

Builder

Pattern with Lombok's `@Builder`

- The `@Builder` annotation from Lombok generates a builder for your class.
- Simplifies object creation, especially for classes with many fields.
- No need to write boilerplate builder code manually.

```java
import lombok.Builder;
import lombok.ToString;
@Builder
@ToString
public class User {
    private String username;
    private int age;
    private String email;
}
User user = User.builder()
    .username("alice")
    .age(30)
    .email("alice@example.com")
    .build();
```

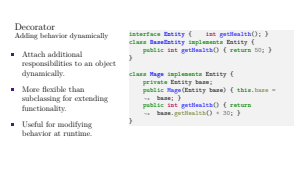

Decorator

Adding behavior dynamically

- Attach additional responsibilities to an object dynamically.
- More flexible than subclassing for extending functionality.
- Useful for modifying behavior at runtime.

```
interface Entity {    int getHealth(); }
class BaseEntity implements Entity {
    public int getHealth() { return 50; }
}

class Mage implements Entity {
    private Entity base;
    public Mage(Entity base) { this.base =
    ↪  base; }
    public int getHealth() { return
    ↪  base.getHealth() + 30; }
}
```

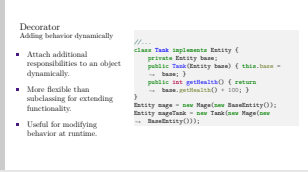



Decorator

Adding behavior dynamically

- Attach additional responsibilities to an object dynamically.
- More flexible than subclassing for extending functionality.
- Useful for modifying behavior at runtime.

```
//...
class Tank implements Entity {
    private Entity base;
    public Tank(Entity base) { this.base =
    ↪  base; }
    public int getHealth() { return
    ↪  base.getHealth() + 100; }
}
Entity mage = new Mage(new BaseEntity());
Entity mageTank = new Tank(new Mage(new
↪  BaseEntity()));
```

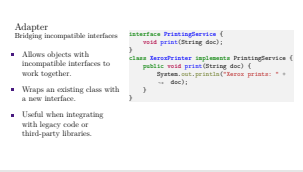



Adapter

Bridging incompatible interfaces

- Allows objects with incompatible interfaces to work together.
- Wraps an existing class with a new interface.
- Useful when integrating with legacy code or third-party libraries.

```java
interface PrintingService {
    void print(String doc);
}
class XeroxPrinter implements PrintingService {
    public void print(String doc) {
        System.out.println("Xerox prints: " +
        ↪  doc);
    }
}
```

Adapter

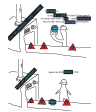
Bridging incompatible interfaces

- Allows objects with incompatible interfaces to work together.
- Wraps an existing class with a new interface.
- Useful when integrating with legacy code or third-party libraries.

```
//...
interface AirPrint {
    void airPrint(String doc);
}
class AirPrintAdapter implements
↪  PrintingService {
    private AirPrint airPrinter;
    public AirPrintAdapter(AirPrint airPrinter)
    ↪  {
        this.airPrinter = airPrinter;
    }
    public void print(String doc) {
        airPrinter.airPrint(doc);
    }
}
```


Factories

Factories

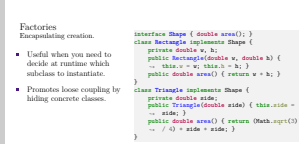


Factories

Encapsulating creation.

- Useful when you need to decide at runtime which subclass to instantiate.
- Promotes loose coupling by hiding concrete classes.

```
interface Shape { double area(); }
class Rectangle implements Shape {
    private double w, h;
    public Rectangle(double w, double h) {
    ↪  this.w = w; this.h = h; }
    public double area() { return w * h; }
}
class Triangle implements Shape {
    private double side;
    public Triangle(double side) { this.side =
    ↪  side; }
    public double area() { return (Math.sqrt(3)
    ↪  / 4) * side * side; }
}
```

Factories

Encapsulating creation.

- Useful when you need to decide at runtime which subclass to instantiate.
- Promotes loose coupling by hiding concrete classes.

```
//...
class ShapeFactory {
    public static Shape create(String type,
    ↪  double side) {
        switch (type) {
            case "square": return new
            ↪  Rectangle(side, side);
            case "circle": return new
            ↪  Circle(side);
            default: throw new
            ↪  IllegalArgumentException("Unknown
            ↪  type: " + type);
        }
    }
}
```


Factories

Encapsulating creation.

- Useful when you need to decide at runtime which subclass to instantiate.
- Promotes loose coupling by hiding concrete classes.

```
//...
Shape s1 = ShapeFactory.create("square", 3);
Shape s2 = ShapeFactory.create("circle", 2);
```


Factories
Encapsulating creation.
- Useful when you need to decide at runtime which subclass to instantiate.
- Promotes loose coupling by hiding concrete classes.

```
//...
Shape s1 = ShapeFactory.create("square", 3);
Shape s2 = ShapeFactory.create("circle", 2);
```

Generics

Type-safe code reuse

- Generics can be limited to subclasses of a base class.
- Example: `findFirst` only works for arrays of `Animal` or its subclasses.

```
class Animal {
    void speak() { System.out.println("..."); }
}
class Dog extends Animal {
    void speak() { System.out.println("Woof!");
    ↪  }
}
```


Generics

Type-safe code reuse

- Generics can be limited to subclasses of a base class.
- Example: `findFirst` only works for arrays of `Animal` or its subclasses.

```
static <T extends Animal> T getFirst(T[] array)
↪  {
    if (array.length > 0) {
        return array[0];
    }
    return null;
}
```


Examples I

- Code examples on Gitlab.
- We will discuss them, looking into different aspect.
- Feel free to expand/go through it at your own pace at home!

Examples I

- Code examples on Gitlab.
- We will discuss them, looking into different aspect.
- Feel free to expand/go through it at your own pace at home!

1.? 2. logger 3. observer

Examples II

- Questions:
  - Any further insights for you?
  - What is a typical application of a singleton?
  - Which Pattern is very useful in reactive systems?

Examples II
- Questions:
  - Any further insights for you?
  - What is a typical application of a singleton?
  - Which Pattern is very useful in reactive systems?

Feedback

Feedback

Bibliography I

References

[1] R. Martin, **Clean Architecture: A Craftsman's Guide to Software Structure and Design** (Robert C. Martin Series), eng, 1st ed. Hoboken: Pearson Education, Limited, 2017, ISBN: 0134494164.

[2] R. C. Martin, **Agile principles, patterns, and practices in c,** eng, 2007.

[3] S. Stelting, **Applied java patterns,** eng, 2002.

Books give good overview, if you want more theory - they are even translated to German if you feel the need to read about Software in German ;)