System Analysis and Design

The Principles of Object Oriented Design: SOLID Principles

Salahaddin University
College of Engineering
Software Engineering Department
2011-2012

Amanj Sherwany
http://www.amanj.me/wiki/doku.php?id=teaching:su:system_analysis_and_design

SOLID Principles

- In computer programming, SOLID is a mnemonic acronym introduced by Robert C. Martin in early 2000s.
- It stands for five basic principles of Object-Oriented Programming and Design.
- The principles when applied together intend to make it more likely that a programmer will create a system that is easy to maintain and extend over time.

SOLID Principles, *Cont'd*

- SOLID, stands for:
  - **S**ingle Responsibility Principle (SRP)
  - **O**pen-Closed Principle (OCP)
  - **L**iskov Substitution Principle (LSP)
  - **I**nterface Segregation Principle (LSP)
  - **D**ependency Inversion Principle (DIP)

Single Responsibility Principle

- There should never be more than one reason for a class to change.
- This principle states that every object should have a single responsibility, and that responsibility should be entirely encapsulated by the class.
- All its services should be narrowly aligned with that responsibility.

What is a Responsibility?

- In the context of the Single Responsibility Principle (SRP) we define a responsibility to be “a reason for change.”
- If you can think of more than one motive for changing a class, then that class has more than one responsibility.
- For example, consider the Modem interface in the next slide:

What is a Responsibility? *Cont'd*

```
interface Modem{
    public void dial(String pno);
    public void hangup();
    public void send(char c);
    public char recv();
}
```

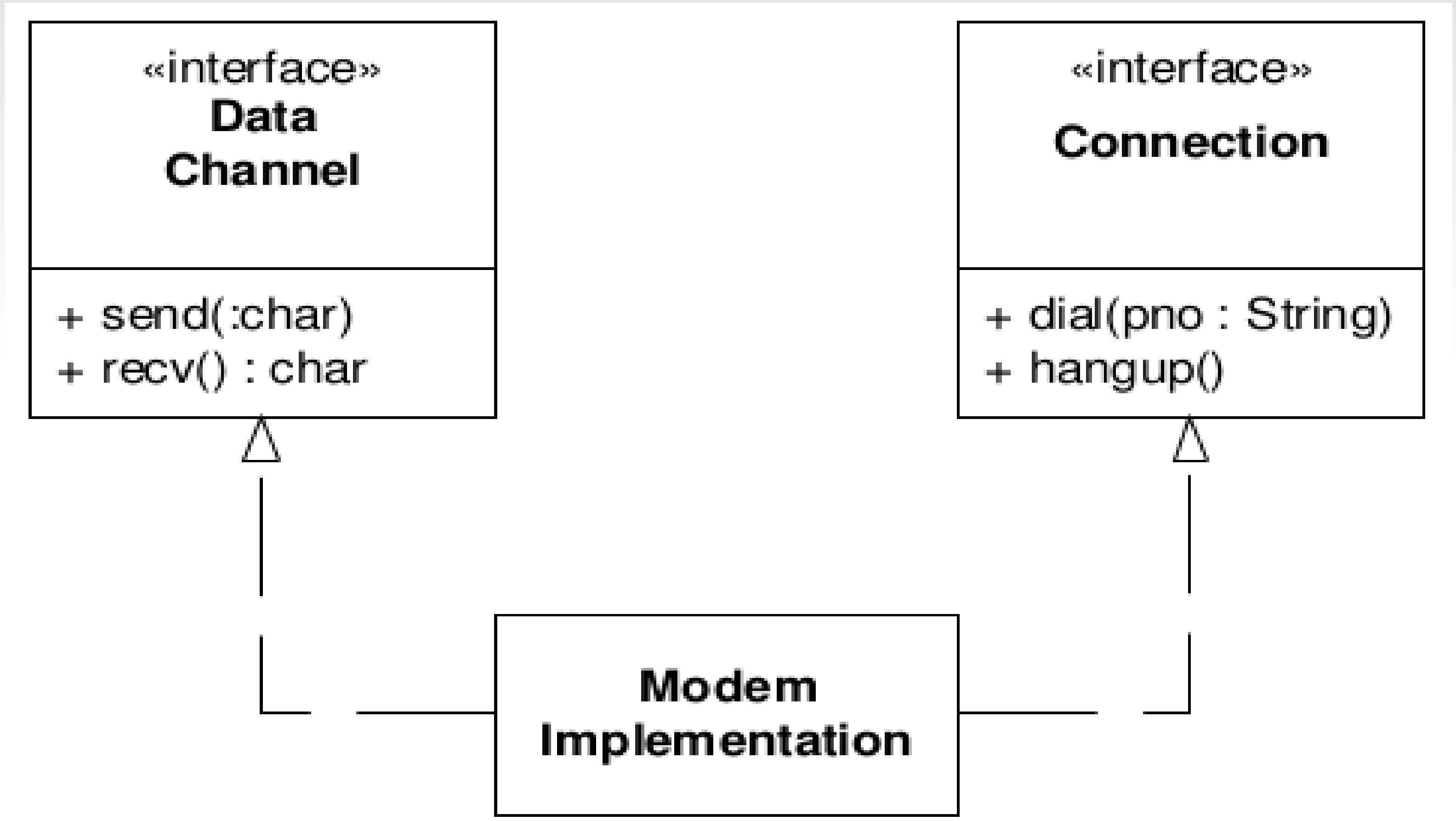
What is a Responsibility? *Cont'd*

- Although all of the four functions belong to a modem
- BUT the class have two responsibilities
  - Connection management responsibility
  - Data communication responsibility
- Should these two responsibilities be separated?

What is a Responsibility? *Cont'd*

YES

What is a Responsibility? *Cont'd*



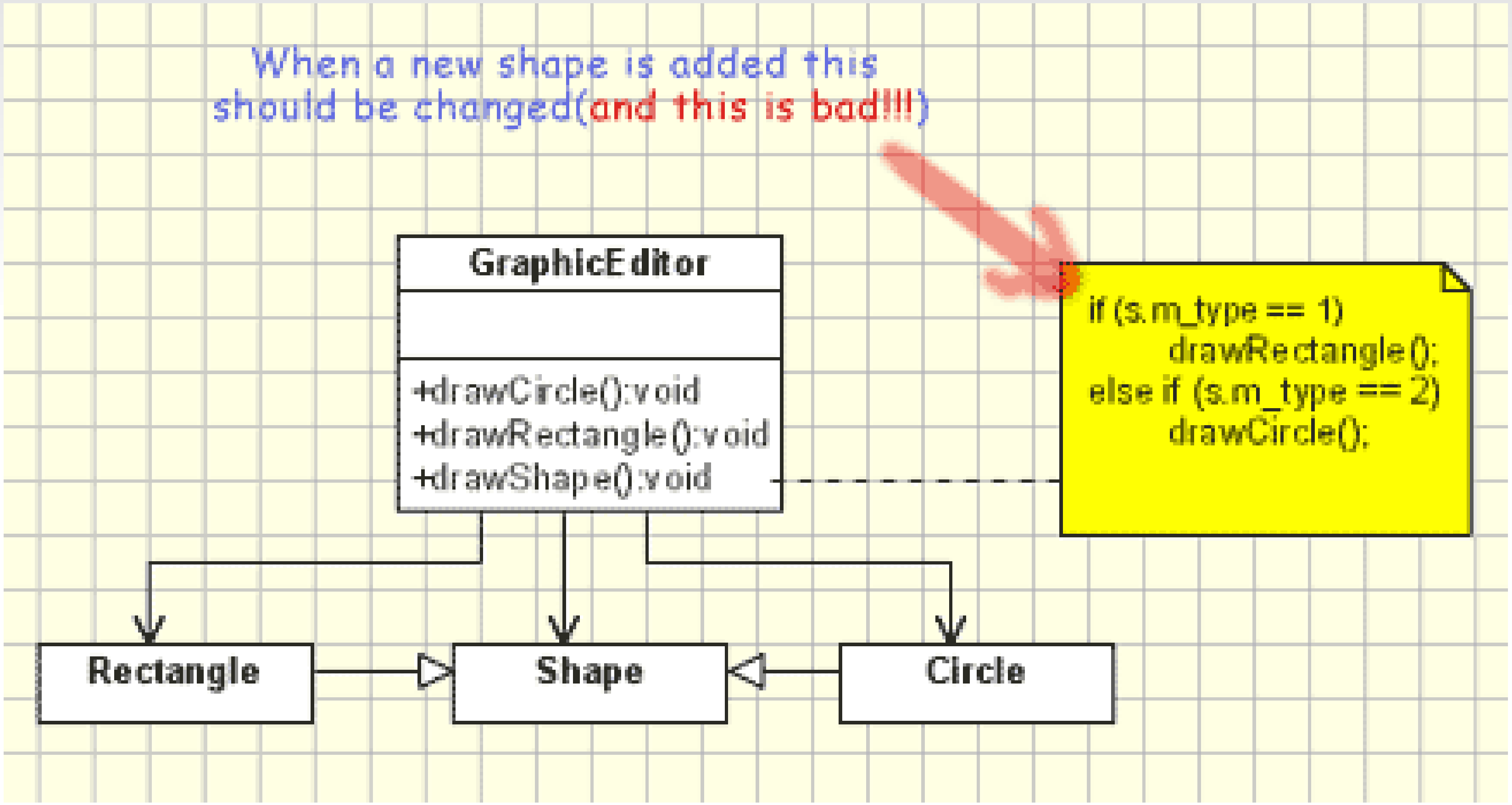
Open/Closed Principle

- In OOP, the Open/Closed Principle (or OCP) states "software entities (class, modules, functions, etc.) should be open for extension, but closed for modification.
- That is, such an entity can allow its behaviour to be modified without altering its source code.

Open/Closed Principle, *Cont'd*

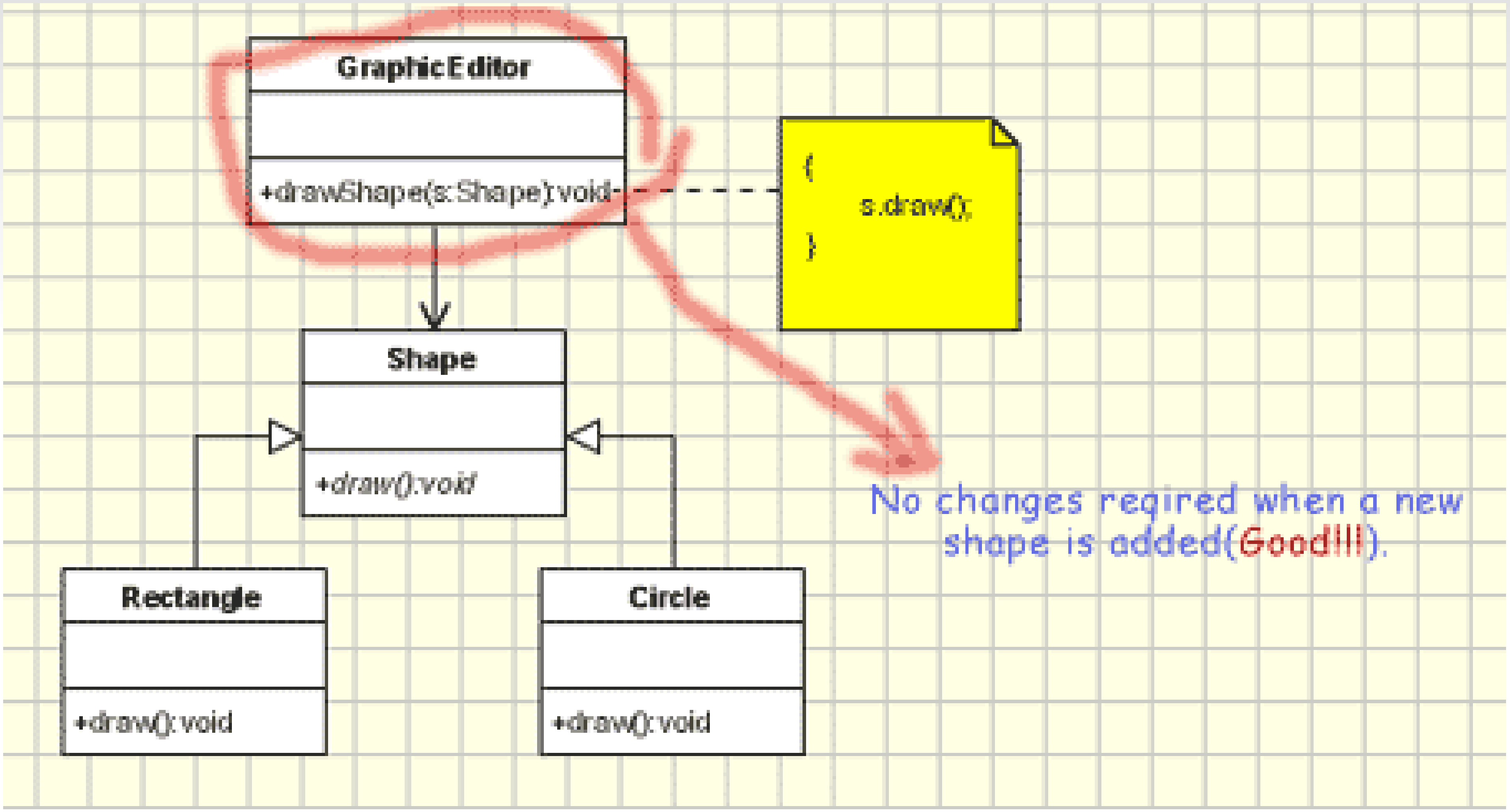
- This is especially valuable in a production environment, where changes to source code may necessitate code review, unit tests, and other such procedures to qualify it for use in a product:
  - Code obeying the principle doesn't change when it is extended, and therefore needs no such effort.

Open/Closed Principle, *Cont'd*



```
if (s.m_type == 1)
        drawRectangle();
else if (s.m_type == 2)
        drawCircle();
```

Open/Closed Principle, *Cont'd*



Liskov Substitution Principle

- Known as LSP.
- States that, in a computer program, if S is a subtype of T, then objects of type T maybe replaced with objects of type S
  - i.e., objects of type S may be substituted for objects of type T, without altering any of the desirable properties of that program.

Liskov Substitution Principle, *Cont'd*

- A typical example that violates LSP is a Square class that derives from a Rectangle class.
  - Assuming getter and setter methods exist for both width and height.
  - The Square class always assumes that the width is equal to the height.
  - If a Square object is used in a context where a Rectangle is expected, unexpected behaviour may occur because the dimensions of a Square cannot be modified independently.

Liskov Substitution Principle, *Cont'd*

- This problem cannot be easily fixed:
  - If we can modify the setter methods in Square class so that they preserve the Square invariant (i.e., keep the dimensions equal), then these methods will weaken (violate) the postconditions for the Rectangle setters, which state that dimensions can be modified independently
- Violations of LSP, like this one, may or not be a problem in practice, depending on the postconditions of invariants that are actually expected by the code that uses classes violating LSP.

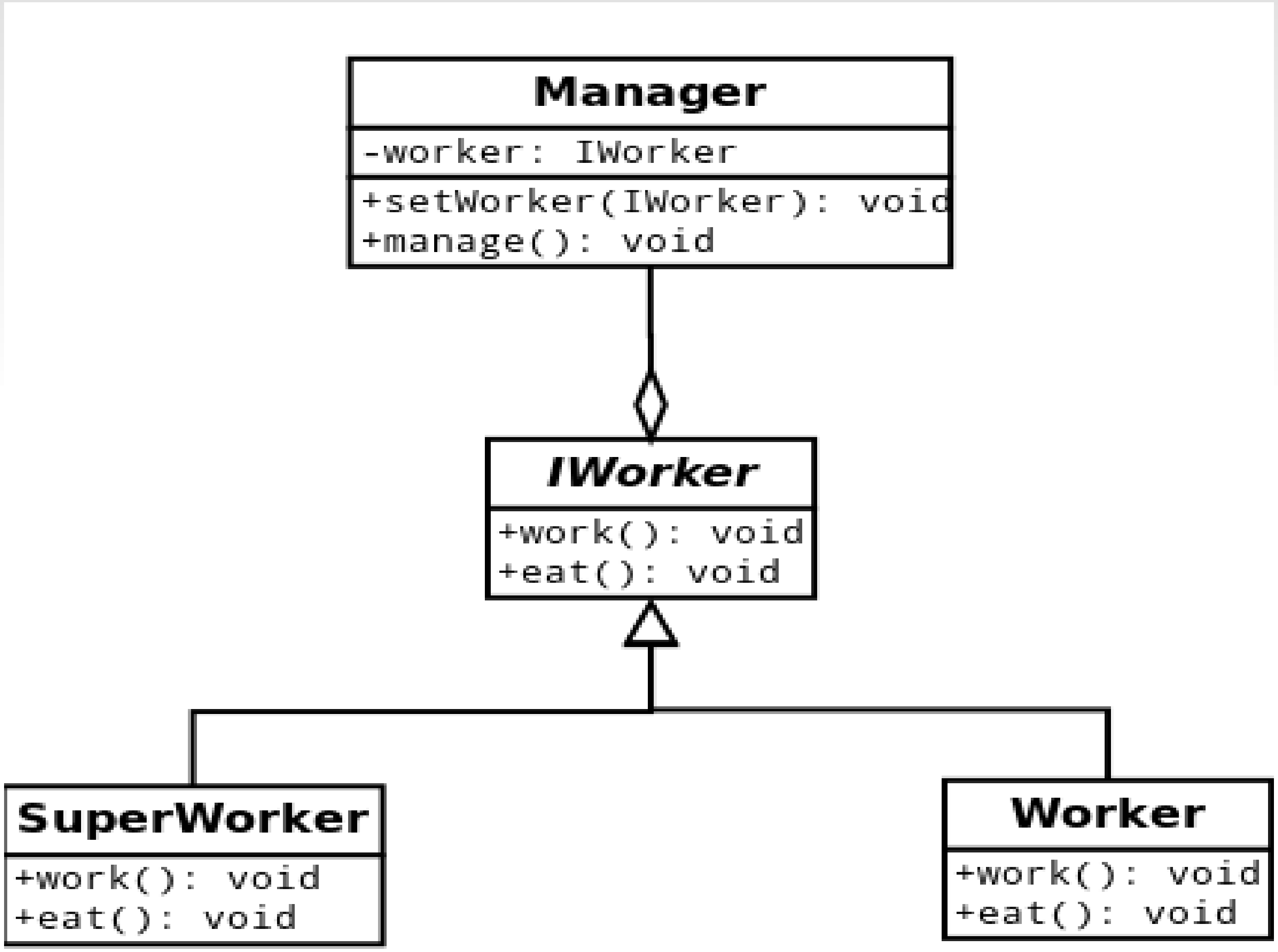
Interface Segregation Principle

- The Interface Segregation Principle (ISP) is a software development principle used for clean development and is intended to help developers avoid making their software impossible to change.
- If followed, the ISP will help a system stay decouple and thus easier to refactor, change and redeploy.

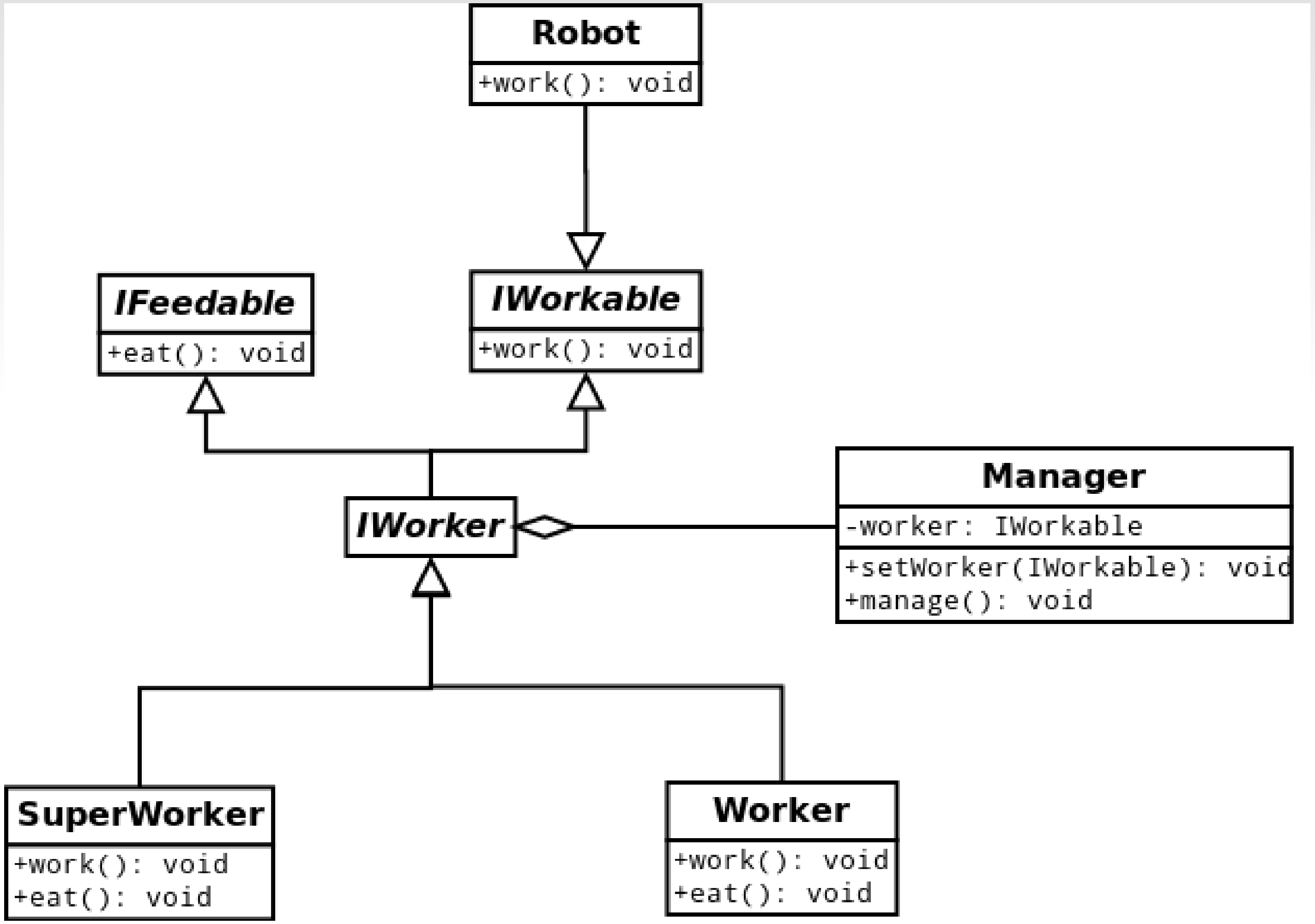
Interface Segregation Principle, *Cont'd*

- The ISP says that once an interface has become too 'fat' it needs to be split into smaller and more specific interfaces so that any client of the interface will only know about the methods that are relevant to them.

Interface Segregation Principle, *Cont'd*



Interface Segregation Principle, *Cont'd*



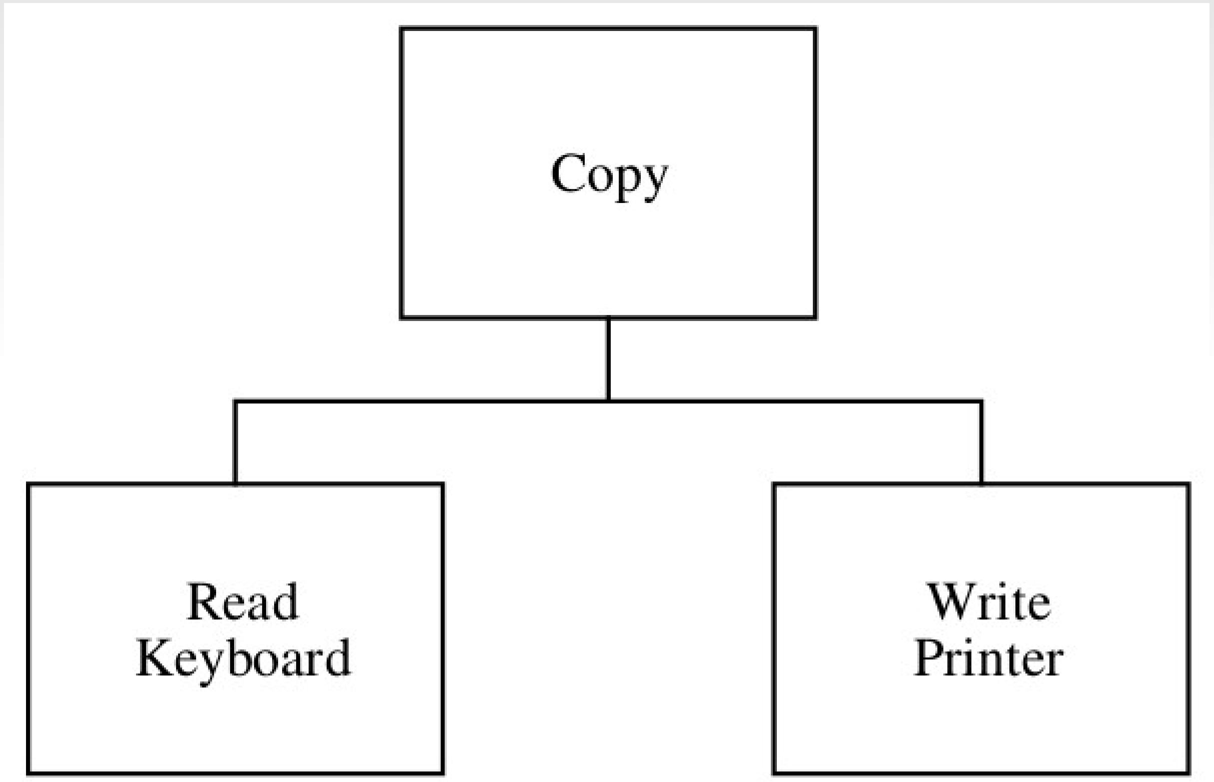
Dependency Inversion Principle

- In OOP, the Dependency Inversion Principle (DIP) states:
    - High-level modules should not depend on low-level modules. Both should depend on abstractions.
    - Abstractions should not depend upon details. Details should depend upon abstractions.

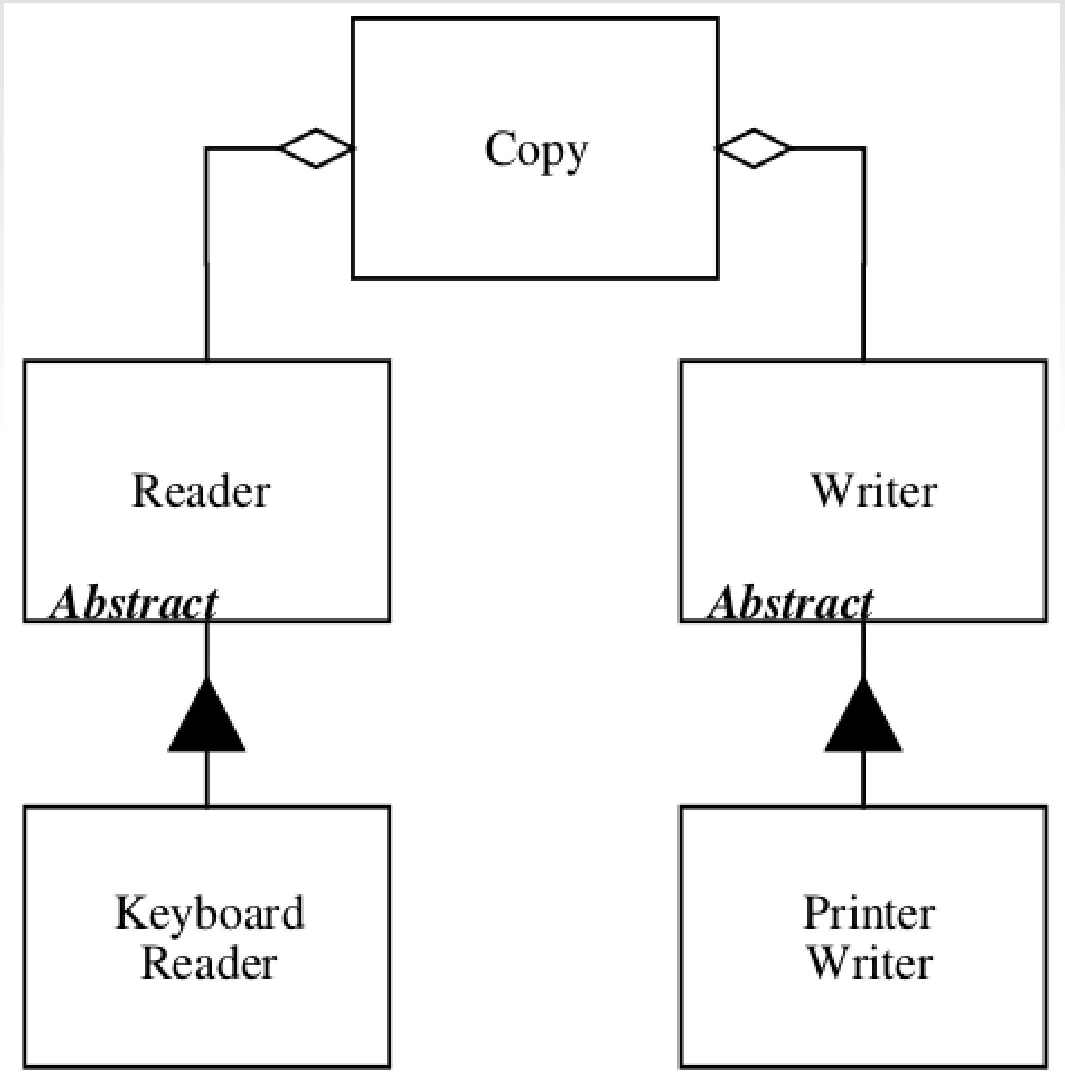
Dependency Inversion Principle, *Cont'd*

- The goal of DIP is to decouple high-level components from low-level components such that reuse with different low-level component implementations becomes possible.
- Applying inversion principle can also be seen as applying the Adapter Pattern.
  - i.e. the high-level class defines its own adapter interface which is the abstraction that the high-level class depends on.

Dependency Inversion Principle, *Cont'd*



Dependency Inversion Principle, *Cont'd*



Finally

- By now, almost all of you should know something about patterns and OO principles.
- You should learn how to apply them, otherwise they are useless.
- And you can learn how to apply them only after practising a lot!