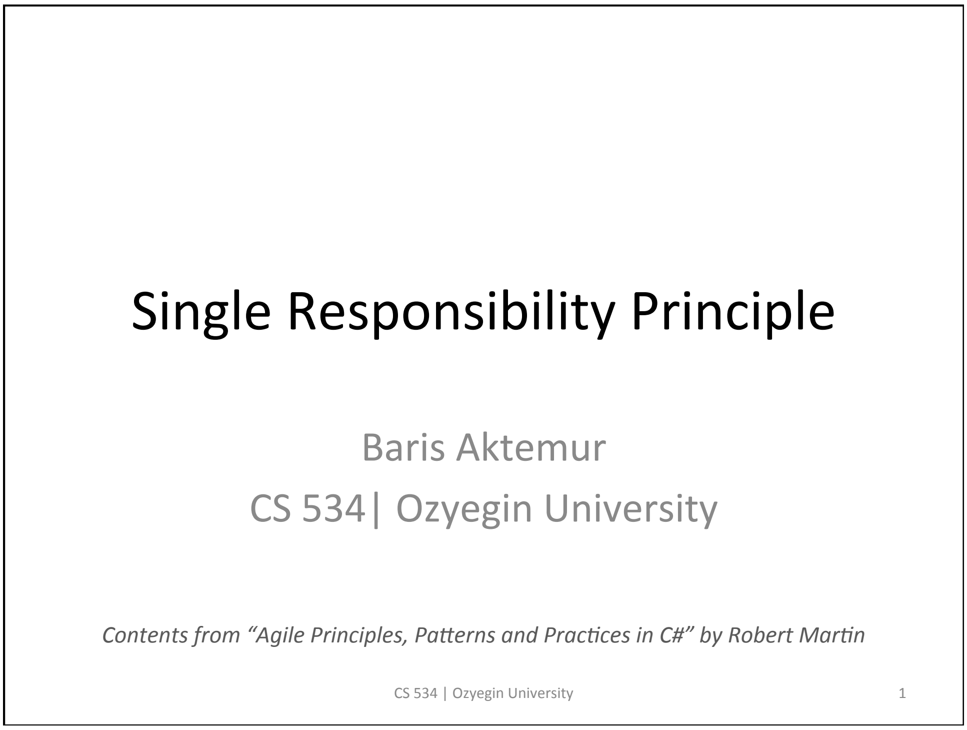
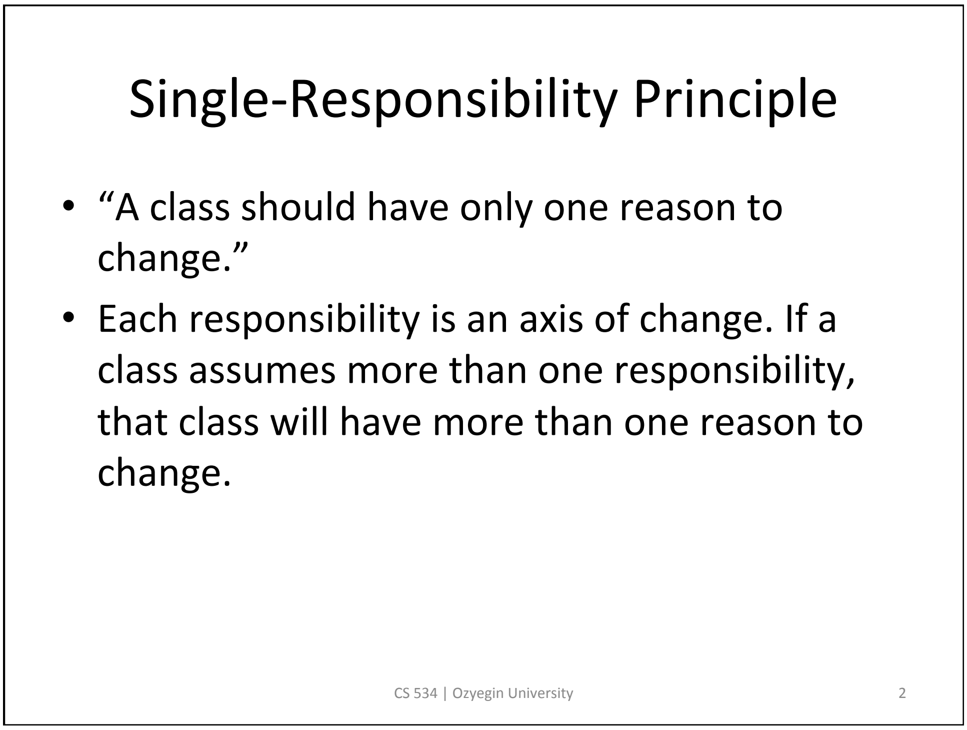


Single Responsibility Principle

Baris Aktemur

CS 534| Ozyegin University

Contents from "Agile Principles, Patterns and Practices in C#" by Robert Martin



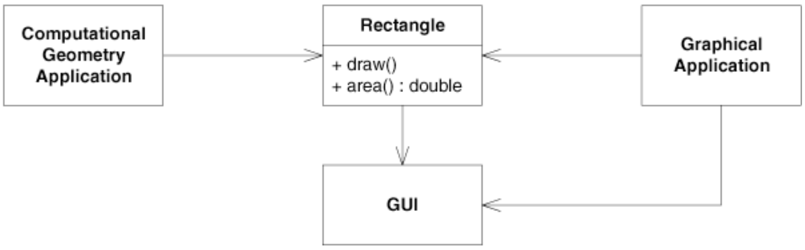
Single-Responsibility Principle

- "A class should have only one reason to change."
- Each responsibility is an axis of change. If a class assumes more than one responsibility, that class will have more than one reason to change.

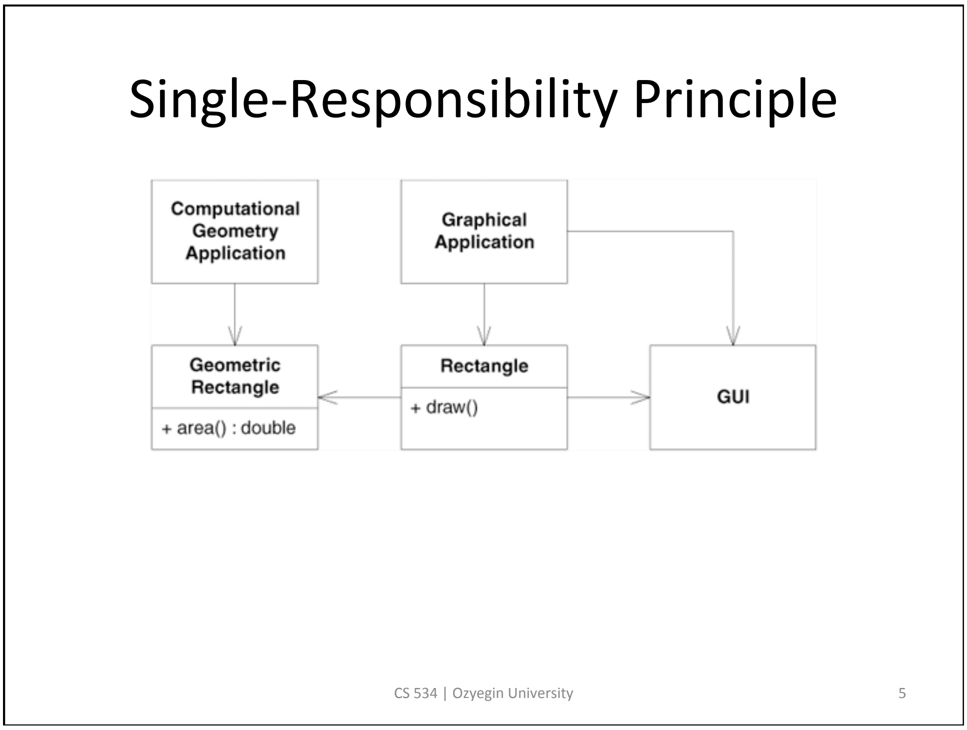
Single-Responsibility Principle

- If a class has more than one responsibility, the responsibilities become coupled.
- Changes to one responsibility may impair or inhibit the class's ability to meet the others.
- This kind of coupling leads to fragile designs that break in unexpected ways when changed.

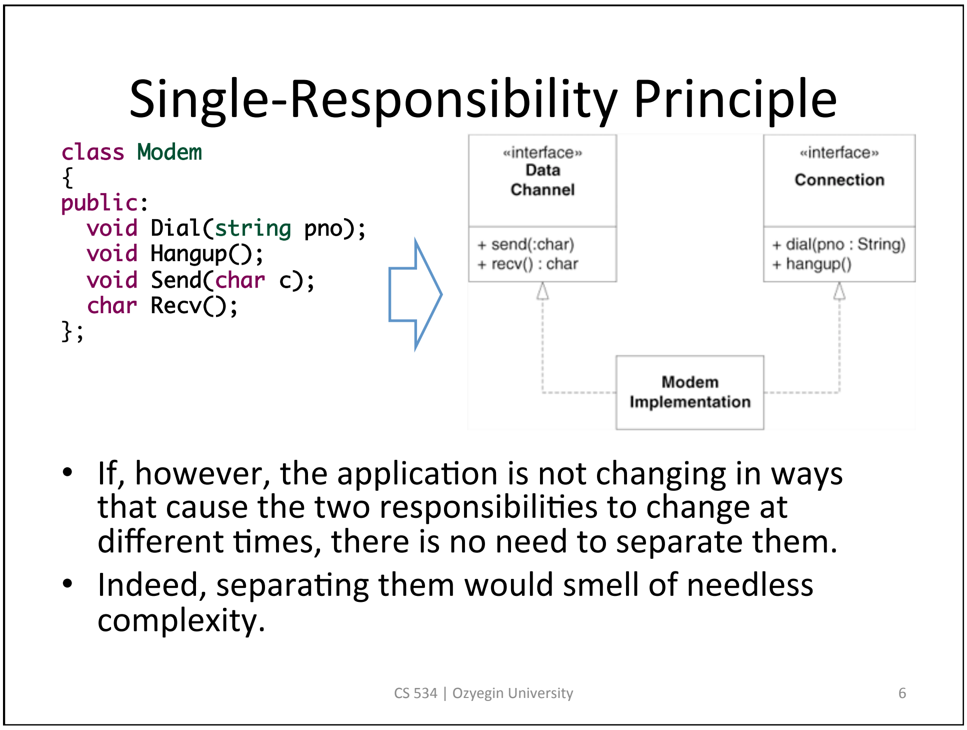
Single-Responsibility Principle



- The Rectangle class has two responsibilities.
- If a change to the GraphicalApplication causes the Rectangle to change for some reason, that change may force us to rebuild, retest, and redeploy the ComputationalGeometryApplication.



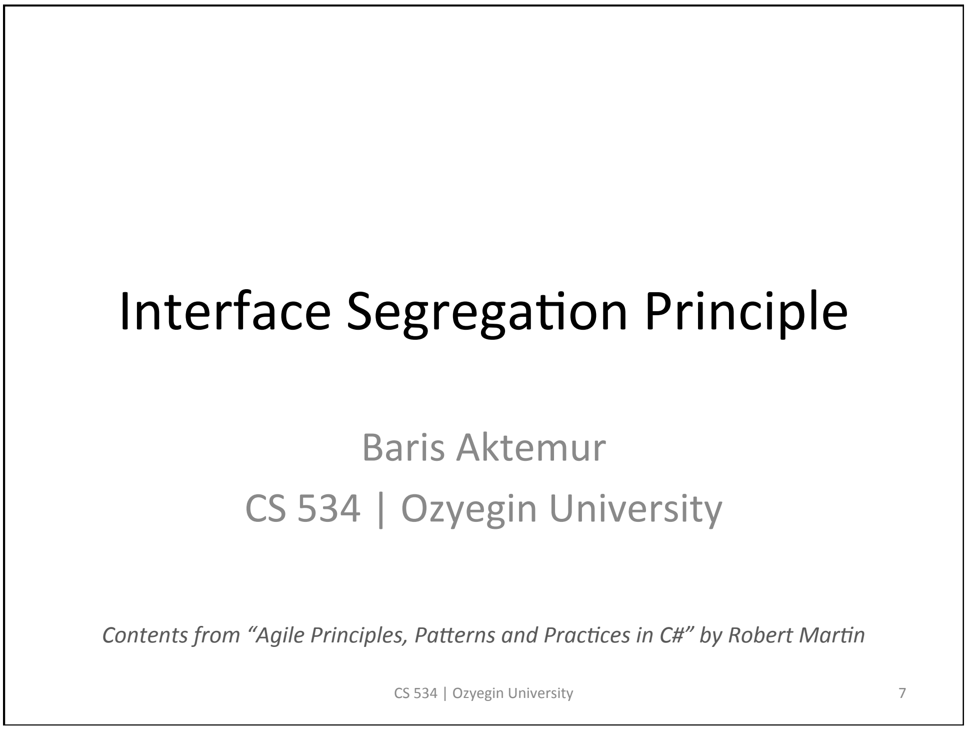
Single-Responsibility Principle



Single-Responsibility Principle

```
class Modem
{
public:
  void Dial(string pno);
  void Hangup();
  void Send(char c);
  char Recv();
};
```

- If, however, the application is not changing in ways that cause the two responsibilities to change at different times, there is no need to separate them.
- Indeed, separating them would smell of needless complexity.

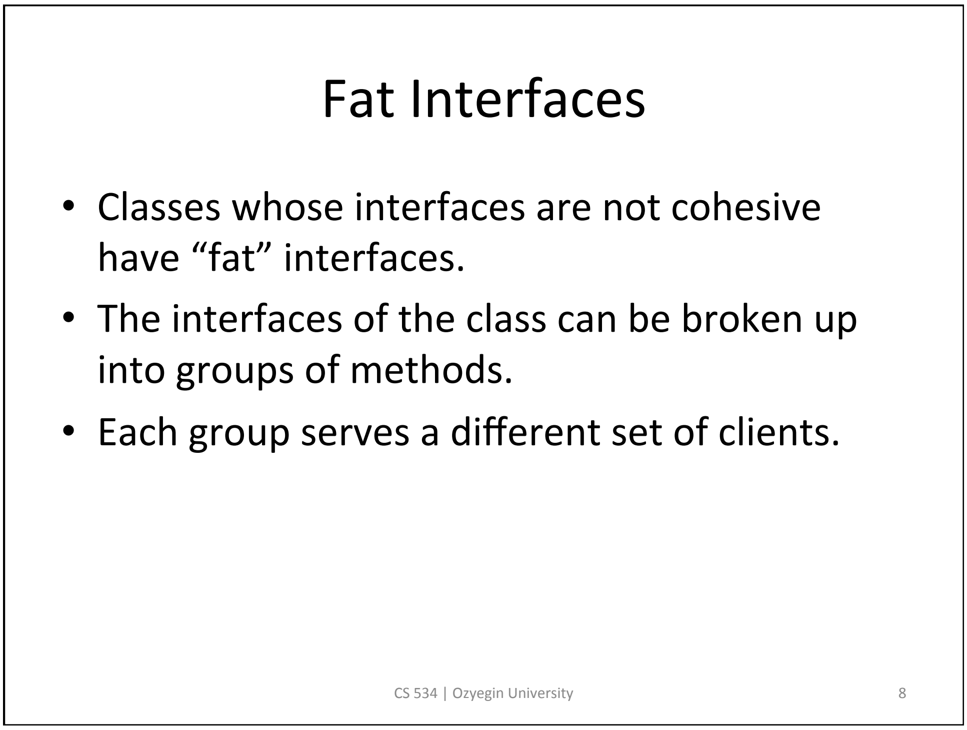


Interface Segregation Principle

Baris Aktemur

CS 534 | Ozyegin University

Contents from "Agile Principles, Patterns and Practices in C#" by Robert Martin

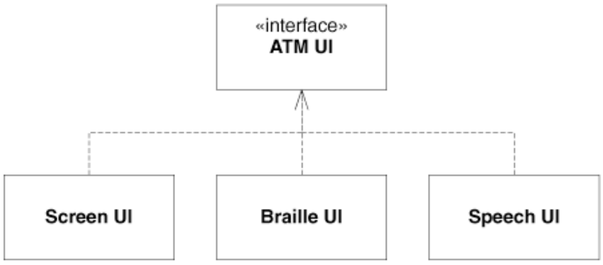


Fat Interfaces

- Classes whose interfaces are not cohesive have "fat" interfaces.
- The interfaces of the class can be broken up into groups of methods.
- Each group serves a different set of clients.

Interface Pollution

- Clients should not be forced to depend on methods they do not use.
- Case: ATM with multiple interfaces

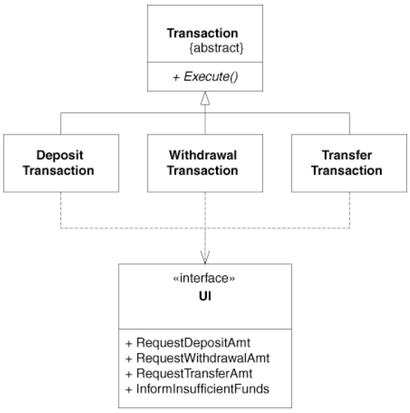


«interface»
ATM UI

Screen UI

Braille UI

Speech UI



Transaction
{abstract}
+ Execute()

Deposit Transaction

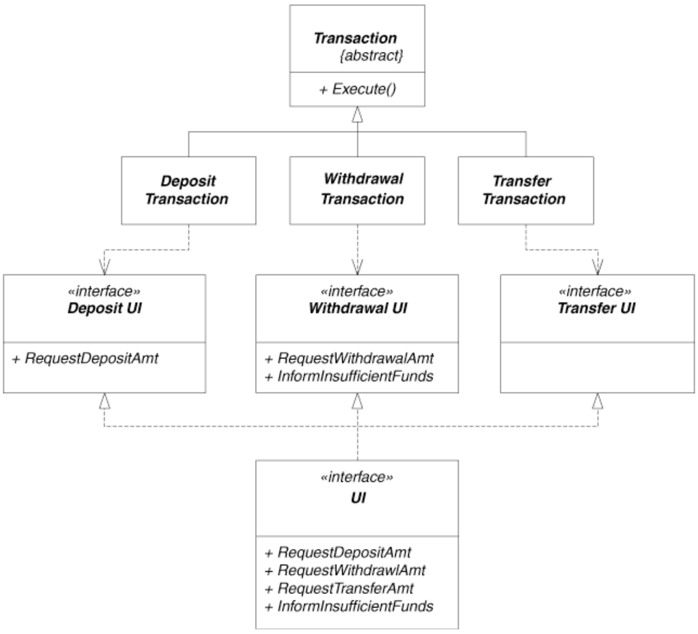
Withdrawal Transaction

Transfer Transaction

«interface»
UI

+ RequestDepositAmt
+ RequestWithdrawalAmt
+ RequestTransferAmt
+ InformInsufficientFunds

- Adding a new transaction changes the UI interface
- All the other Transactions have to be recompiled



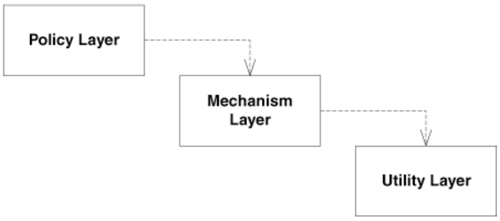
Dependency-Inversion Principle

Baris Aktemur

CS 534 | Ozyegin University

Contents from “Agile Principles, Patterns and Practices in C#” by Robert Martin

The Dependency-Inversion Principle

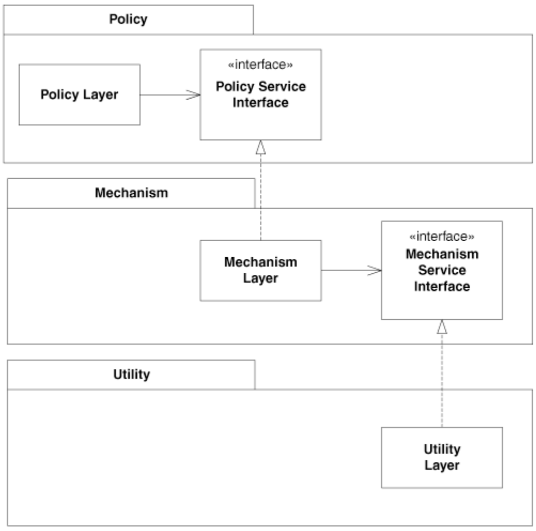


Naive layering scheme

- 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.

The Dependency-Inversion Principle

- It is the high-level, policy-setting modules that ought to be influencing the low-level detailed modules.
- The modules that contain the high-level business rules should take precedence over, and be independent of, the modules that contain the implementation details.
- It is high-level, policy-setting modules that we want to be able to reuse.
  - We are already quite good at reusing low-level modules in the form of subroutine libraries.
  - When high-level modules depend on low-level modules, it becomes very difficult to reuse those high-level modules in different contexts.



Policy

Policy Layer

«interface»
Policy Service Interface

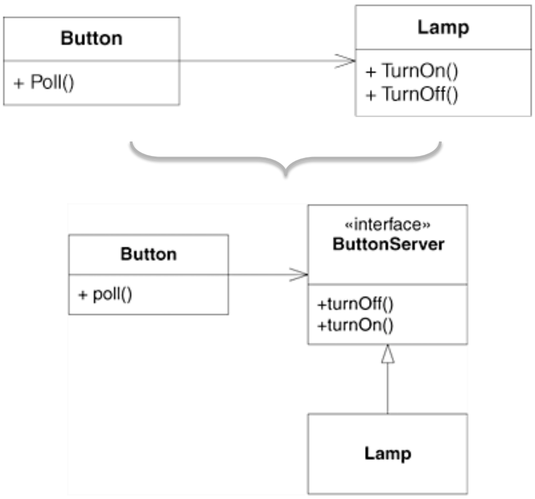
Mechanism

Mechanism Layer

«interface»
Mechanism Service Interface

Utility

Utility Layer



Button

+ Poll()

Lamp

+ TurnOn()
+ TurnOff()

Button

+ poll()

«interface»
ButtonServer

+turnOff()
+turnOn()

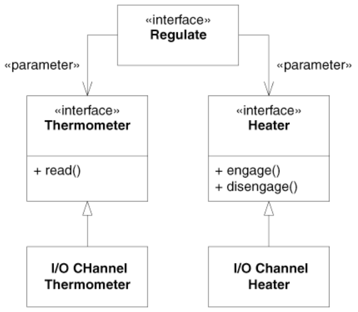
Lamp

Thermostat Algorithm

```
const byte TERMOMETER = 0x86;
const byte FURNACE = 0x87;
const byte ENGAGE = 1;
const byte DISENGAGE = 0;

void Regulate(double minTemp, double maxTemp) {
    for(;;) {
        while (in(THERMOMETER) > minTemp)
            wait(1);
        out(FURNACE,ENGAGE);

        while (in(THERMOMETER) < maxTemp)
            wait(1);
        out(FURNACE,DISENGAGE);
    }
}
```

Thermostat Algorithm

```
void Regulate(Thermometer t, Heater h,
              double minTemp, double maxTemp)
{
    for(;;) {
        while (t.Read() > minTemp)
            wait(1);
        h.Engage();

        while (t.Read() < maxTemp)
            wait(1);
        h.Disengage();
    }
}
```



Now the algorithm is nicely reusable.