Design Principles: SOLID

Topics

Name your favourite design principle which:

1) Limits for whom we change a file.
2) Adds new code for changes.
3) Makes substitutable objects.
4) Prevents depending on things you don’t need.
5) Prevents high-level policies from depending on low-level details.

Design Principles

- Design principles help us design software which is:
  - more understandable
  - ..
  - more maintainable
- We have seen
  - Separate aspects that change
    from those that stay the same
  - Classes should be
    open for extension, but closed for modification
  - Program to an interface, not an implementation
  - Favour composition over inheritance

SOLID

- SRP: Single Responsibility Principle
  - Each part of the system must have only one *reason* to change.
- OCP: Open-Closed Principle
  - For a software system to be easy to change, those changes must be done through adding new code, not changing existing code.
- LSP: Liskov Substitution Principle
  - To build a software system from interchangeable parts, the parts must adhere to a contract which allows the parts to be interchangeable.
- ISP: Interface Segregation Principle
  - Don't depend on things you don't use.
- DIP: Dependency Inversion Principle
  - Code that implements high-level policy should not depend on code that implements low-level details.

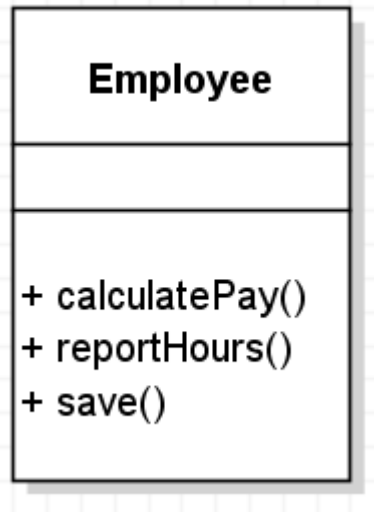
SRP

Single Responsibility Principle (SRP)

- ..
  - Actor: A group of stakeholders
- Idea
  - The contents of a module are there to satisfy the needs of one group.

SRP Violation: Multiple Actors

- Actors' Needs
  - calculatePay(): specified by accounting
  - reportHours(): specified by human resources
  - save(): specified by IT administrators

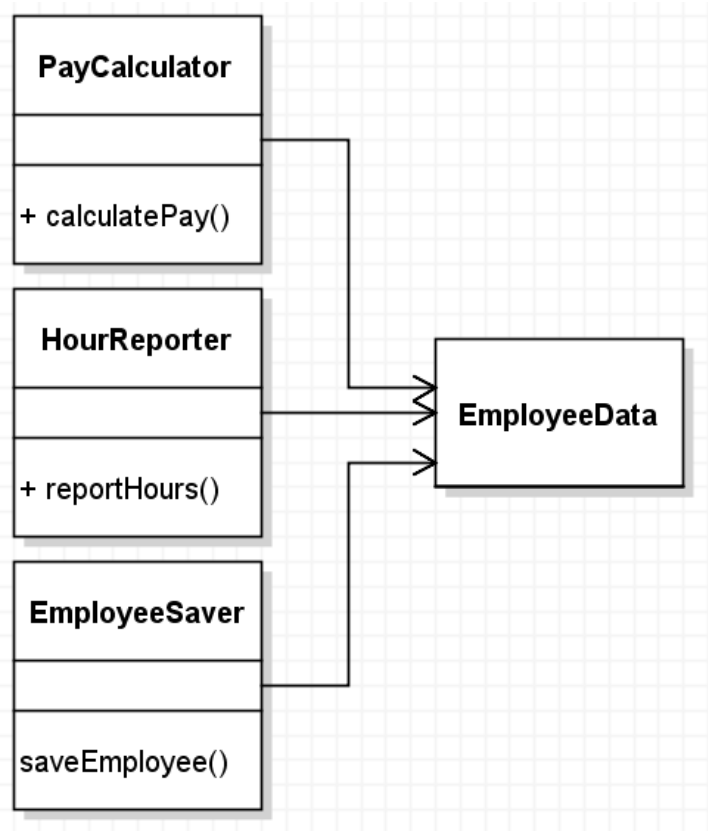


- Design couples Employee to three different actors
  - Imagine a regularHours() function used by calculatePay() and reportHours()
  - If *accountants* ask for a change to regularHours(), the change unexpectedly impacts reportHours() and *HR*
- SRP: Separate the code so that changes needed by one actor

..

SRP Solution

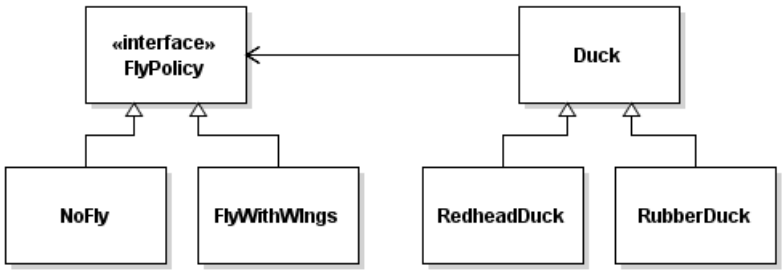
- Move each actor's needs to own class
  - Store data in its own EmployeeData class
  - Create three independent classes to process the data
- No part of the processing code is beholden to multiple stakeholders.

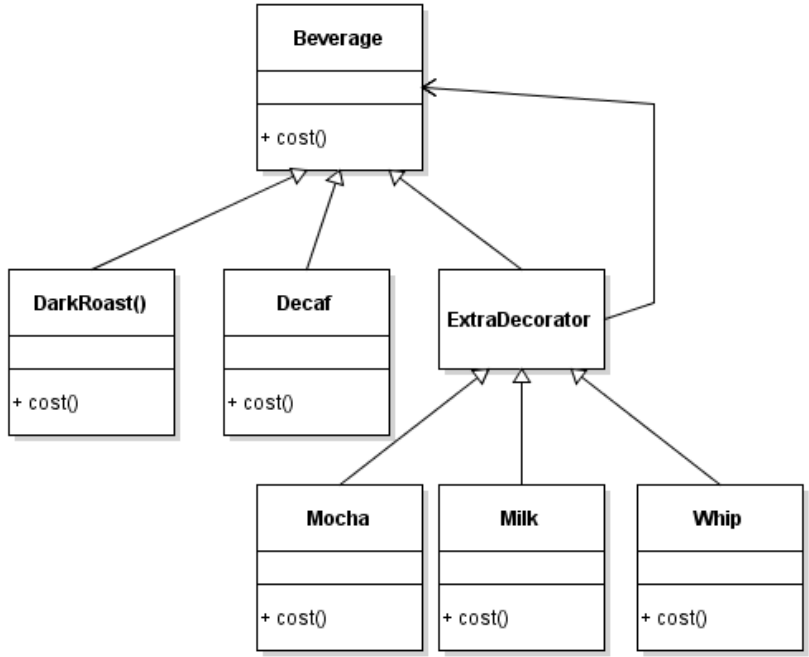


OCP

Open Closed Principle (OCP)

- "A software artifact should be open for extension but closed for modification."





(Bertrand Meyer. Object Oriented Software Construction, Prentice Hall, 1988, p23)

OCP and Checked Exceptions

High-Level

```
void printResult() {
    int value = sumValues();
    display(value);
}
```

Logic Layer

```
int sumValues() {
    return dbGetValue("Sales")
           + dbGetValue("Service");
}
```

DB Layer

```
int dbGetValue(String record) {
    return ...;
}
```

What happens when the DB Layer throws a checked exception?

OCP and Checked Exceptions

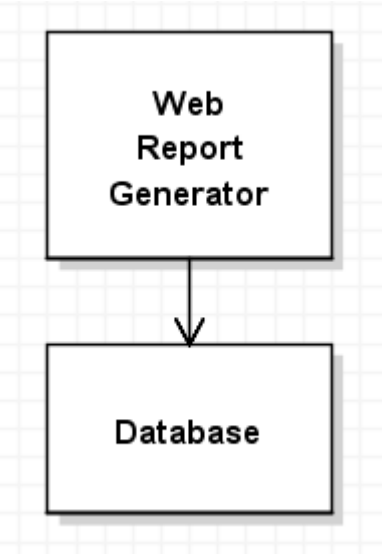
```
void printResult() {
    int value = 0;
    try {
        value = sumValues();
    } catch (DbException e) {
        display("ERROR");
    }
    display(value);
}

int sumValues()                         throws DbException
{
    return dbGetValue("Sales")
           + dbGetValue("Service");
}

int dbGetValue(String record)           throws DbException
{
    return ...;
}
```

OCP and Checked Exceptions

- Using Checked Exceptions Violates OCP
  - A low-level change to one module
    ..
- Solutions
  - Throw unchecked exceptions
  - Wrap exceptions inside custom (checked) MyDatabaseException:
    Changes to exceptions thrown are wrapped inside custom exception.

OCP & Architecture

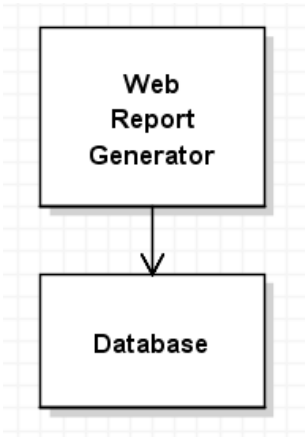
- Design is a spectacular failure if
  ..
- Ex: Generating a business report
  - already implemented for the web
    (scrollable, negative numbers in red)
  - now adding B&W print
    (pagination, negative numbers in brackets)
- To add the print report, what needs to:
  - be changed?
  - be added?



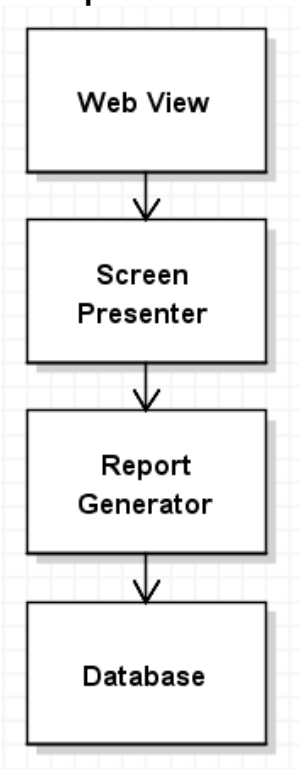
OCP & Architecture

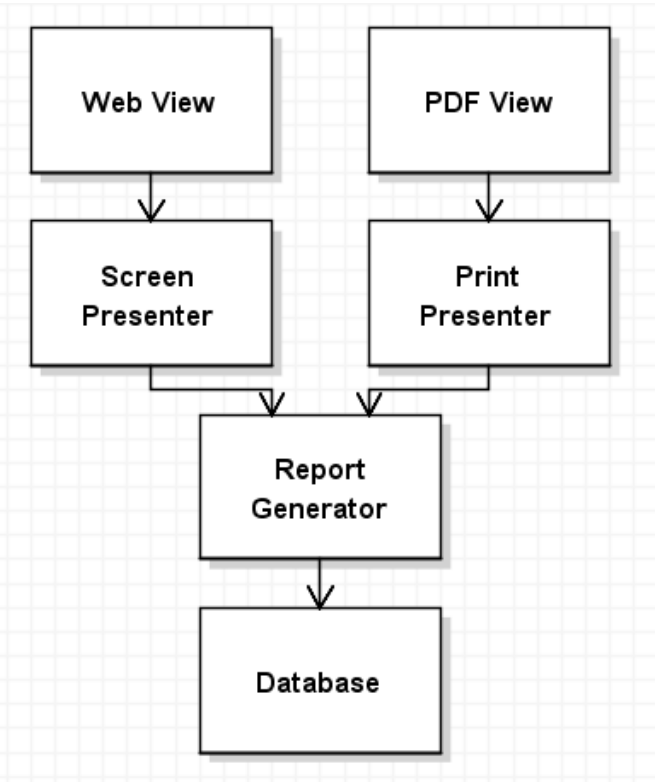
- To add the print report, what needs to
  be changed? **be added**?

One class



Multiple classes



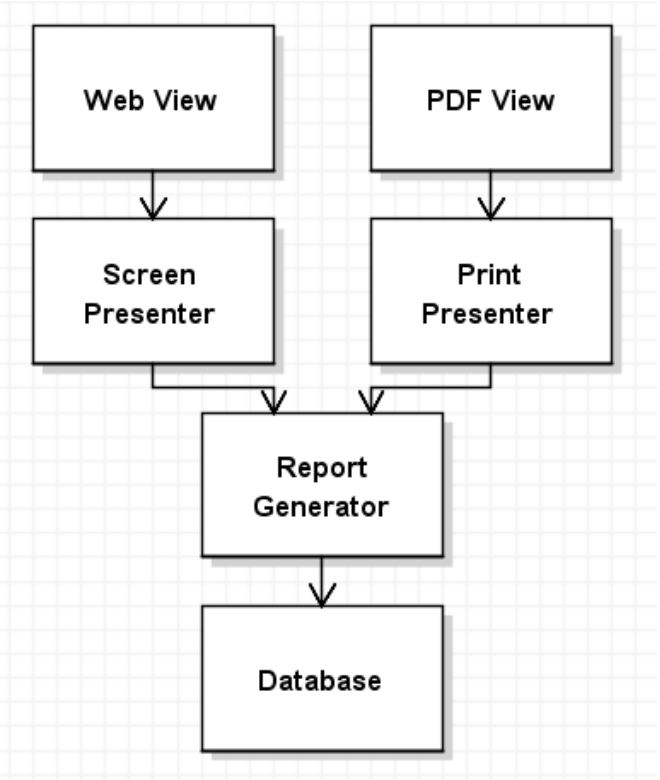


(Clean Architecture, Robert C. Martin, 2018)

OCP & Architecture (cont)

- OCP is about protection from change
  - If component A should be protected from changes in component B, then..

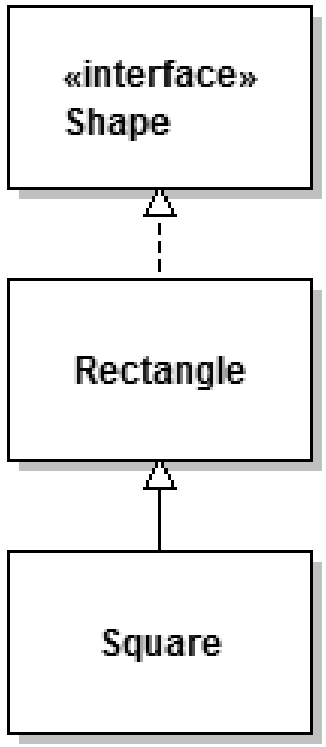
- Ex: ReportGenerator is protected from ScreenPresenter
  - What does ReportGenerator depend on?
  - It's the business rules;
    it's least likely to change;
    it’s most likely to be reused.
- We'll see later how to use the dependency inversion principle.



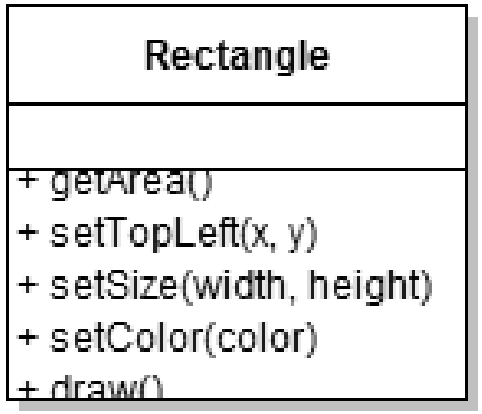
LSP

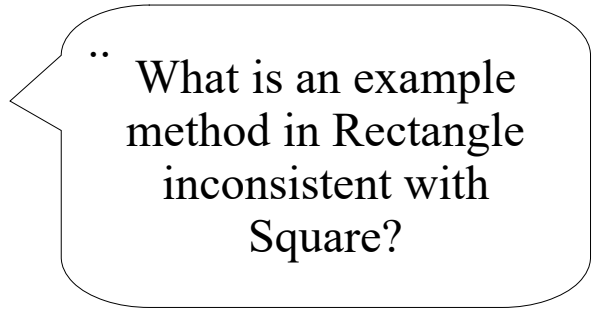
Inheritance

- Idea: Represents a..
- Example:



  - Square is-a Rectangle, and gives reuse.
  - But..





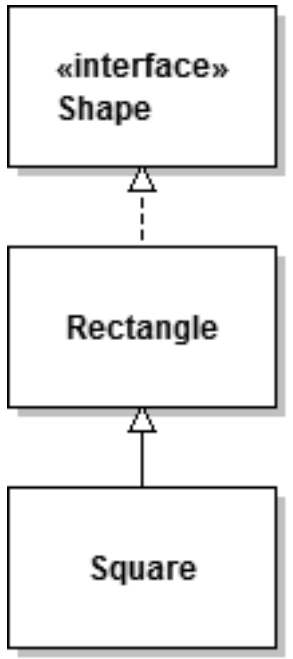
  - How can we describe this problem?

Inheritance: LSP

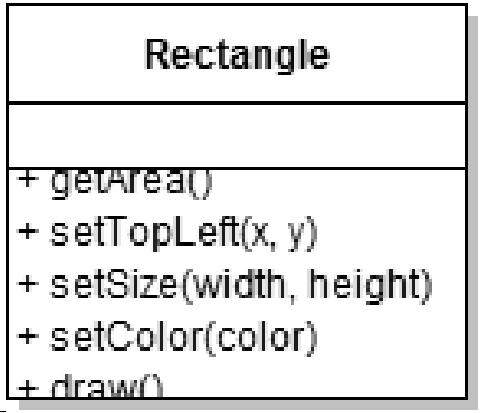
- Liskov Substitution Principle (LSP)
  B can inherit from A only if..

    1)..
    that A's method accepts (or more) and

    2)..
    that A's method does (or more).



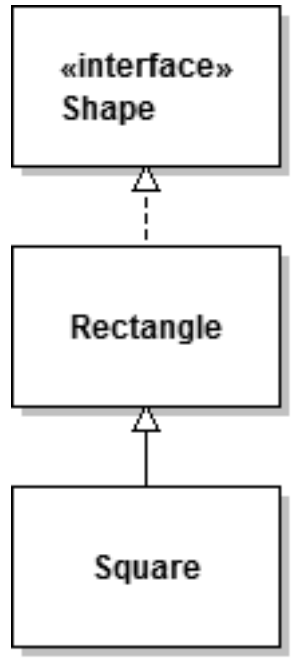
- What methods in Rectangle fail LSP for Square?
  -



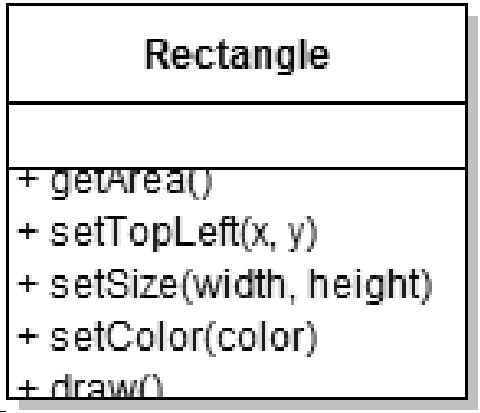
  - Square does not do the same things with all values as Rectangle: fails LSP.

Is-A: LSP & Immutable

- LSP & Immutable
  - Would making Rectangle and Square immutable help?
  -

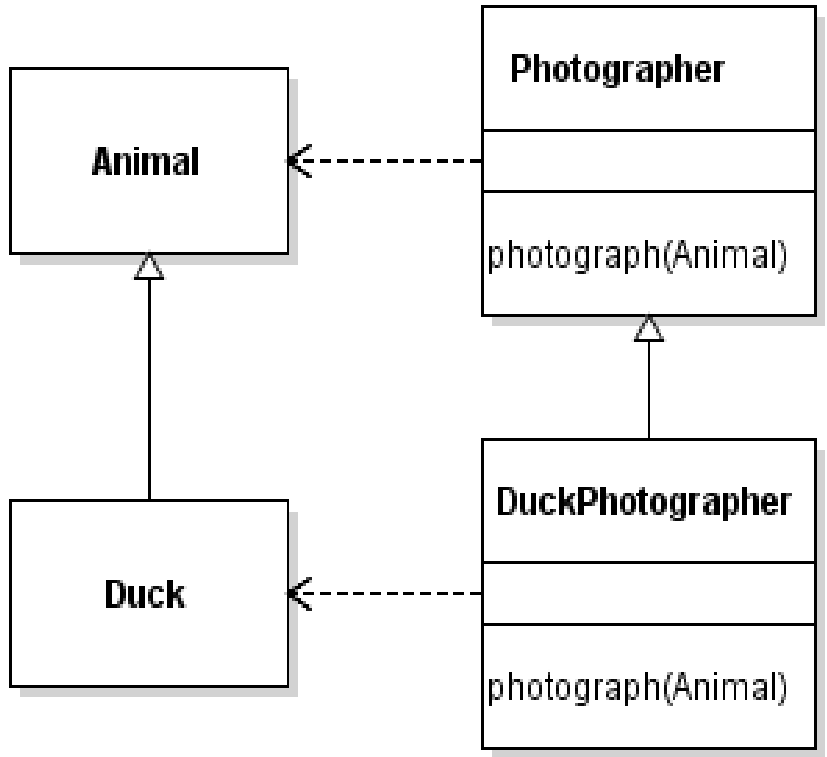


- Inheritance must satisfy the SLP
  so all derived objects are interchangable.



Is-A LSP: Example

- Photographer can photograph any Animal.
  DuckPhotographer only wants to photograph Ducks.
- DuckPhotographer.photograph()
  wants to reject non-ducks
  - Could throw an
    IllegalArgumentException?
- DuckPhotographer
  ..
  - ..



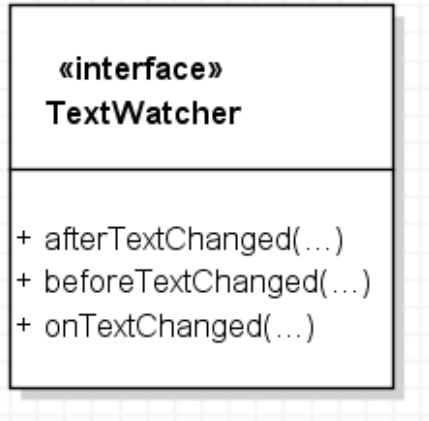
Is-A LSP

- Rephrase LSP:
  - Client code using a reference to the base class must be able to..
  - i.e., behaviour is unchanged.

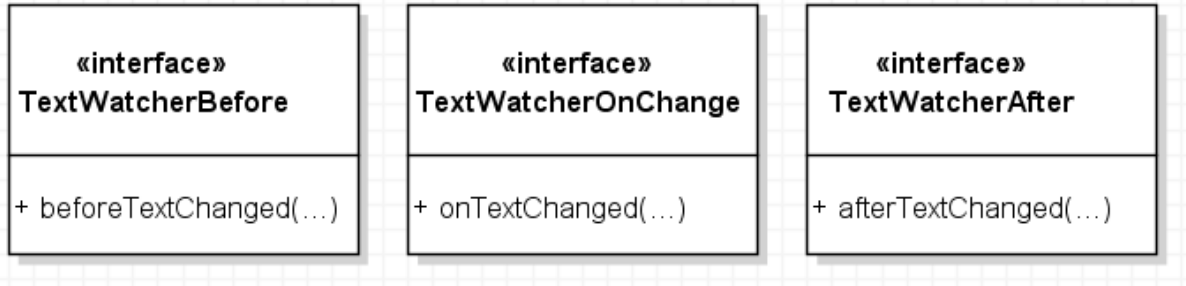
ISP

Interface Segregation Principle (ISP)

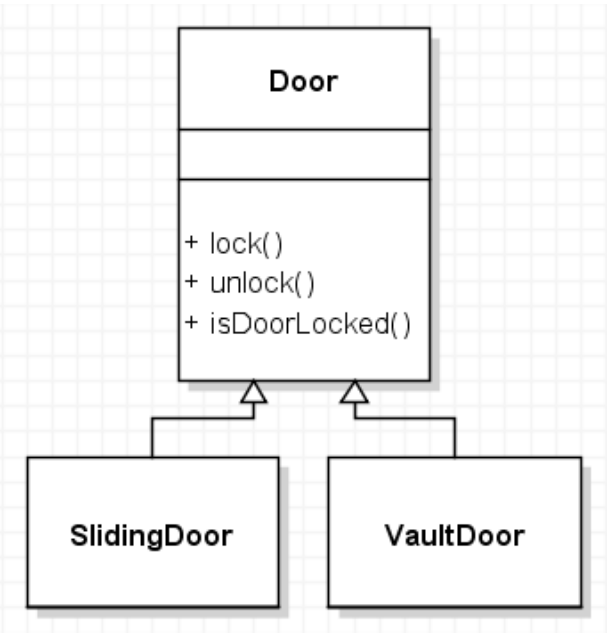
- Clients should not be forced to..



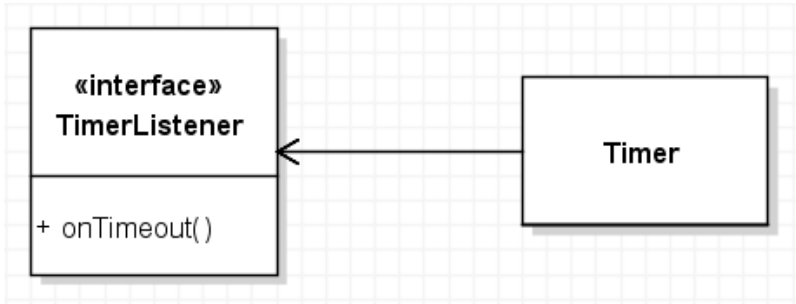
Decompose into interface for each client



ISP Door Example



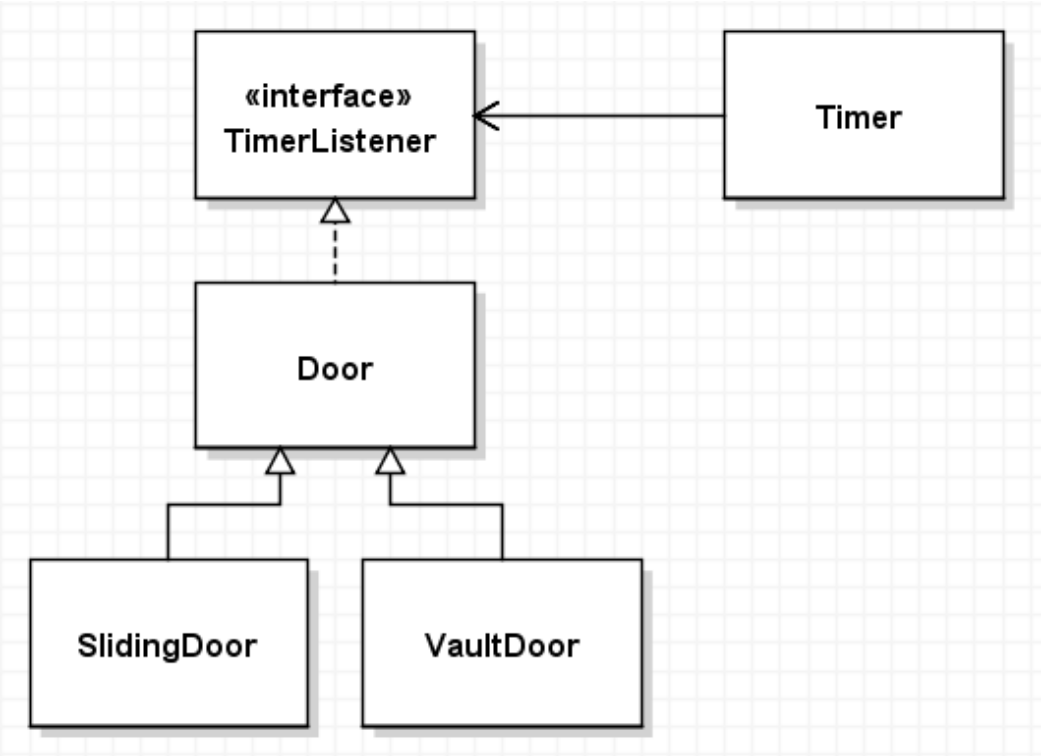
- Add a timed alarm to door using a timer & listener



- We need VaultDoor to be a TimerListener.

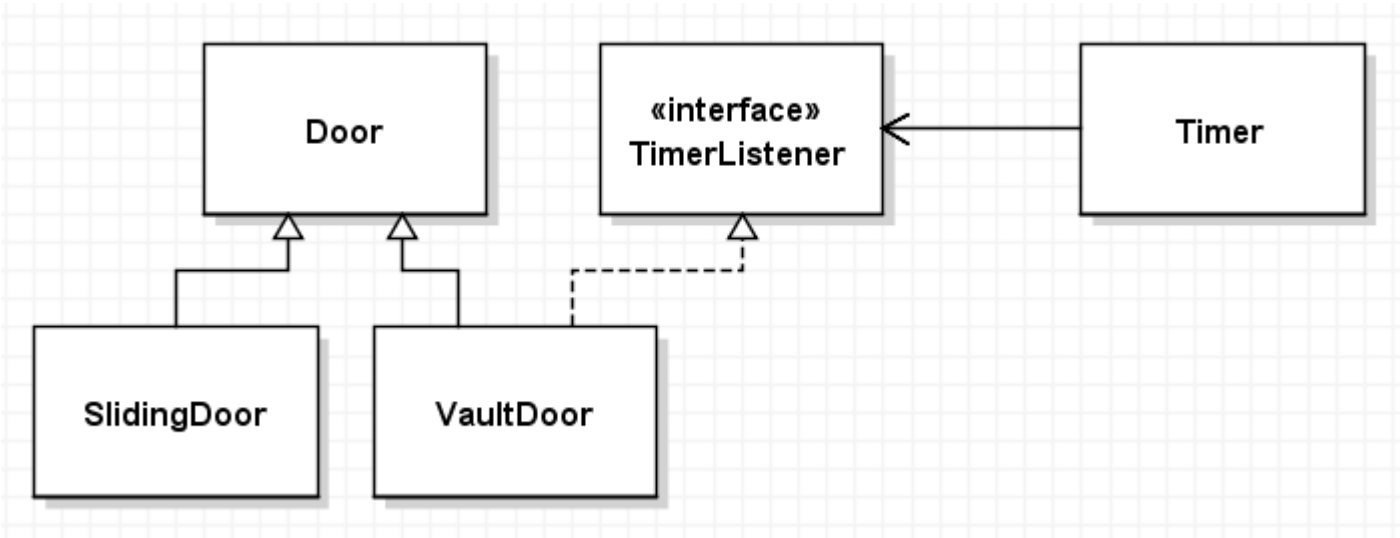
ISP Door Example

- What is wrong with making Door a TimerListener so that VaultDoor can register with the Timer?
  - ..
  - Derived classes need to implement TimerListener
  - Change to TimerListener interface requires changes to all derived classes.



Possible ISP solution

- Don’t force classes to implement interfaces they don’t need.



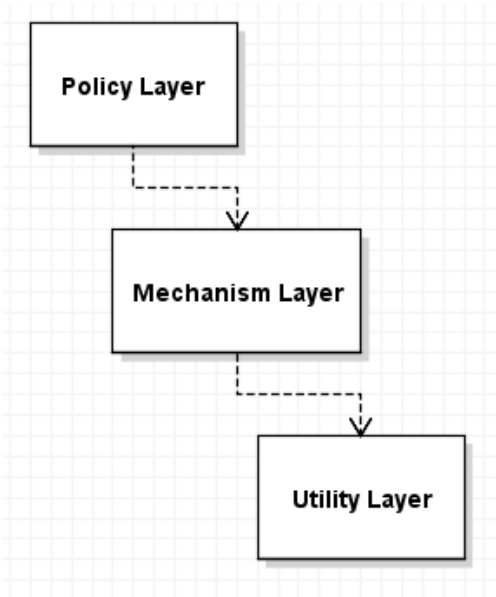
DIP

Dependency Inversion Principle (DIP)

- Flexible code..
  not on concrete implementations.
  - Exception for stable classes like String
  - Apply DIP to *volatile* classes we are actively developing.
    Use interfaces; they are much less volatile

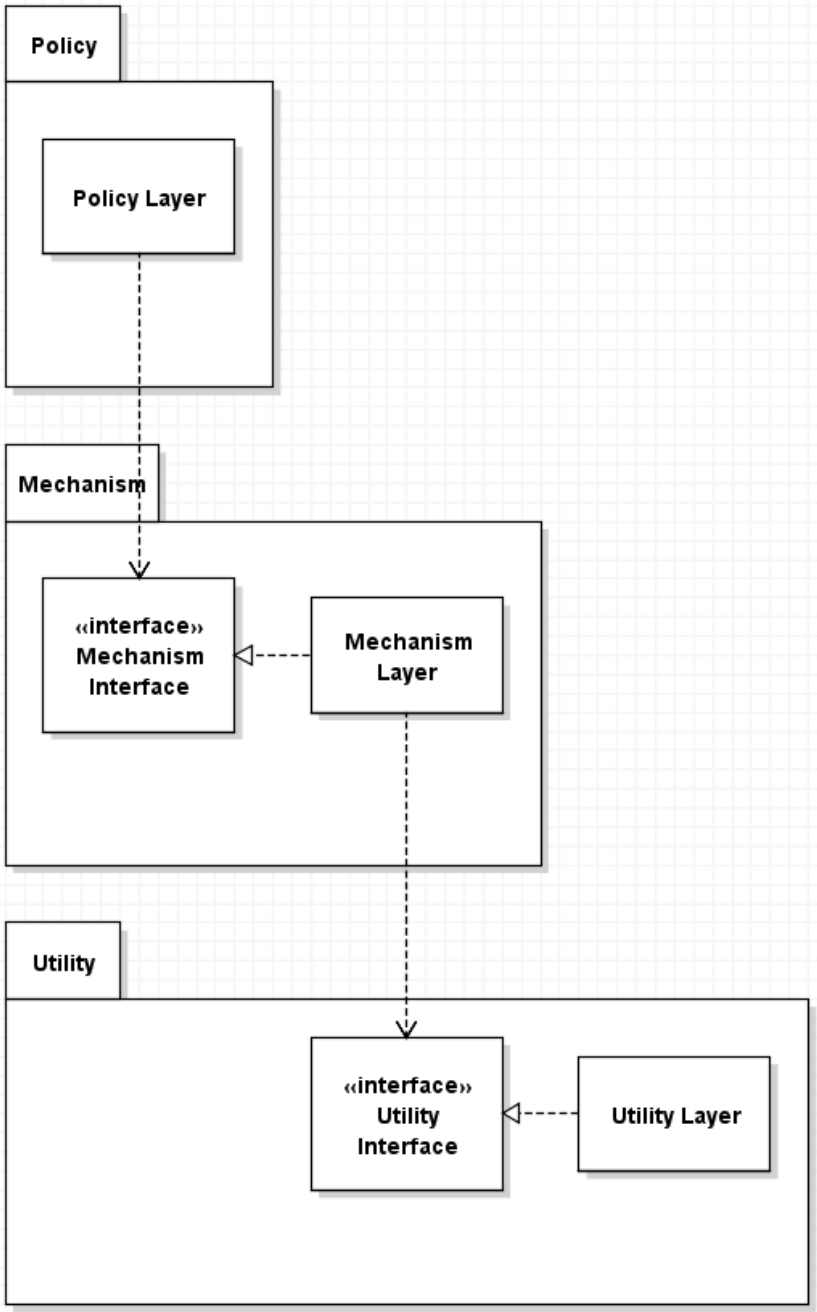
Common Case

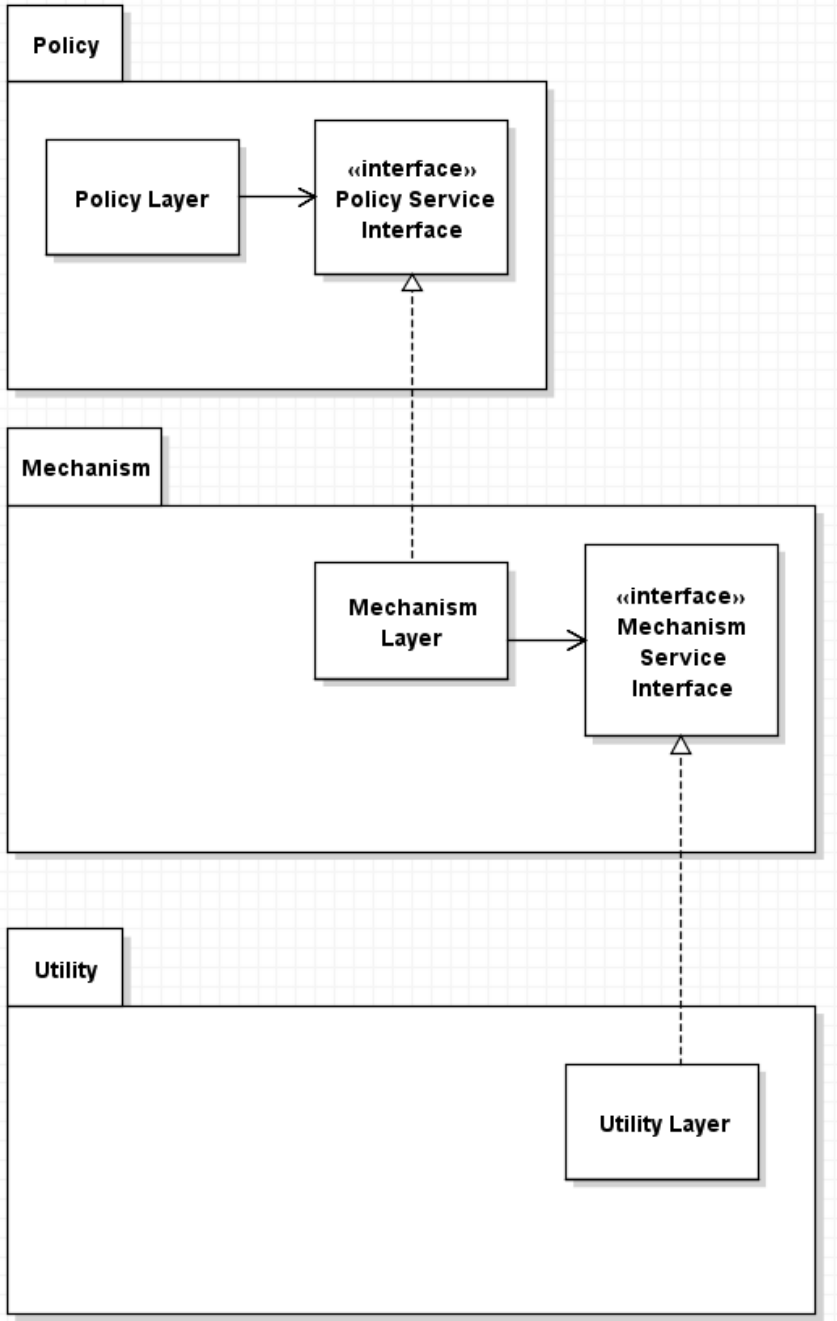
- High-level code
  often depends on low-level implementations



- Scope of Change
  - Changes in the lowest levels (changing a class name / method signature)..
- Can we "invert" this so highest level is isolated from change?
  "Dependency Inversion"

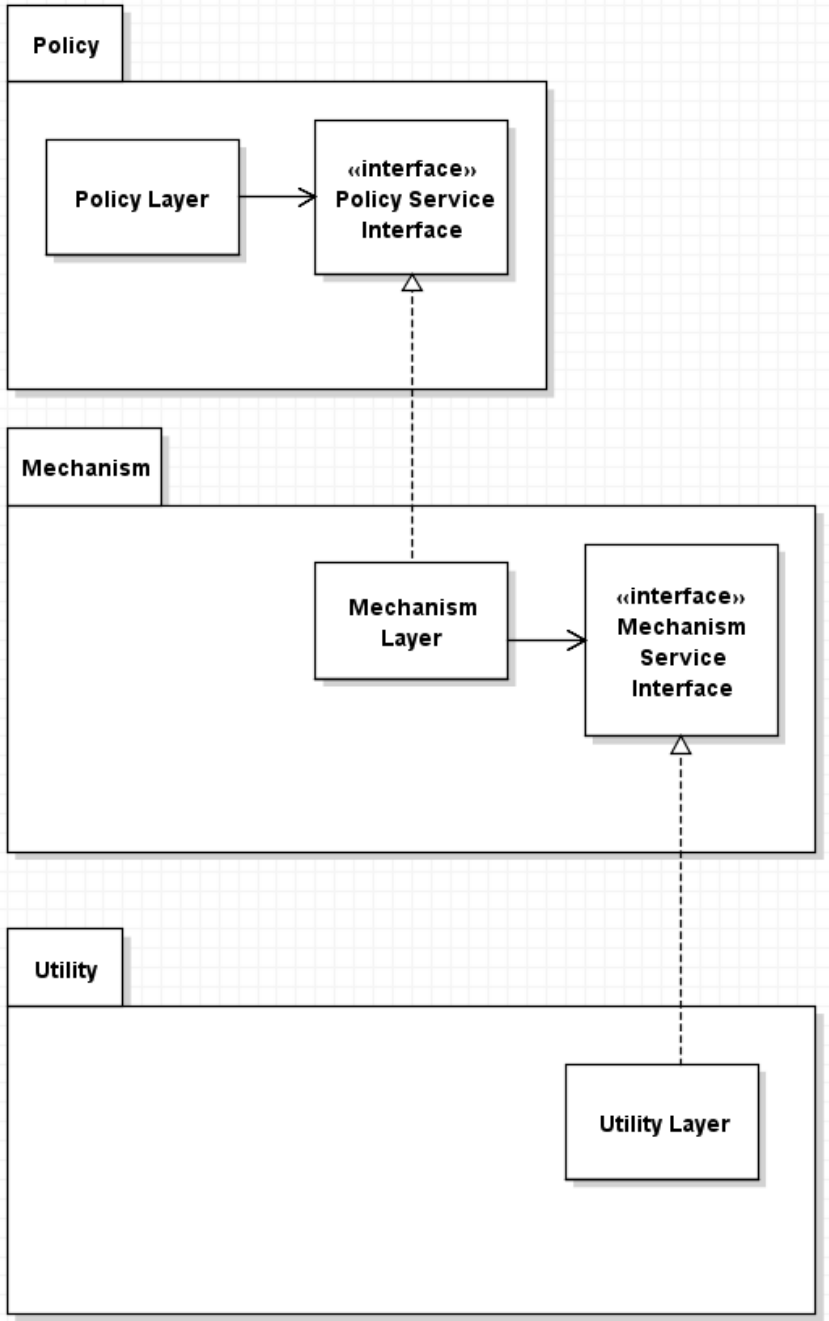
rsion


Policy
Policy Layer
Mechanism
«interface»
Mechanism
Interface
Mechanism
Layer
Utility
«interface»
Utility
Interface
Utility Layer


Policy
Policy Layer
«interface»
Policy Service
Interface
Mechanism
Mechanism
Layer
«interface»
Mechanism
Service
Interface
Utility
Utility Layer

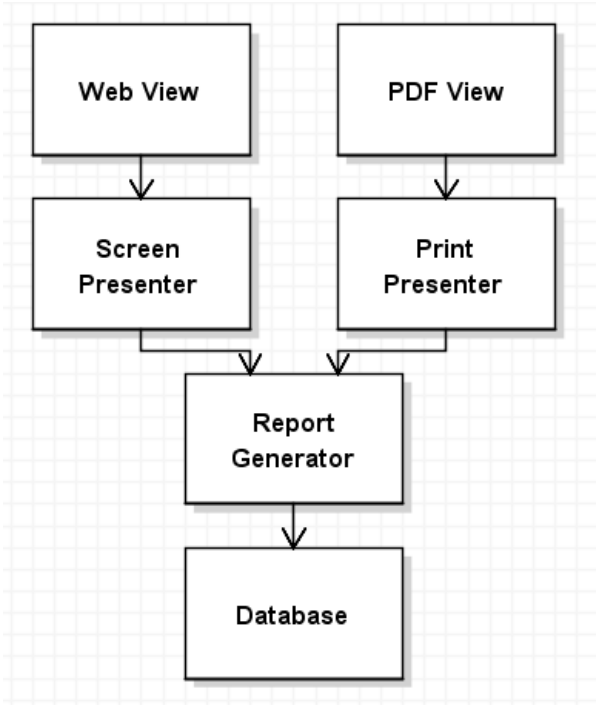
Dependency Inversion

- Idea:
  - ..
- This is dependency inversion: Lower level depends on the interface in higher level
- This is Ownership Inversion: higher level owns the interface.
  - It dictates services it needs, lower levels can implement that to service its needs.
- ..

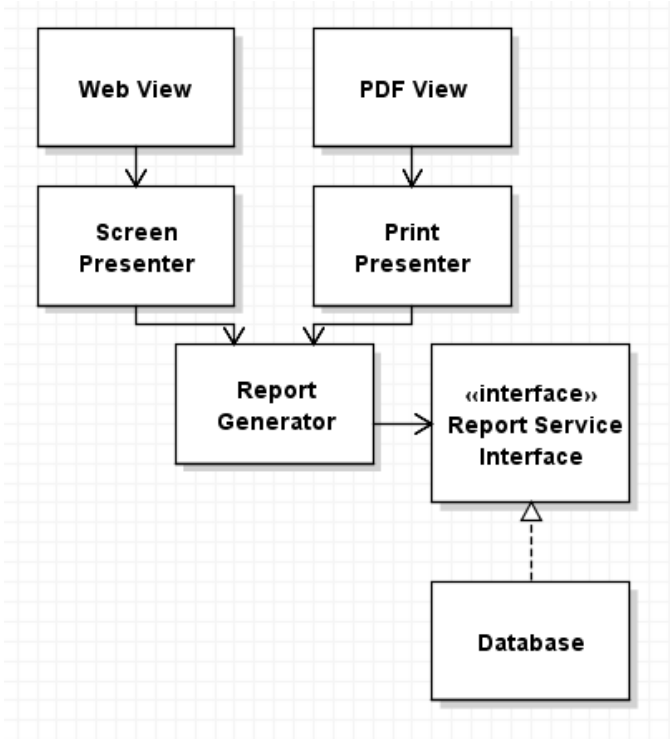


OCP & DIP Revisited

- Before DIP, Report Generator depends on DB



- After DIP: DB depends on Report Generator



SOLID Summary

- SRP: Single Responsibility Principle
  - Each part of the system must have only one *reason* to change.
- OCP: Open-Closed Principle
  - For a software system to be easy to change, those changes must be done through adding new code, not changing existing code.
- LSP: Liskov Substitution Principle
  - To build a software system from interchangeable parts, the parts must adhere to a contract which allows the parts to be interchangeable.
- ISP: Interface Segregation Principle
  - Don't depend on things you don't use.
- DIP: Dependency Inversion Principle
  - Code that implements high-level policy should not depend on code that implements low-level details.