Object-Oriented Design

IF2210 – Pemrograman Berorientasi Objek

Semester II – 2024/2025

by: IL, rev: SAR

Bahan Bacaan:
URL dan juga tautan di dalamnya

- http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod
- https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)
- http://www.ecs.syr.edu/faculty/fawcett/handouts/cse687/Presentations/principles.pdf
- http://www.oodesign.com/design-principles.html
- https://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/

Bad design

- What makes a design bad? Robert Martin suggests:
  - **Rigidity:** It is hard to change because every change affects too many other parts of the system.
  - **Fragility:** When you make a change, unexpected parts of the system break.
  - **Immobility:** It is hard to reuse in another application because it cannot be disentangled from the current application.
- The design principles discussed in the following are all aimed at preventing “bad” design.

S.O.L.I.D. Principles

- Single responsibility principle
- Open closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle

Single Responsibility Principle

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

i.e.

"A class should have only one job."


SINGLE RESPONSIBILITY PRINCIPLE
Just Because You Can, Doesn't Mean You Should

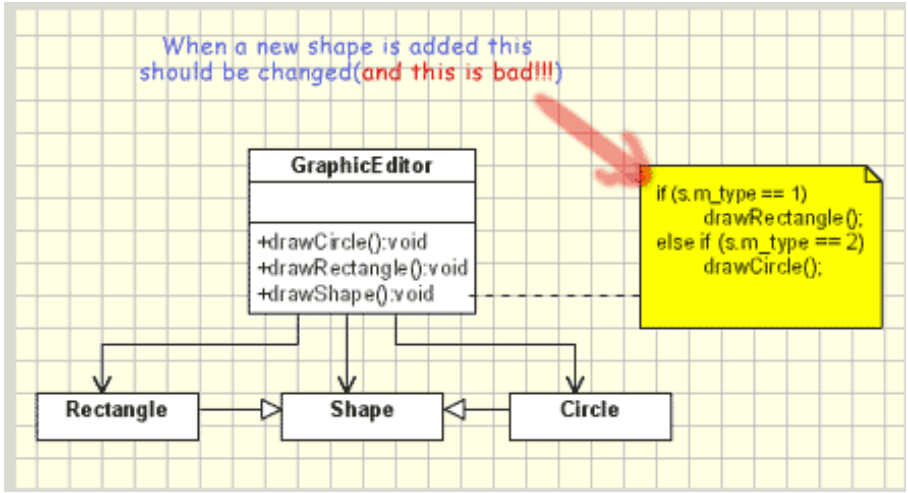
Open Closed Principle

“Objects or entities should be open for extension, but closed for modification.”

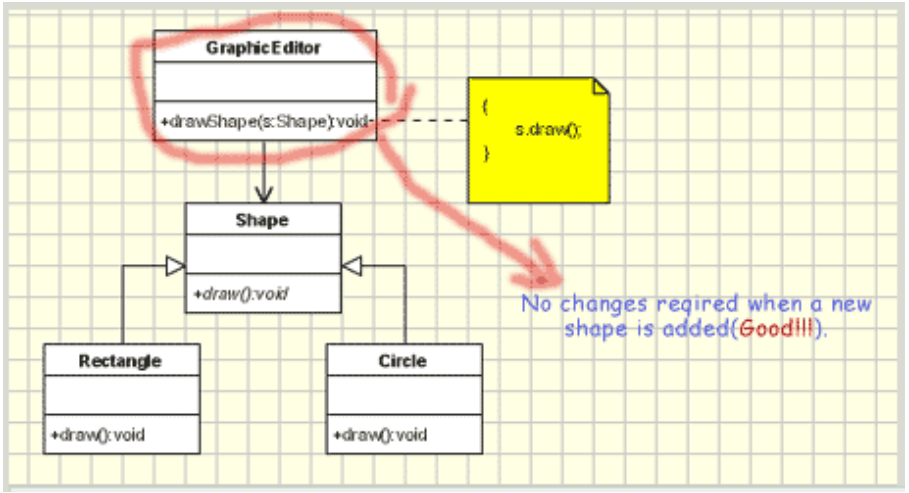
OPEN CLOSED PRINCIPLE

Open Chest Surgery Is Not Needed When Putting On A Coat

✗ Bad Example



✓ Good Example

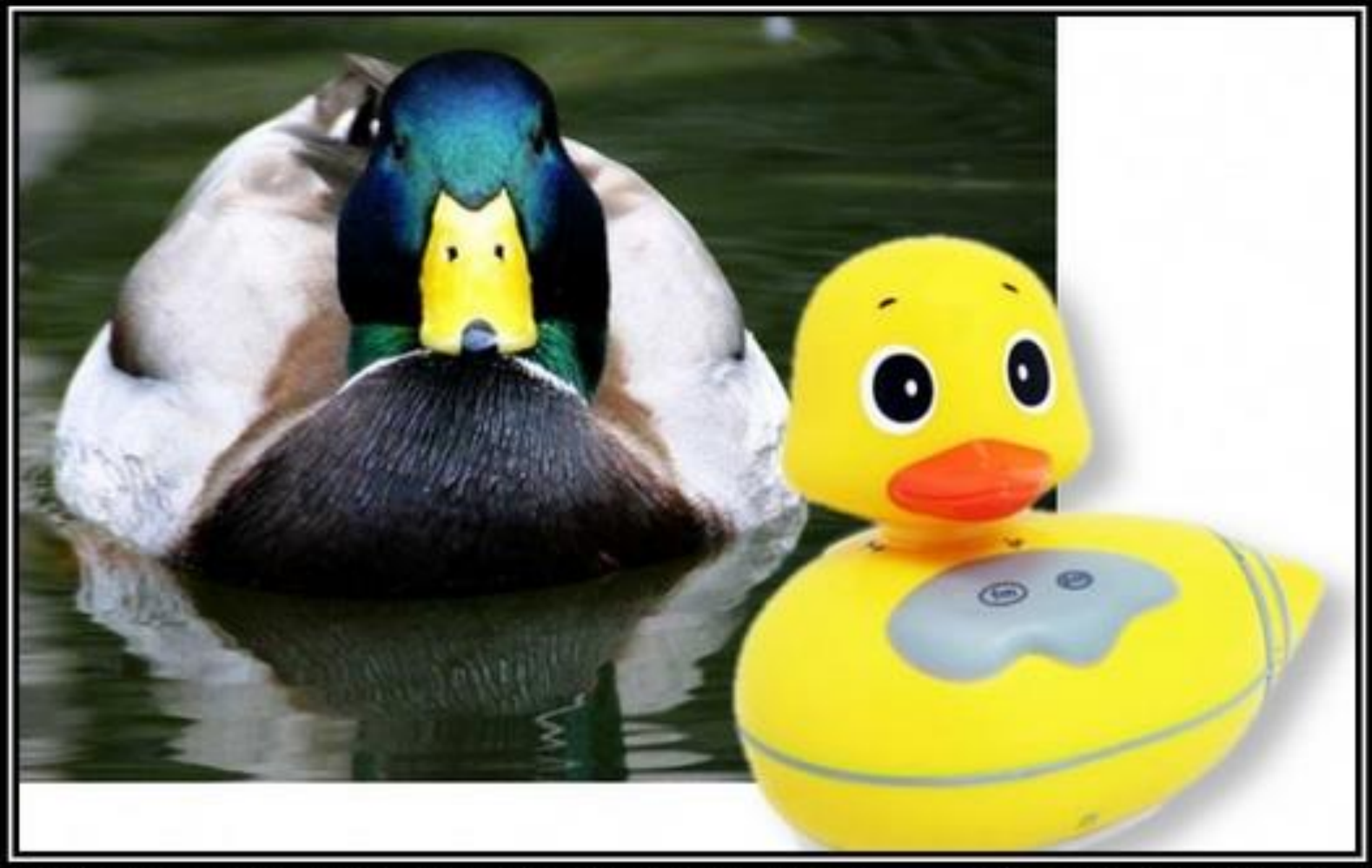


Liskov Substitution Principle

"Let q(x) be a property provable about objects of x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T."

i.e.

"Every subclass should be substitutable for their base class."



LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

- A great example illustrating LSP (given by Uncle Bob in a podcast I heard recently) was how sometimes something that sounds right in natural language doesn't quite work in code.
- In mathematics, a `Square` is a `Rectangle`. Indeed it is a specialization of a rectangle. The "is a" makes you want to model this with inheritance. However if in code you made `Square` derive from `Rectangle`, then a `Square` should be usable anywhere you expect a `Rectangle`. This makes for some strange behavior.
- Imagine you had `SetWidth()` and `SetHeight()` methods on your Rectangle base class; this seems perfectly logical. However if your `Rectangle` reference pointed to a `Square`, then `SetWidth()` and `SetHeight()` doesn't make sense because setting one would change the other to match it. In this case `Square` fails the Liskov Substitution Test with `Rectangle` and the abstraction of having `Square` inherit from `Rectangle` is a bad one.

Interface Segregation Principle

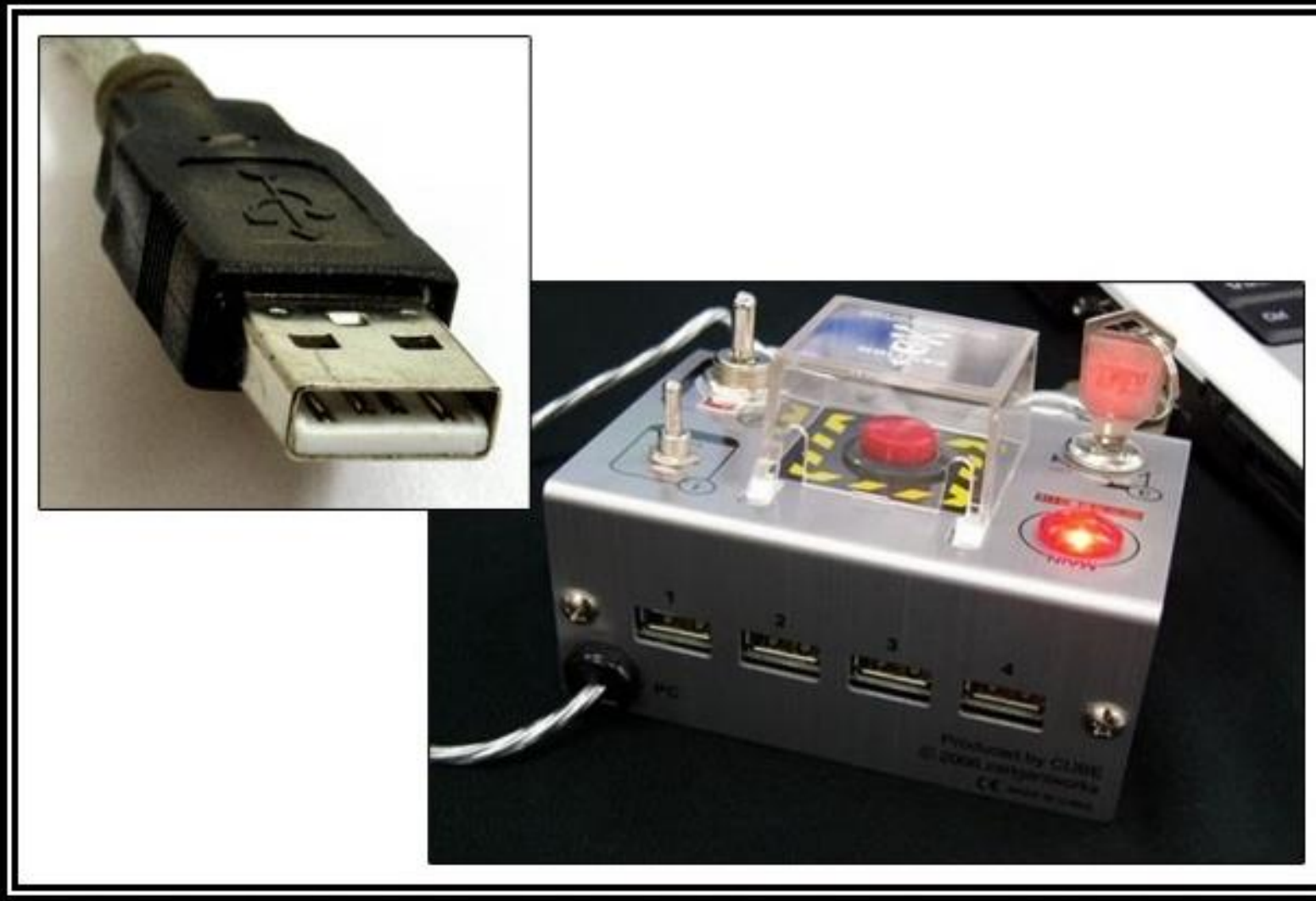
“A client should never be forced to implement an interface that it doesn't use.”

Or,

“Clients shouldn't be forced to depend on methods they do not use.”

i.e.

“Make fine grained interfaces that are client specific.”

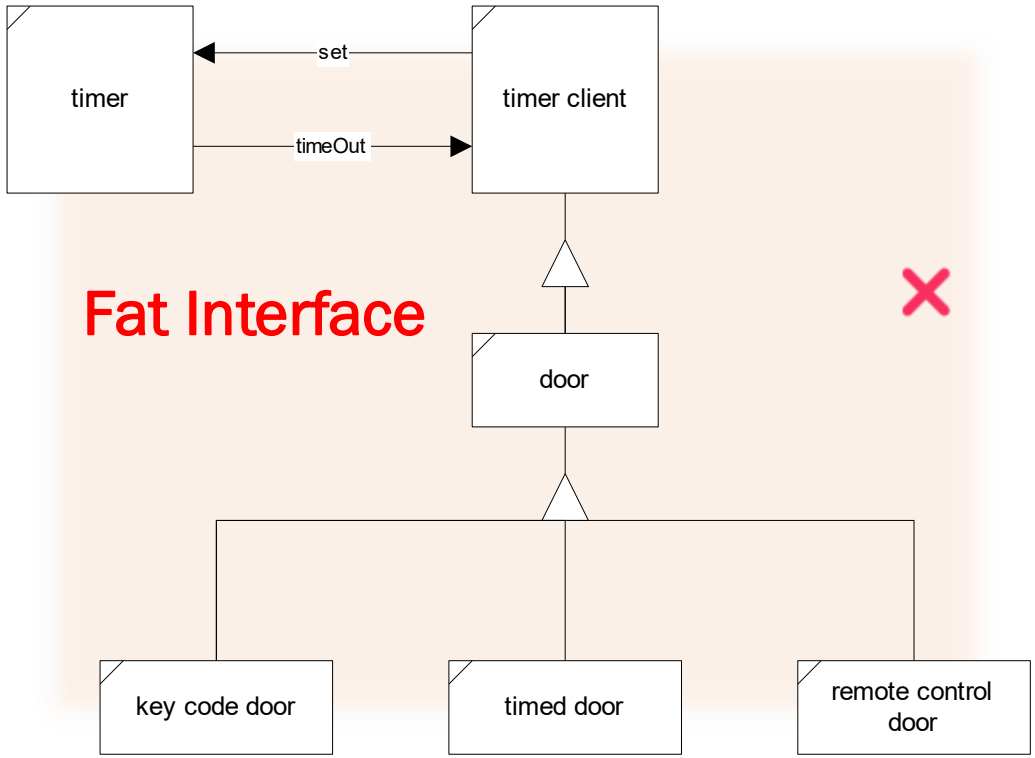


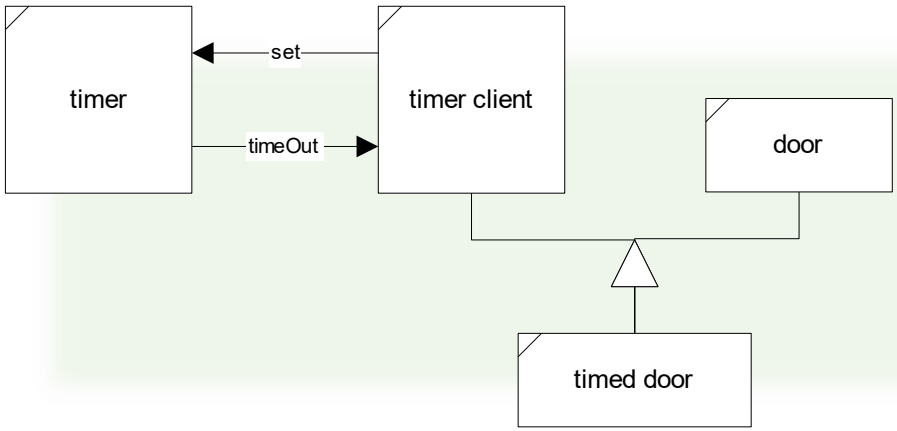
INTERFACE SEGREGATION PRINCIPLE

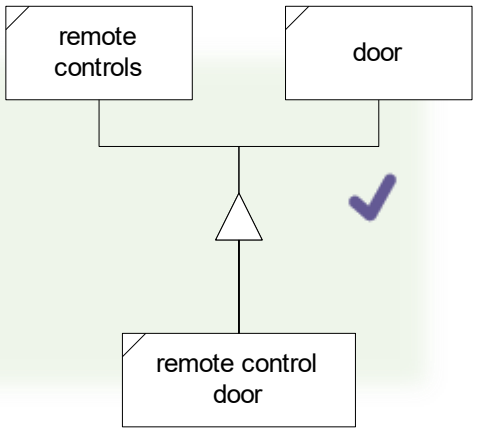
You Want Me To Plug This In, Where?

The Interface Segregation Principle states that:

- fat interfaces lead to inadvertent couplings between clients that ought to be isolated
- fat interfaces can be segregated, through multiple inheritance, into abstract base classes that break unwanted coupling between components.
- clients simply mix-in the appropriate interfaces for their activities.





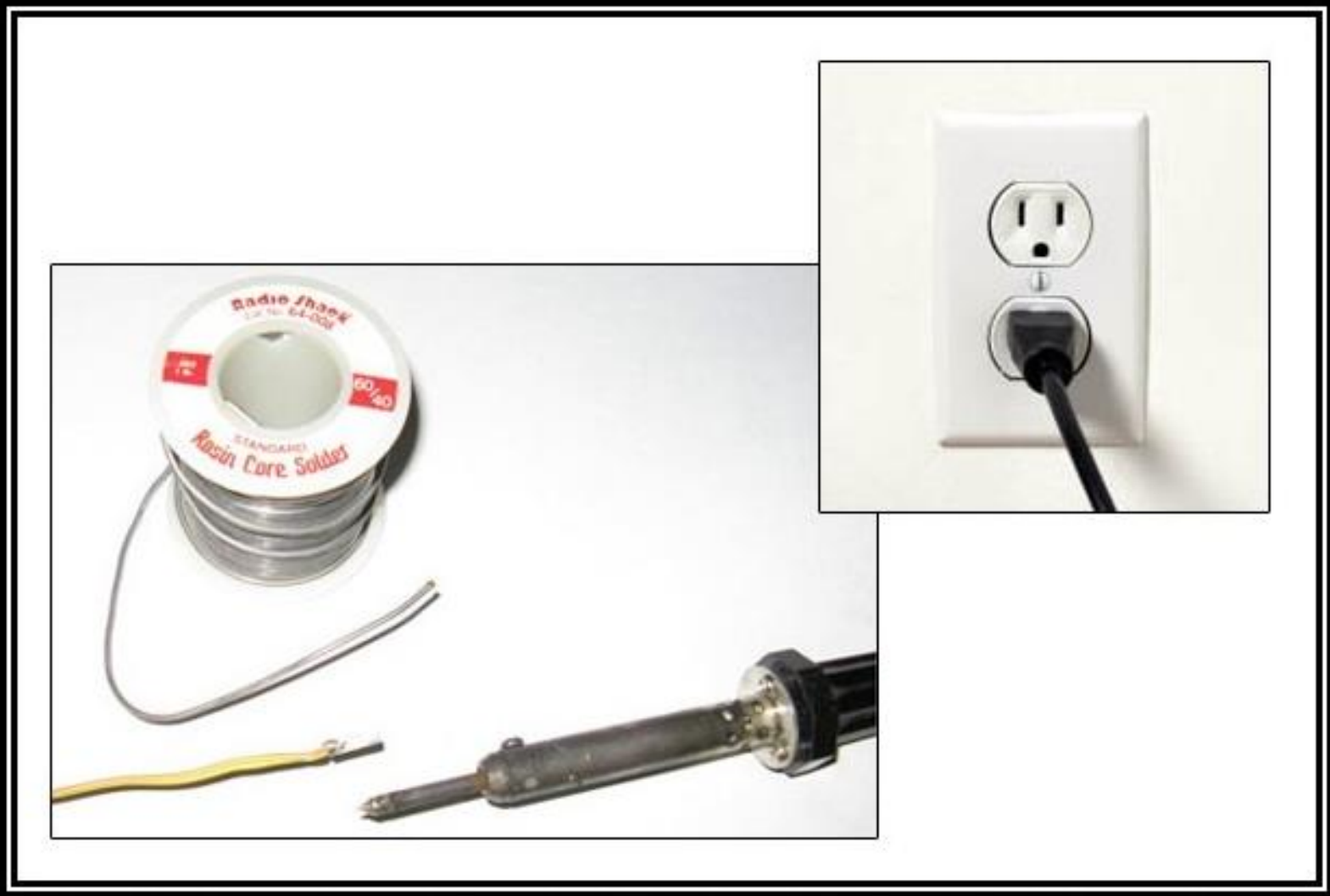


Dependency Inversion Principle

"Entities must depend on abstractions not on concretions."

i.e.

"The high level module must not depend on the low level module, but they should depend on its abstractions."



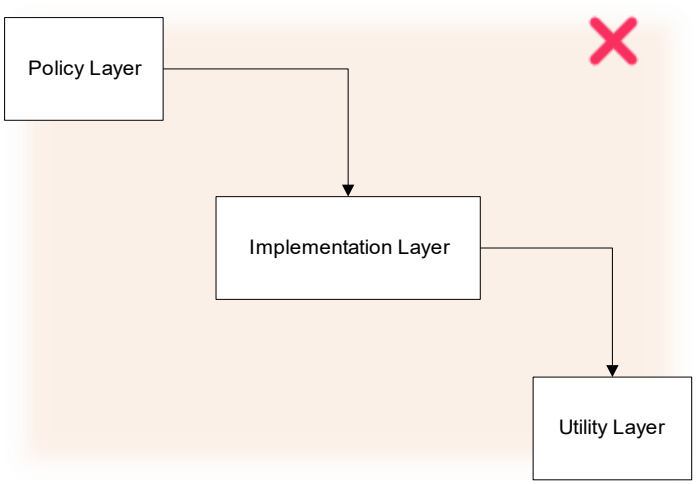
DEPENDENCY INVERSION PRINCIPLE

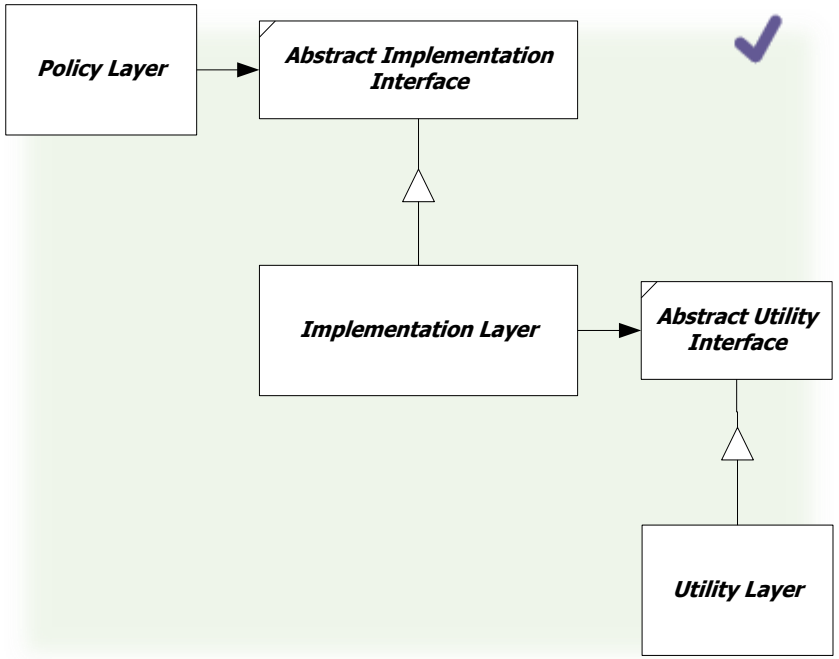
Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

Dependency Inversion Principle

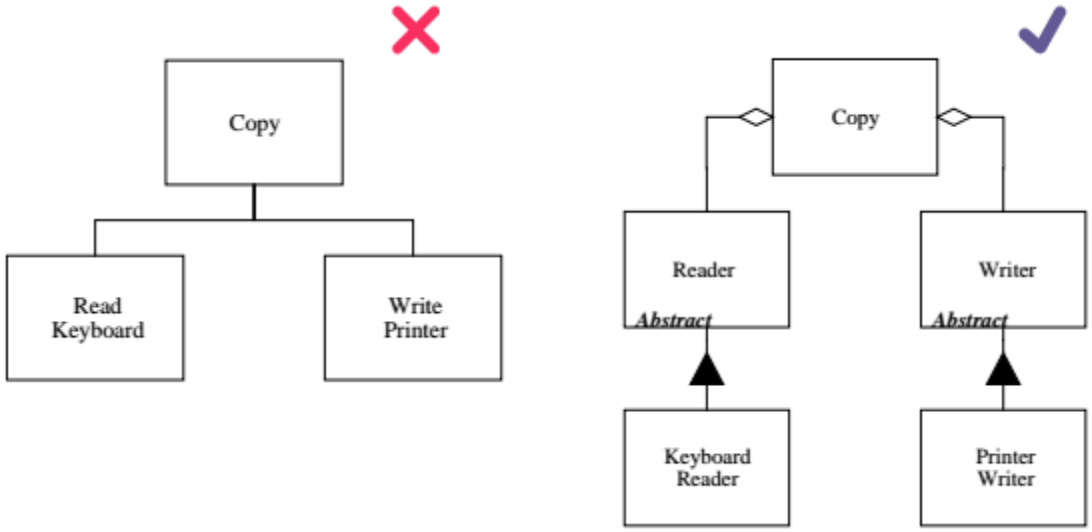
- High level components should not depend upon low level components.
  Instead, both should depend on abstractions.
- Abstractions should not depend upon details. Details should depend upon the abstractions.

- We all can agree that complex systems need to be structured into layers. But if that is not done carefully the top levels tend to depend on the lower levels.
  - On the previous page we show a “standard” architecture that appears to be practical and useful.
  - Unfortunately, it has the ugly property that policy layer depends on implementation layer which depends on utility layer, e.g., dependencies all the way down.





Another Example



```java
// Dependency Inversion Principle -
// Bad example
class Worker {
  public void work() { /* some work */ }
}

class Manager {
  Worker worker;

  public void setWorker(Worker w) { worker = w; }

  public void manage() { worker.work(); }
}

class SuperWorker {
  public void work() { /* some heavy work */ }
}
```

```java
// Dependency Inversion Principle -
// Good example
interface IWorker {
  public void work();
}

class Worker implements IWorker {
  public void work() { /* some work */ }
}

class SuperWorker implements IWorker {
  public void work() { /* some heavy work */ }
}

class Manager {
  IWorker worker;

  public void setWorker(IWorker w) { worker = w; }

  public void manage() { worker.work(); }
}
```

Partitioning into Packages

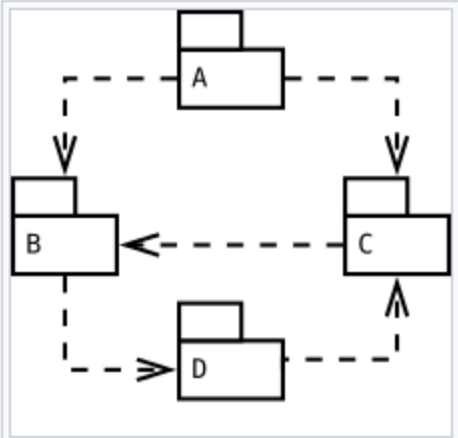
- As software becomes large and complex we need to enforce some form of partitioning that is larger than the class and smaller than a program.
- Packages represent a grouping of classes into a cohesive structure that represents a single high-level abstraction.
- Packages allow us to reason about and reuse software on a large scale without being swamped with detail.

Package Principles (1)

- The first three package principles are about package cohesion, they tell us what to put inside packages:
  - **REP** (The Release Reuse Equivalency Principle): *The granule of reuse is the granule of release.*
  - **CCP** (The Common Closure Principle)*: Classes that change together are packaged together.*
  - **CRP** (The Common Reuse Principle)*: Classes that are used together are packaged together.*

Package Principles (2)

- Coupling between packages:
  - **ADP** (The Acyclic Dependencies Principle)*: The dependency graph of packages must have no cycles.*
  - **SDP** (The Stable Dependencies Principle)*: Depend in the direction of stability.*
  - **SAP** (The Stable Abstractions Principle)*: Abstractness increases with stability.*



Circular dependency example

Sidebar
UML 0.9 Lexicon

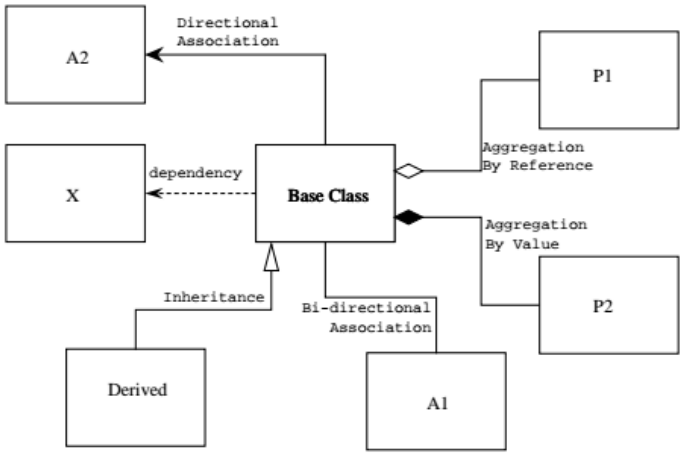


Figure 3
Package Diagram *with* Cycles

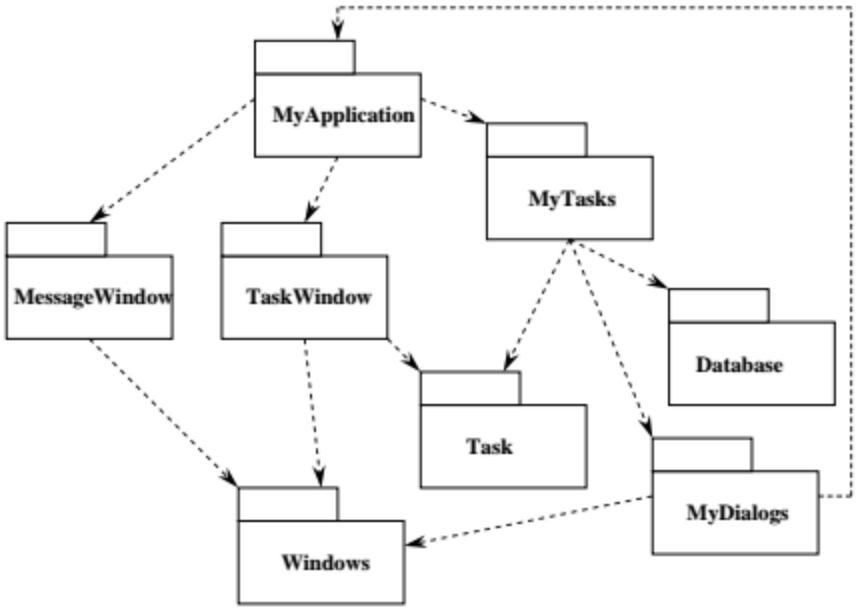
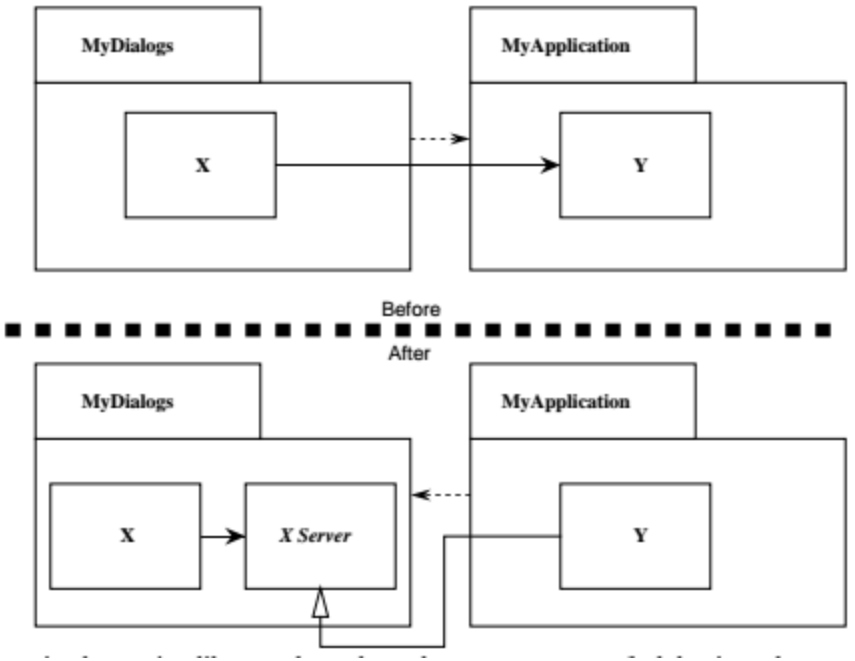


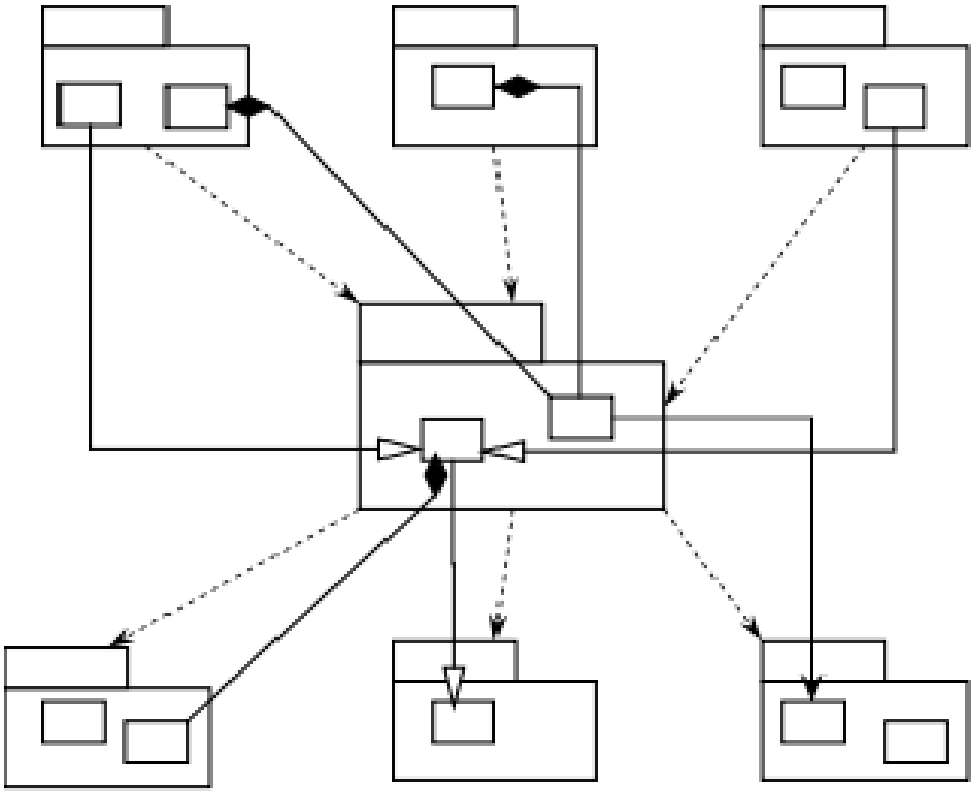
Figure 4
Breaking the Cycle with Dependency Inversion



Measuring Dependency: Positional Stability

- Positional stability is based on the number of dependencies that enter and leave a package:
  - **Afferent Couplings:**
    Ca = number of classes outside package depending on classes inside package
  - **Efferent Couplings:**
    Ce = number of classes inside package that depend on outside classes
  - **Instability:**
    $I = Ce/(Ca + Ce)$
    $I \in [0,1]$, I=0 $\Rightarrow$ maximum stability, I=1 $\Rightarrow$ minimum stability
- If we are careful only to `#include` files that we depend on and we isolate one class per file, then we can compute I by counting `#includes`.

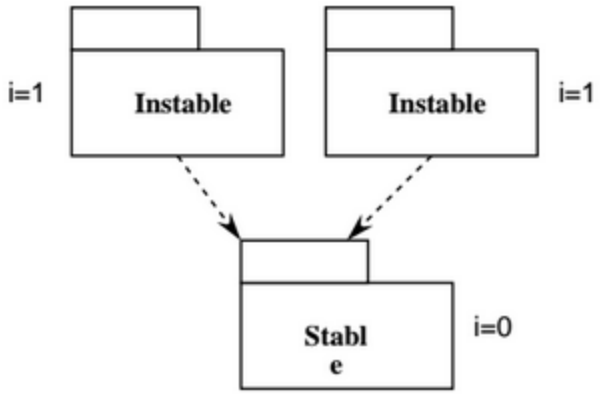
Measuring Dependency (2)



$Ca=4$
$Ce=3$
$I=3/7$

Not all packages should be stable

- If all the packages in a system were maximally stable, the system would be unchangeable.
  - This is not a desirable situation.
- We'd want to design our package structure so that some packages are instable and some are stable, e.g.:



*"The changeable packages are on top and depend on the stable package at the bottom. Putting the instable packages at the top of the diagram is my own convention. By arranging them this way then any arrow that puts **up** is **violating the SDP**."*

Measuring Abstraction: Stable Abstractions Principle

- A measure of abstraction is the ratio of the number of abstract classes to total number of classes:

  A = number of abstract classes / total number of classes

  $A \in [0,1]$.
  $A = 1 \Rightarrow$ maximum abstraction, $A = 0 \Rightarrow$ minimum abstraction

- The distance of a design from the balanced state is measured by:

  $D = A + I - 1$

  This is the perpendicular distance of the design from the balanced line on the following chart.

Measuring Abstraction (2)