1

Object vs Structured design

2

S.O.L.I.D design principles

3

Architectural design principles

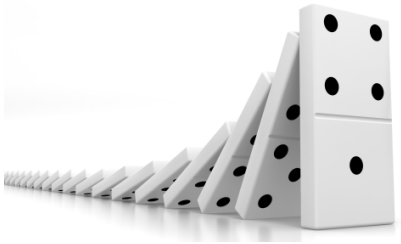
Single Responsibility Principle

Open-Closed Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle



S.O.L.I.D.

Principles

Open-Closed Principle (OCP)

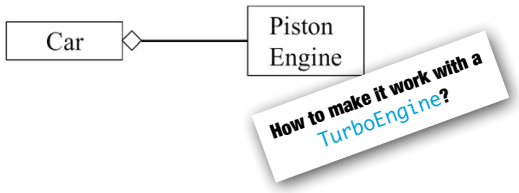
Software entities should be **open for extension**,

but **closed for modification**

Modules should be written so they can be
extended without requiring them to be modified

Signs that a module is not closed

Sign #1: Dependency on concrete providers



Car **needs to be changed heavily!**

Sign #2: Checking Runtime Type Information (RTTI)

shape.h
```
enum ShapeType {circle, square};

struct Shape {
  ShapeType itsType;
};
```

square.h
```
struct Circle {
  ShapeType itsType;
  double itsSide;
  Point itsTopLeft;
};

void DrawSquare(struct Square*);
```

circle.h
```
struct Circle {
  ShapeType itsType;
  double itsRadius;
  Point itsCenter;
};

void DrawCircle(struct Circle*);
```

drawAllShapes.cc
```
typedef struct Shape *ShapePointer;
void DrawAllShapes(ShapePointer list[], int n) {
  int i;
  for (i=0; i<n; i++) {
    struct Shape* s = list[i];
    switch (s->itsType) {
      case square: DrawSquare((struct Square*)s); break;
      case circle: DrawCircle((struct Circle*)s); break;
    }
  } }
```

from R.C.Martin, M.Micah - Agile Principles, Patterns, and Practices in C#, 2006

RTTI can take many different forms…

flags

instanceof

enums

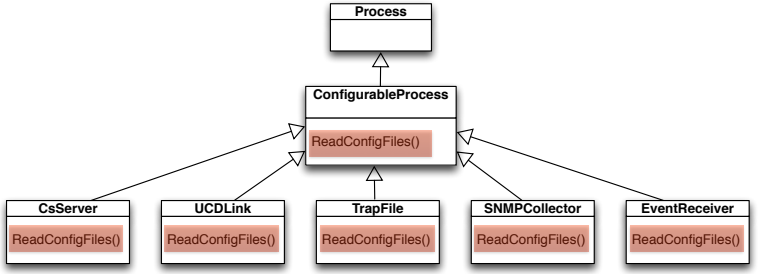
is

dynamic_cast

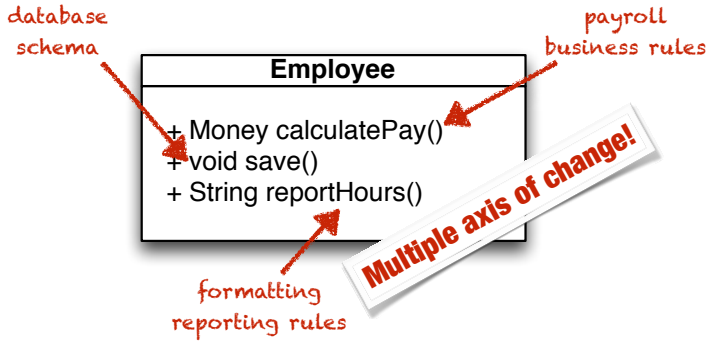
It is usually a sign that the hierarchy "cries"

for a dynamically bound service.

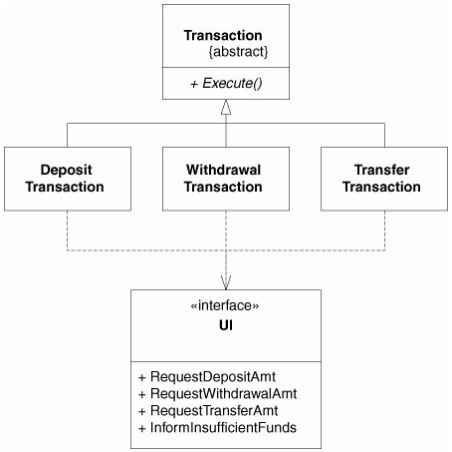
Sign #3: Code Duplication



Sign #4: Schizophrenic Responsibilities

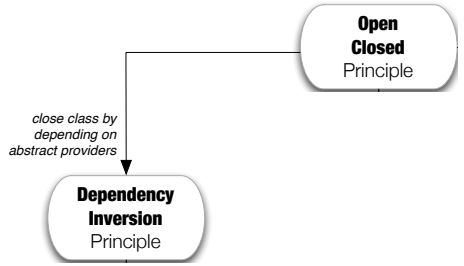


Sign #5: Unexpected dependencies

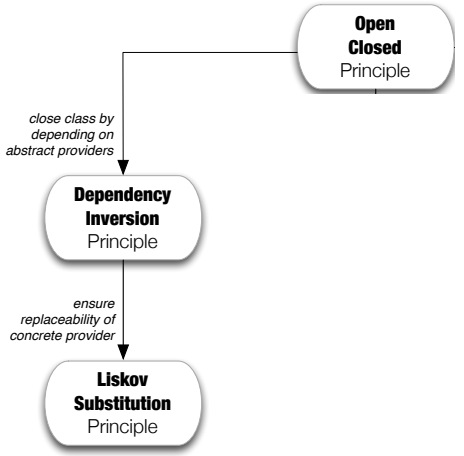


from R.C.Martin, M.Micah - Agile Principles, Patterns, and Practices in C#, 2006

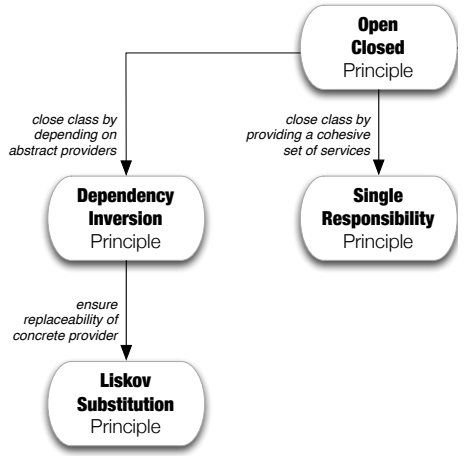
Remaining principles help us to close modules



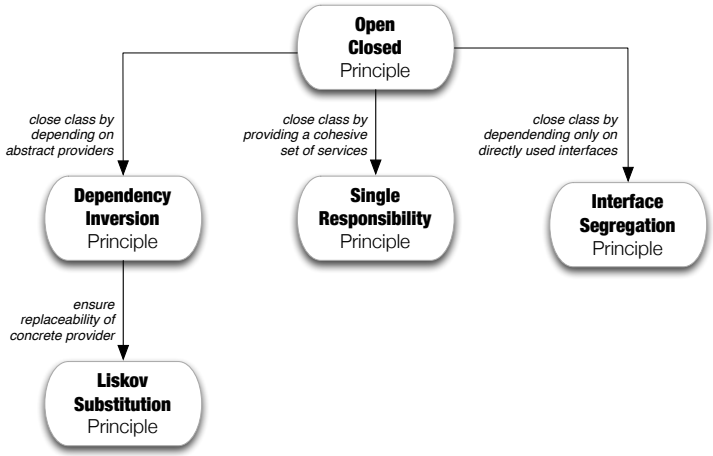
Remaining principles help us to close modules

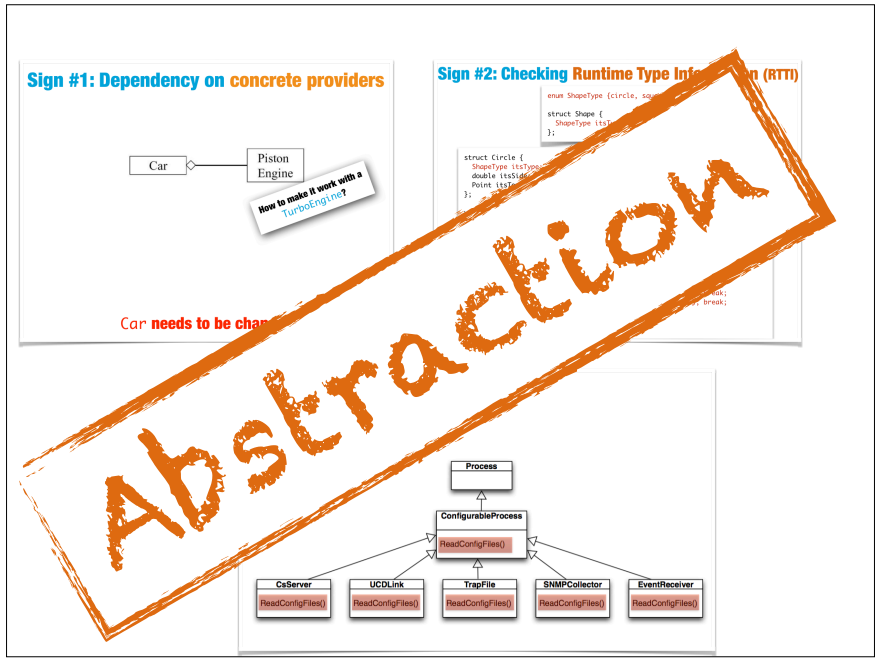


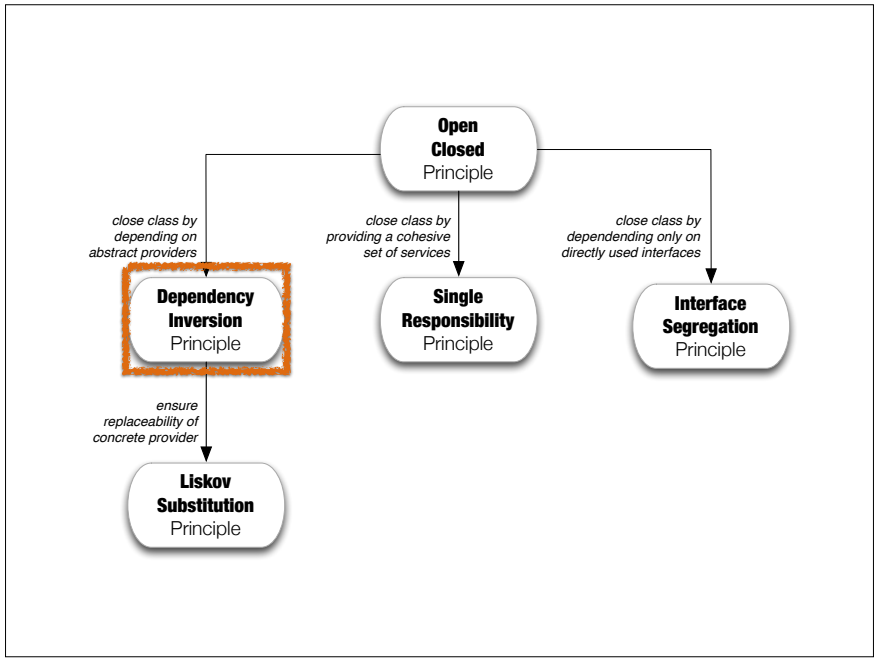
Remaining principles help us to close modules



Remaining principles help us to close modules



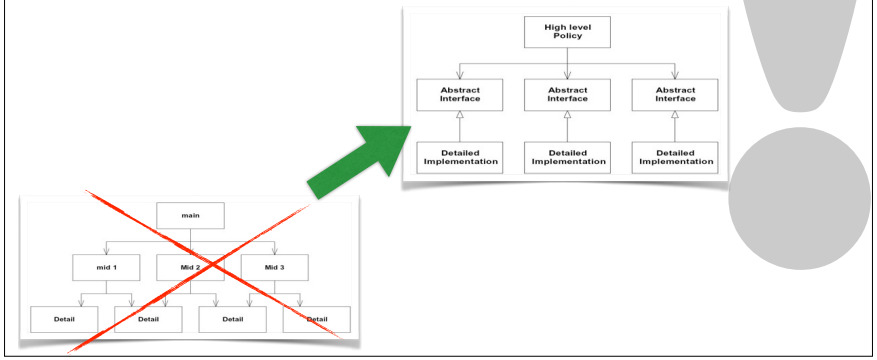




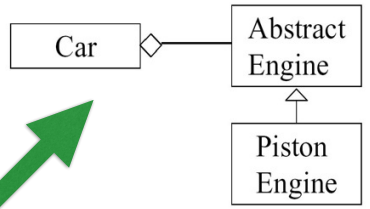
Dependency Inversion Principle (DIP)

High level modules should **not depend upon low level** modules.
Both should **depend upon abstractions**.

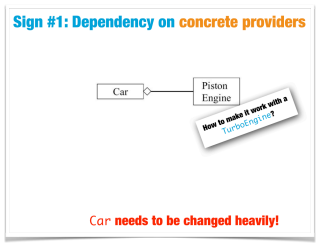
Design to an **interface**, not to an **implementation**!



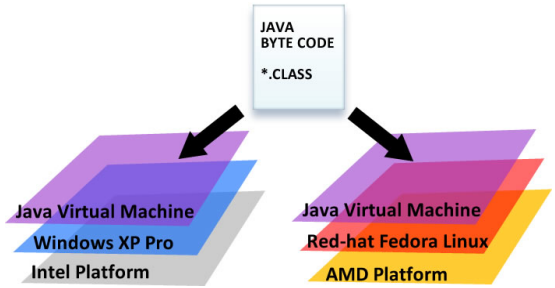
Depending on abstractions



Car **should not depend on any concrete** Engine**!**

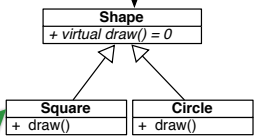


Why are Java applications portable?



Define abstractions

```
void DrawAllShapes(ArrayList<Shape> list) {
      for(Shape shape : list)
        shape.draw();
}
```



Closed to adding new Shapes!

Sign #2: Checking Runtime Type Information (RTTI)

shape.h

```
enum ShapeType {circle, square};
struct Shape {
  ShapeType itsType;
};
```

square.h

```
struct Circle {
  ShapeType itsType;
  double itsSide;
  Point itsTopLeft;
};
void DrawSquare(struct Square*);
```

circle.h

```
struct Circle {
  ShapeType itsType;
  double itsRadius;
  Point itsCenter;
};
void DrawCircle(struct Circle*);
```

drawAllShapes.cc

```
typedef struct Shape *ShapePointer;
void DrawAllShapes(ShapePointer list[], int n) {
  int i;
  for (i=0; i<n; i++) {
    struct Shape* s = list[i];
    switch (s->itsType) {
      case square: DrawSquare((struct Square*)s); break;
      case circle: DrawCircle((struct Circle*)s); break;
    }
  }
}
```

What if `Circle` objects must be drawn first?

Strategic Closure

No significant program can be **100% closed**.
We should seek not complete, but **strategic closure**!

1. **Abstraction to gain strategic closure**
   - insert extension "hooks" in the class
2. **Data-driven approach to gain strategic closure**
   - externalize volatile decisions in a separate file (preferable a configuration file)

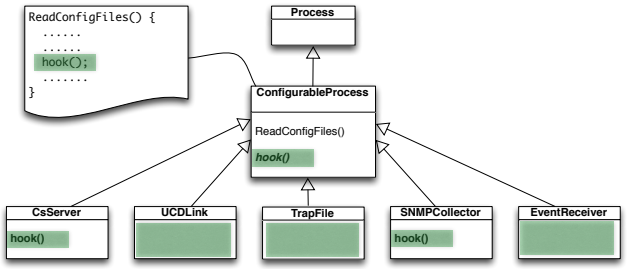
Data-driven approach to gain closure

```
public class ShapeComparer : IComparer {
  private static Hashtable priorities = new Hashtable();

  static ShapeComparer() {
    priorities.Add(typeof(Circle), 1);
    priorities.Add(typeof(Square), 2);
  }

  private int PriorityFor(Type type) {
    if(priorities.Contains(type)) return (int)priorities[type];
    return 0;
  }

  public int Compare(object o1, object o2) {
    int priority1 = PriorityFor(o1.GetType());
    int priority2 = PriorityFor(o2.GetType());
    return priority1.CompareTo(priority2);
  }
}
```

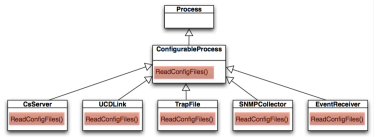
```
public void DrawAllShapes(ArrayList shapes) {
    shapes.Sort(new ShapeComparer());
    foreach(Shape shape in shapes)
      shape.Draw();
}
```

from R.C.Martin, M.Micah - Agile Principles, Patterns, and Practices in C#, 2006

Abstract a customizable algorithm



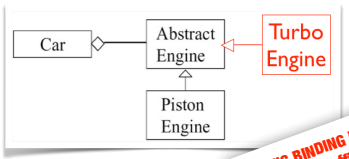
Sign #3: Code Duplication



Apply
Template Method
pattern

Any client-code which can legally call another class's methods
must be able to substitute any subclass of that class without modification

is TurboEngine
a proper substitution?

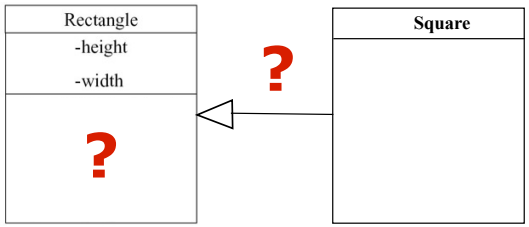


DYNAMIC BINDING is necessary, but insufficient!

Substitution is about semantics!

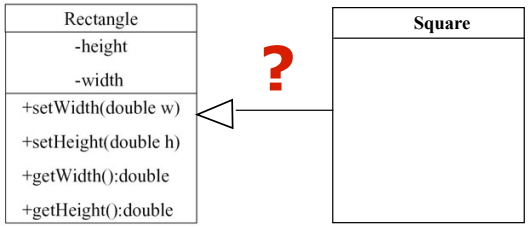
Substitution is about behavior!

Another troubling question: Square IS-A Rectangle?



from R.C.Martin, M.Micah - Agile Principles, Patterns, and Practices in C#, 2006

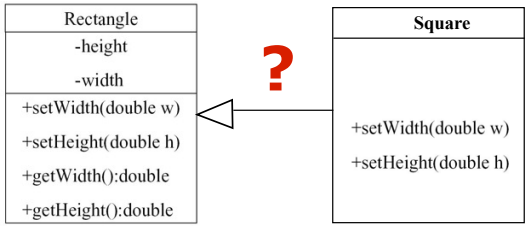
Square IS-A Rectangle?



from R.C.Martin, M.Micah - Agile Principles, Patterns, and Practices in C#, 2006

Square IS-A Rectangle?

IT DEPENDS!




```
void shapeClient(Rectangle& r) {
   r.setWidth(5); r.setHeight(4);
  // How large is the area?
}
```

what if r is a Square!?!

from R.C.Martin, M.Micah - Agile Principles, Patterns, and Practices in C#, 2006

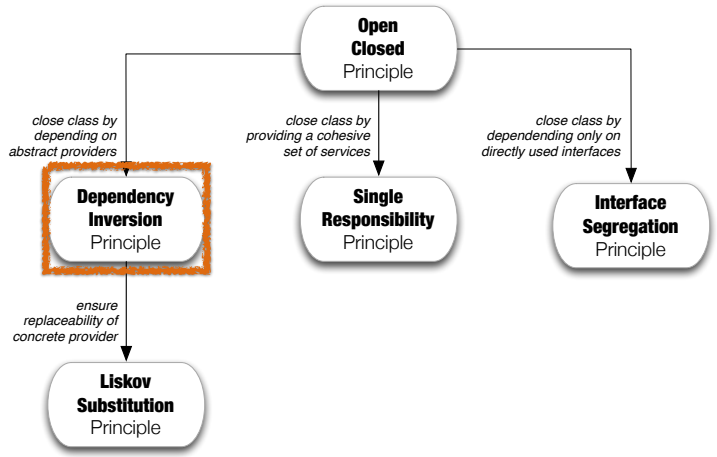
IS-A relation judged by compatible behavior!

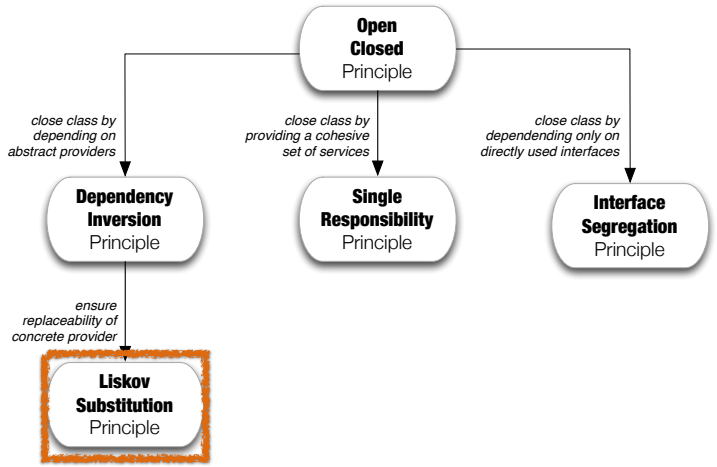
Interface = Signature + Contract

Contract of a function has 3 parts:

1. **preconditions**: what does the function **require** to run correctly
2. **postconditions**: what result does the function **guarantee**
3. **invariants**: what does the function **guarantee** to be preserved

Design By Contract (DBC) = use contracts to specify interfaces





Liskov Substitution Principle

When redefining a method in a derivate class,
you **may only replace its precondition by a weaker** one,
and its **postcondition by a stronger** one

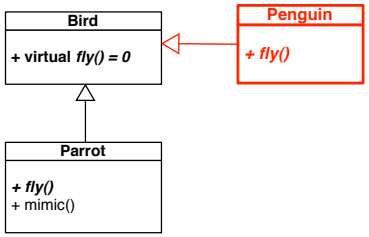
Derived classes, should require no more and promise no less!

Example...

```
int Base::f(int x);
// REQUIRE: x is odd
// PROMISE: return even int
```

```
int Derived::f(int x);
// REQUIRE: x is int
// PROMISE: return 8
```

Counter-example




```
class Penguin extends Bird {
   ...
   public void fly() {
      error ("Penguins don't fly!");
   }
};
```

?

Does NOT model: "Penguin can't fly"

```
void PlayWithBird (Bird aBird) {
    aBird.fly();    // OK if Parrot.
}
```

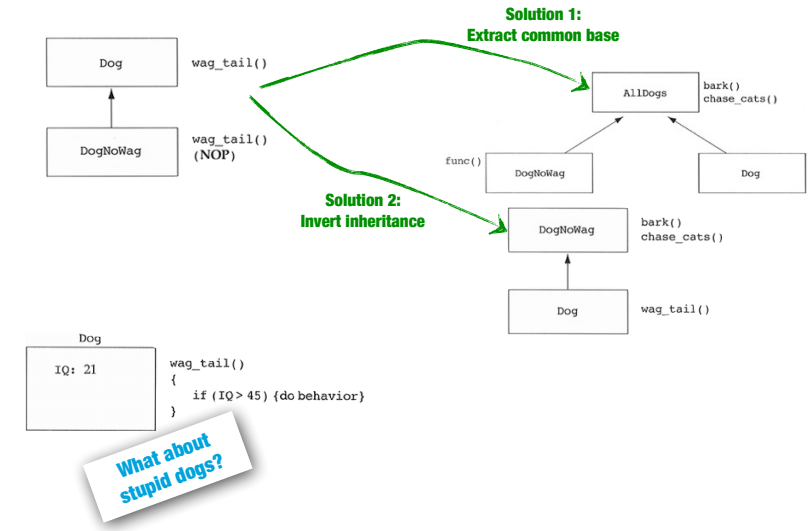
Fails LSP!

from R.C.Martin, M.Micah - Agile Principles, Patterns, and Practices in C#, 2006

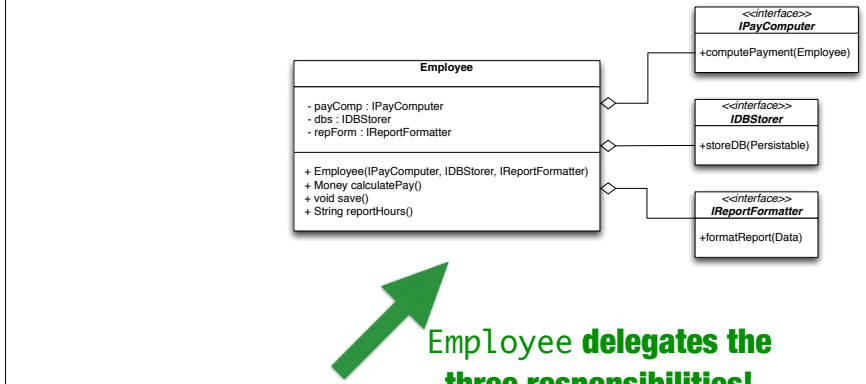
Liskov Substitution Principle (corollary)

It is illegal for a derived class, to override a base-class method
with a **NO-Operation (NOP) method.**

Avoiding NOP overrides

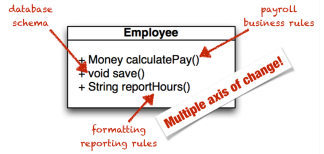


from A. Riel - Object-Oriented Design Heuristics, 1996



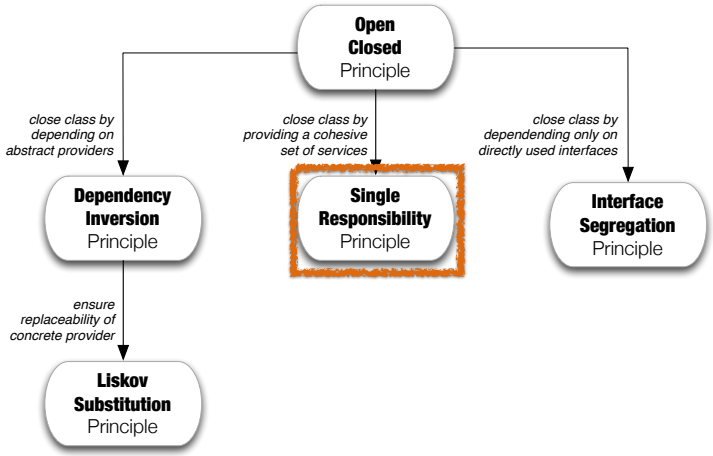
Employee **delegates the three responsibilities!**

Sign #4: Schizophrenic Responsibilities



Write a brief description of the class in about 25 words **without** using the words "**if**", "**and**", "**or**", "**but**"

Is it hard?



Single Responsibility Principle (SRP)

A class or module should have one, and **only one, reason to change**

"*Fool me once shame on you; fool me twice shame on me*"
…. but **within a class**
you shouldn't be able to "*fool me once*" frequently!

An axis of change is an axis of change
only if the changes actually occur!

but doesn't this require more **navigation**

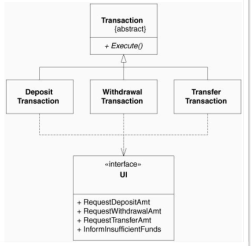
and more **effort to understand** a

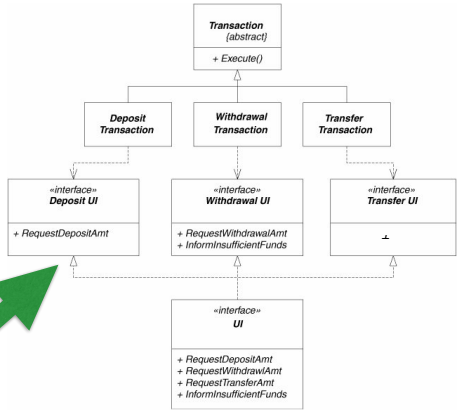
large piece of functionality

What do you prefer?

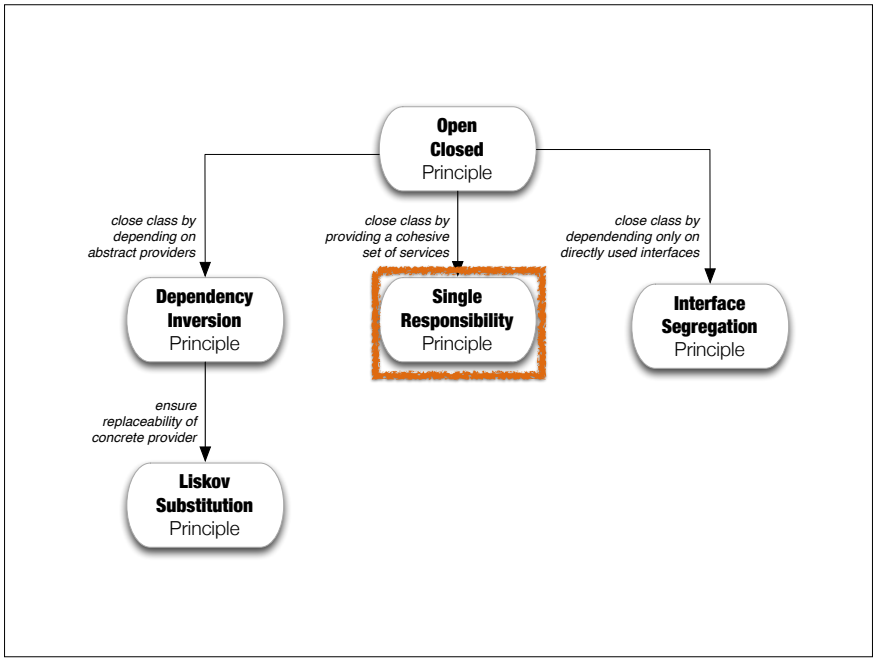
A system with larger, multipurpose classes always hampers us
by insisting we deal with lots of things that we don't need to know right now

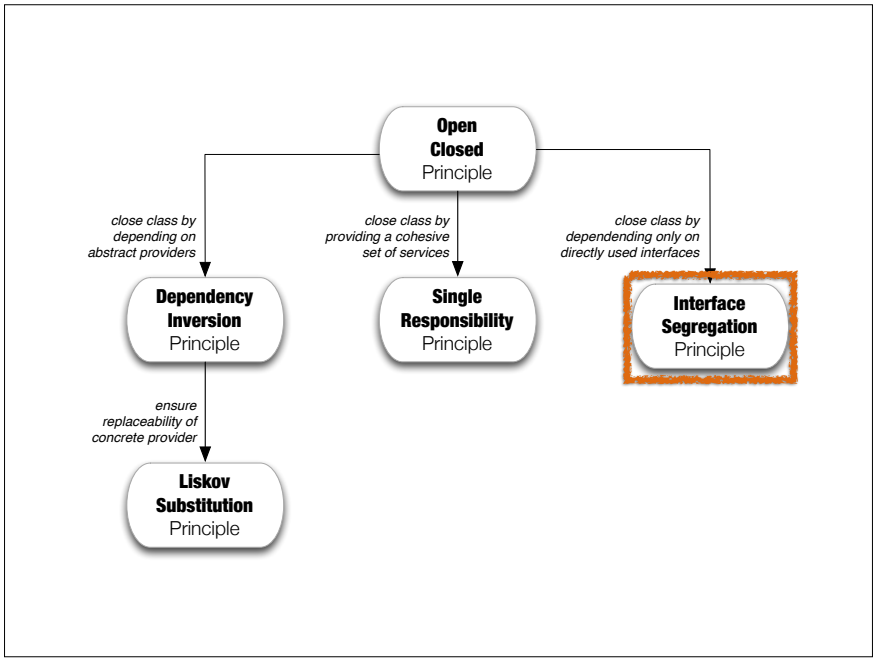
Sign #5: Unexpected dependencies





Segregate the ATM's UI interface!



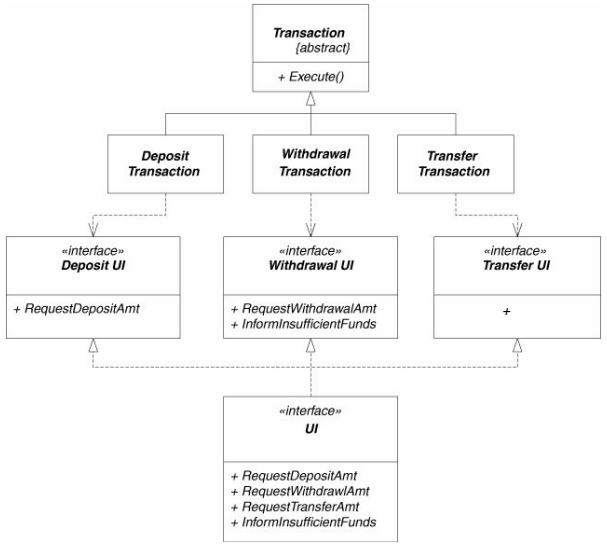


Interface Segregation Principle (SRP)

Clients should never be forced to depend on those methods of a **provider class** that it does **not use**

Avoid for a client to be affected by changes that **other clients** force on the provider class, due to interface methods **unilaterally** used by the latter class.

Segregated ATM UI Interface



from R.C.Martin, M.Micah - Agile Principles, Patterns, and Practices in C#, 2006

Segregated ATM UI Interface

```
interface Transaction {
  void Execute();
}

interface DepositUI {
  void RequestDepositAmount();
}

class DepositTransaction implements Transaction {
  privateDepositUI depositUI;

  public DepositTransaction(DepositUI ui) {
    depositUI = ui;
  }
  public virtual void Execute() {
    /*code*/
    depositUI.RequestDepositAmount();
    /*code*/
  }
}

/** other transaction classes **/

public interface UI : DepositUI, WithdrawalUI, TransferUI {
}
```

from R.C.Martin, M.Micah - Agile Principles, Patterns, and Practices in C#, 2006

Dilemma of a `client` function...

```
void client(DepositUI depUI, TransferUI transUI) { … }
```

vs.

```
void client(UI ui) { … }
```

```
void someFunction(UI ui) {
   /* … */
   client(ui, ui);
   /* … */
}
```

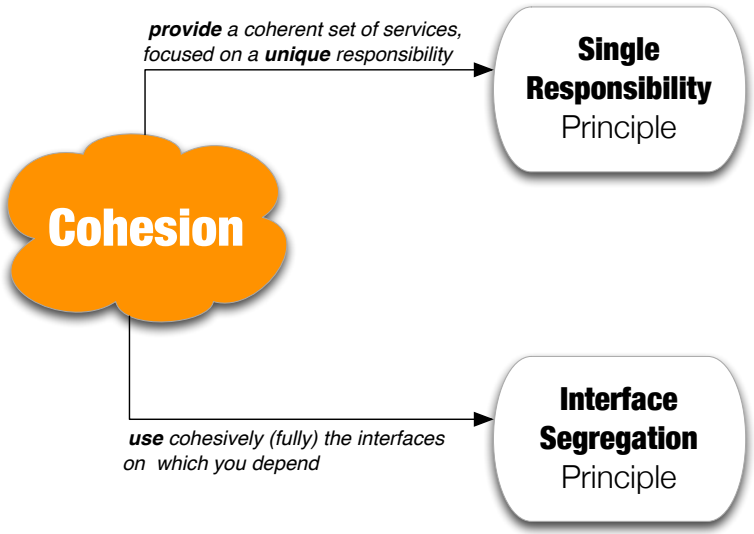
✅

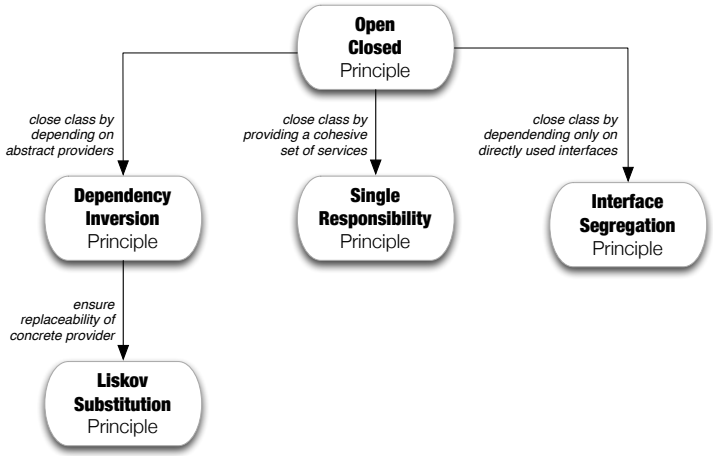
vs.

```
void someFunction(UI ui) {
   /* … */
   client(ui);
   /* … */
}
```

❌

ISP and SRP show two facets of cohesion





1

Object vs Structured design

2

S.O.L.I.D design principles

3

Architectural design principles

1

Object vs Structured design

2

S.O.L.I.D design principles

3

Architectural design principles

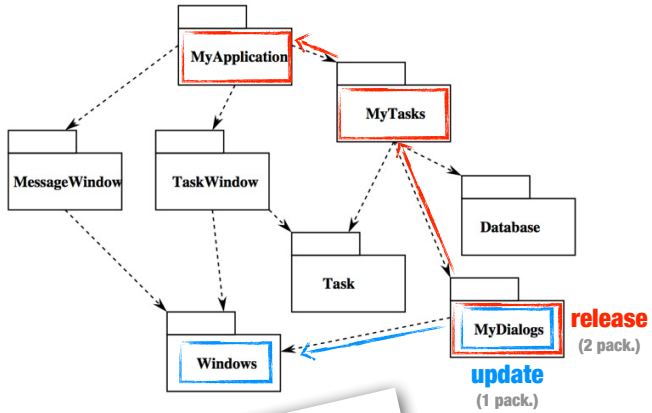
Morning-after syndrome...

many developers modify the same source files

everyone tries to adapt to the changes the **others made**

Weekly Build: a solution that does not scale!

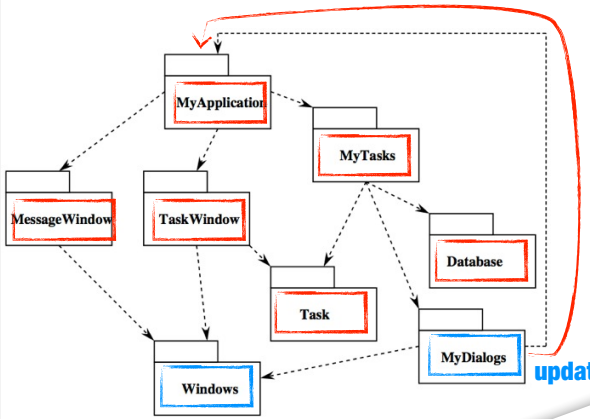
Solution: depend on release versions



System can be built bottom-up!

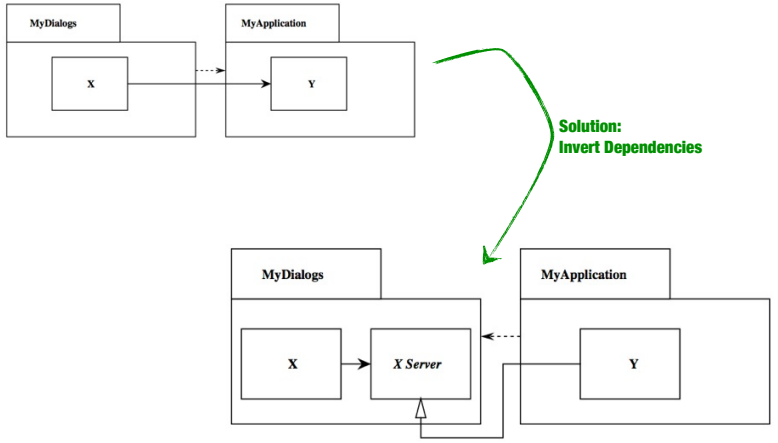
from R.C.Martin, M.Micah - Agile Principles, Patterns, and Practices in C#, 2006

Evil of cyclic dependencies



All components must be released simultaneously!

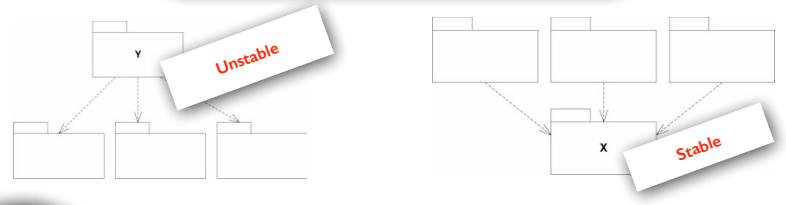
Solution: invert dependencies!



from R.C.Martin, M.Micah - Agile Principles, Patterns, and Practices in C#, 2006

Stability of a component

Stability = Responsibility + Autonomy



from R.C.Martin, M.Micah - Agile Principles, Patterns, and Practices in C#, 2006

Measuring the stability of a component

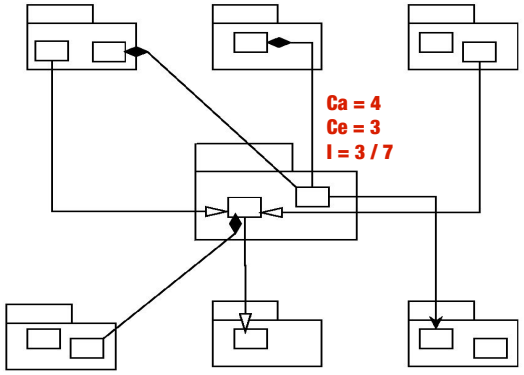
Ce **Efferent coupling = how dependent?**
number of external classes on which the component depends (FANOUT)

Ca **Afferent coupling = how responsible?**
number of external classes that depend on this component (FANIN)

$$I = \frac{C_e}{C_a + C_e}$$

from R.C.Martin, M.Micah - Agile Principles, Patterns, and Practices in C#, 2006

An example



from R.C.Martin, M.Micah - Agile Principles, Patterns, and Practices in C#, 2006

Stable Dependencies Principle

A component should **only depend** upon components that are **more stable than it is**.

A component should depend only on component whose **I metric is lower** than theirs!

Ideal Architecture

1. **Most unstable (changeable) components on top**
2. **Most stable (hard to change) components at the bottom**

Doesn't stable mean rigid?

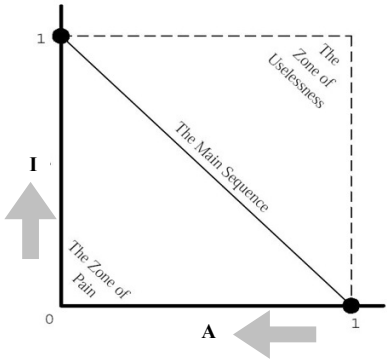
Not if you make them abstract!

Stable Abstractions Principle

The **stability of a component should be proportional to its abstraction!**

1. **Components that are maximally stable should be maximally abstract.**
2. **Unstable components should be concrete.**

When abstraction decreases, instability should increase



$$A = \frac{NoAbstractClasse}{NoClasses}$$

from R.C.Martin, M.Micah - Agile Principles, Patterns, and Practices in C#, 2006