AARHUS UNIVERSITET

Software Engineering and Architecture

Compositional Design Principles

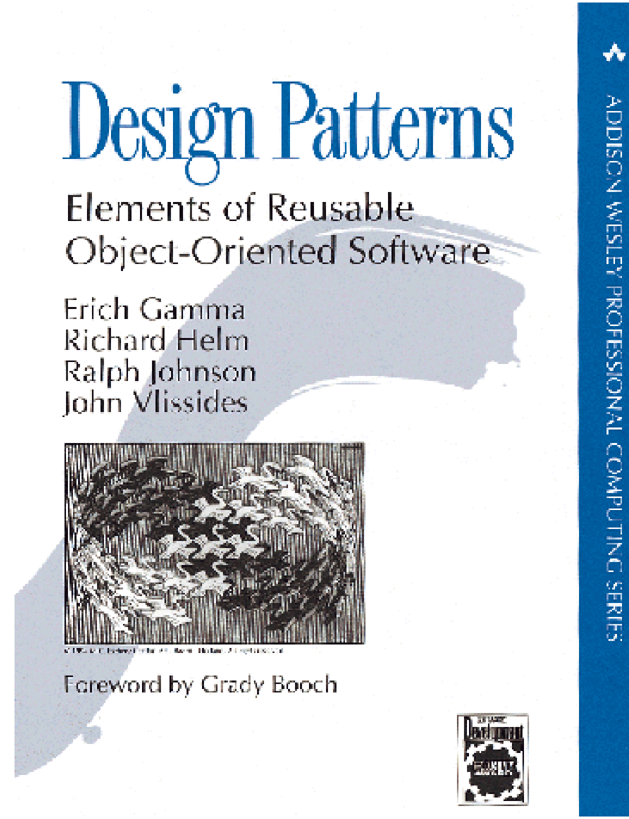
Gang of Four (GoF)

Erich Gamma, Richard Helm
Ralph Johnson & John Vlissides

Design Patterns – Elements of Reusable Object-Oriented Software

Addison-Wesley, 1995.
(As CD, 1998)

First systematic software pattern description.



The most important chapter

- Section 1.6 of GoF has a section called:
- **How design patterns solve design problems**
    - *This section is the gold nugget section*
- It ties the patterns to the underlying coding principles that delivers the real power.

Compositional Design Principles

Compositional Design Principles:

① *Program to an interface, not an implementation.*

② *Favor object composition over class inheritance.*

③ *Consider what should be variable in your design.*
(or: Encapsulate the behavior that varies.)

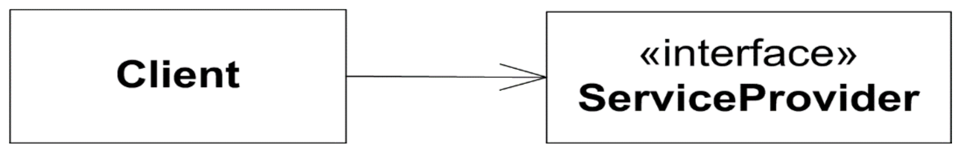
As the 3-1-2 process

③ I identified some behavior that was likely to change…	=	③ *Consider what should be variable in your design.*
① I stated a well-defined responsibility that covers this behavior and expressed it in an interface…	=	① *Program to an interface, not an implementation.*
② Instead of implementing the behavior ourselves I delegated to an object implementing the interface…	=	② *Favor object composition over class inheritance.*

First Principle

GoF's 1st principle

- *Program to an interface, not an implementation*



- In other words
    - **Assume only the role**
    - **(the responsibilities + protocol)**

- … and *never* allow yourself to be coupled to implementation details and concrete behavior

AARHUS UNIVERSITET

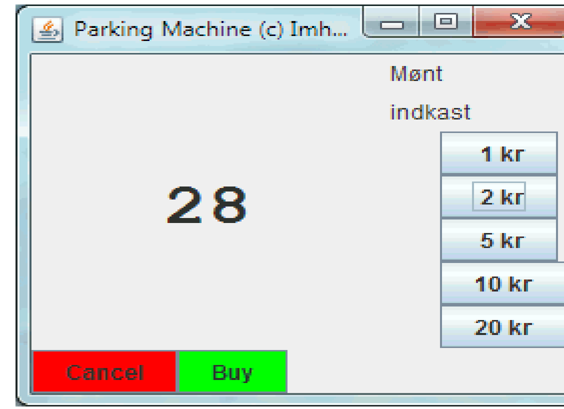
First Principle

- *Program to an interface* because
  - You only collaborate with the **role** – not an individual object
  - You are *free* to use *any* service provider class!
    - Any class that implements that interface…
  - You do not delimit other developers for providing *their* service provider class!
  - You avoid binding others to a particular inheritance hierarchy
    - Which you would do if you use (abstract) classes…

Example

- Early pay station GUI used JLabel for visual output

```
public class ParkingMachineGUI extends JFrame {
    JLabel display;
    ParkingMachine parkingMachine;
```



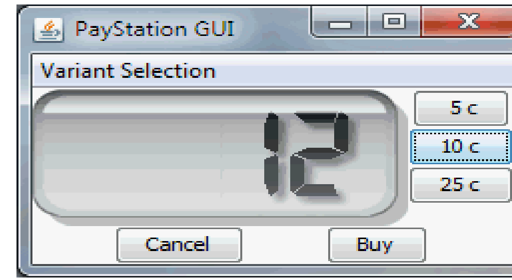
- I only use method: ’setText()’

```
public void updateDisplay() {
  display.setText( ""+parkingMachine.readDisplay() );
}
```

Example

- The I found SoftCollection’s number display, got permission to use it, but...

```
public class ParkingMachineGUI extends JFrame {
    /** The "digital display" where readings are shown */
    LCDDigitDisplay display;
    /** The domain pay station that the gui interacts with */
    PayStation payStation;
```



... And use:

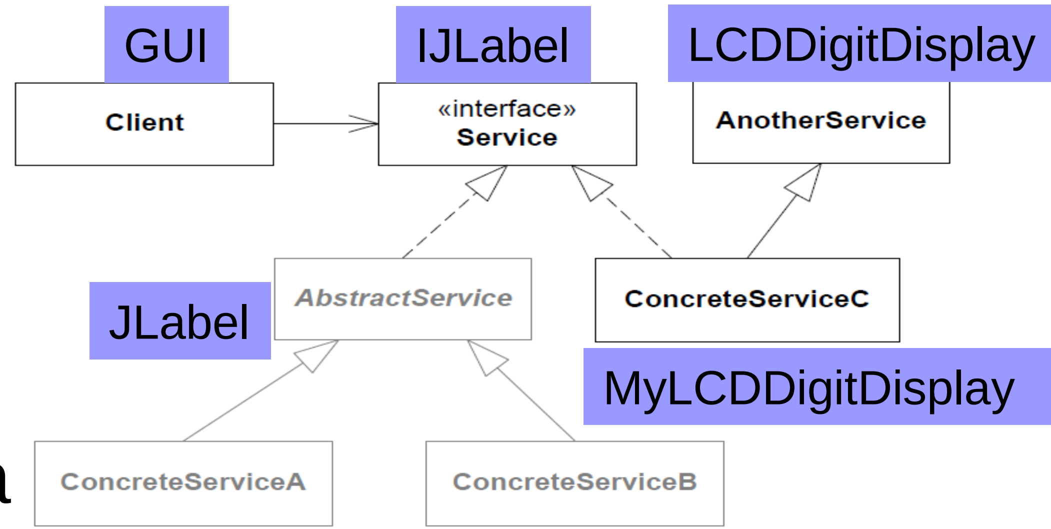
```
/** Update the digital display with whatever the
    pay station domain shows */
private void updateDisplay() {
  String prefixedZeros =
    String.format("%4d", payStation.readDisplay() );
  display.setText( prefixedZeros );
}
```

Morale

- It would have been easy to make the code completely identical, and thus support full reuse, in which I simply configure PayStationGUI with the proper ’text panel’ to use.
- ***But I cannot!***
  - Because LCDDigitDisplay does not inherit JLabel!!!
- Thus instead of *dependency injection* and *change by addition* I get
- ***Change by modification***
  - I have to start my editor just to change one declaration!
  - I can never get a framework out of this!

Could have been solved…

- If JLabel was an *interface* instead!
  - Interface "IJLabel"
  - setText(String s);



- Then there would be no hard coupling to a specific inheritance hierarchy.

Interfaces allow fine-grained behavioral abstractions

- Clients can be *very* specific about the exact responsibility it requires from its service provider – *Role interfaces*

SOLID : I = Interface Segregation

- Example:
  - Collections.sort(List<T> list)

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

  - can sort a list of objects of *any* type, T, if each object implements the interface Comparable<? super T>
  - i.e. must implement method 'int compareTo(T o)'.

- **Low coupling – no irrelevant method dependency!**

Interfaces better express roles

- Interfaces express *specific responsibilities* whereas classes express *concepts*. Concepts usually include more responsibilities and they become broader!

```
public interface Drawing extends
        FigureCollection, SelectionHandler,
        FigureChangeListener, DrawingChangeListenerHandler {
```

- Small, very well defined, roles are easier to reuse as you do not get all the “stuff you do not need...”

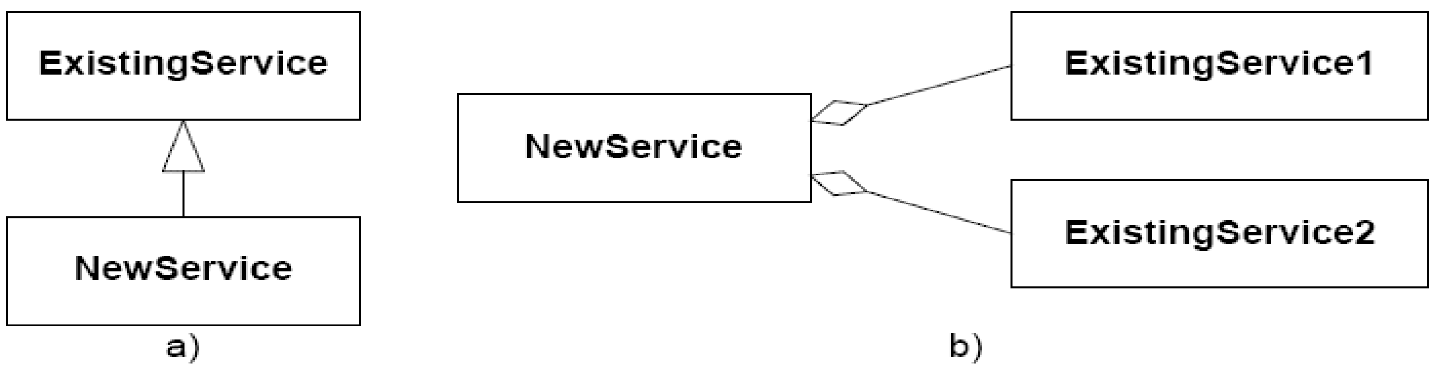
```
public class CompositionalDrawing implements Drawing {
        public CompositionalDrawing() {
          selectionHandler = new StandardSelectionHandler();
          listenerHandler = new StandardDrawingChangeListenerHandler();
          figureChangeListener = new ForwardingFigureChangeHandler( source: this, listenerHandler);
          figureCollection = new StandardFigureCollection(figureChangeListener);
        }
```

Second Principle

GoF's 2nd principle

- *Favor object composition over class inheritance*
- What this statement says is that there are basically *two* ways to reuse code in OO!

And the compositional one should be **favored!**



Benefits of class inheritance

- **Class inheritance**
  - You get the “whole packet” and “tweak a bit” by overriding a single or few methods
    - Fast and easy (very little typing!)
    - Explicit in the code, supported by language
      - (you can directly write “extends”)
- But...

Encapsulation

- *“inheritance breaks encapsulation”*

- Snyder (1986)

Why?

- No encapsulation because
  - Subclass can access every…
    - instance variable/property
    - data structure
    - Method
  - … of any superclass (except those declared private)

- Thus a subclass and superclass are ***tightly coupled***
  - You cannot change the root class' data structure without refactoring every subclass in the complete hierarchy ☹

Only add responsibilities, never remove

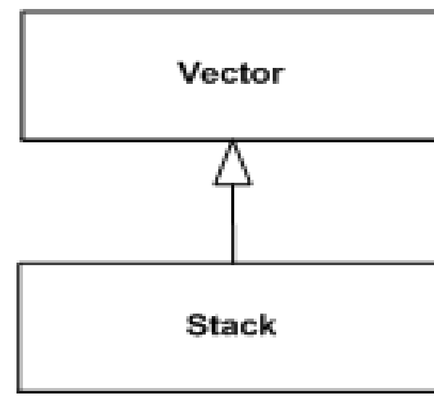
- You buy the full package!
  - All methods, all data structures
  - Even those that are irrelevant or down right wrong!

Example

- Vector<E>   (= an ArrayList ‘almost’)
  - void add(int index, E element)

```
public class Stack<E>
extends Vector<E>
```

- Stack<E> extends Vector<E>
  - E pop()
  - void push(E item)



Argue why this is a design with many liabilities?
How can you rewrite it elegantly using composition?

AARHUS UNIVERSITET

Rewriting to Composition

- Class 'Stack has-a Vector', instead of 'Stack is-a Vector'
  - Much better design!
    - Stack does not have any Vector/List methods, only push() and pop()

```
import java.util.*;

public class StackHasAVector {
  public static void main(String[] args) {
    System.out.println("== Stack has-a vector ==");
    Stack s = new Stack();
    s.push("Item 1");
    s.push("Item 2");
    s.push("Item 3");
    System.out.println(" Popped value (1) = " + s.pop());
    System.out.println(" Popped value (2) = " + s.pop());
  }
}

class Stack {
  // has-a vector (here ArrayList)
  private List<String> contents = new ArrayList<String>();

  public void push(String item) {
    contents.add(0, item);
  }

  public String pop() {
    return contents.remove(0);
  }
}
```

```
csdev@small22:~/proj/frsproject/stack-has-a-vector$ java StackHasAVector
== Stack has-a vector ==
 Popped value (1) = Item 3
 Popped value (2) = Item 2
```

Compile time binding

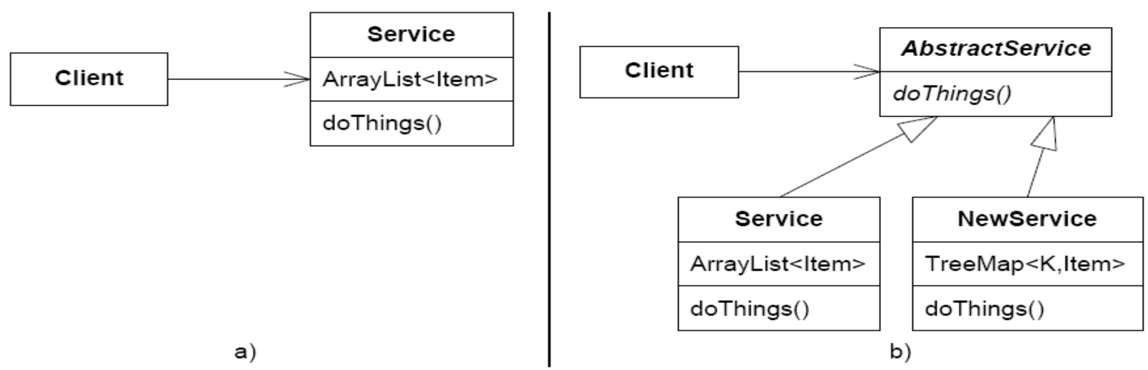
The only way to change behavior in the future (tweak a bit more) is through the *edit-compile-debug-debug-debug-debug* cycle

Any implementing class of List<String> can be substituted here (by Dependency Injection), thus no hard coupling between Stack and “Vector”

```java
import java.util.*;

public class StackHasAVector {
  public static void main(String[] args) {
    System.out.println("== Stack has-a vector ==");
    Stack s = new Stack();
    s.push("Item 1");
    s.push("Item 2");
    s.push("Item 3");
    System.out.println(" Popped value (1) = " + s.pop());
    System.out.println(" Popped value (2) = " + s.pop());
  }
}

class Stack {
  // has-a vector (here ArrayList)
  private List<String> contents = new ArrayList<String>();

  public void push(String item) {
    contents.add(0, item);
  }

  public String pop() {
    return contents.remove(0);
  }
}
```

Recurring modifications

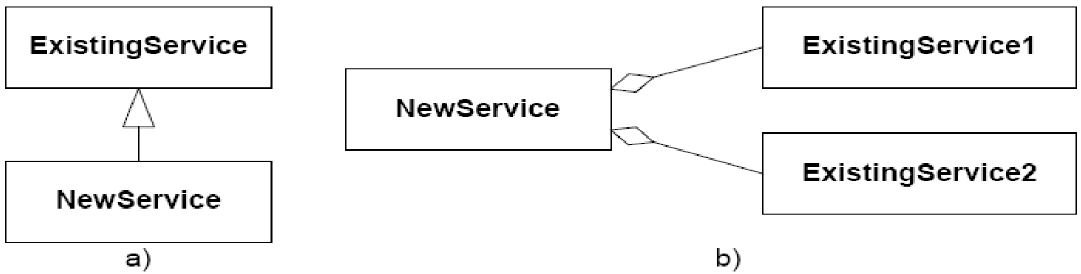
- Constantly bubbling of behavior up into the root class in a hierarchy
  - Review the analysis in the State pattern chapter
- Another example
  - Nice service based upon ArrayList
    - Now – want better performance in new variant



  - *All three classes modified ☹*

Separate Testing

- Often, small and well focused abstractions are easier to test than large classes



  - a) Only *integration testing* possible (NewS. + ExistS.)
  - b) Allows *unit testing* of ‘ExistingService1+2’,
    and often *unit testing* of NewService, by replacing collaborators with Test Stubs ala *StubService1 and StubService2*

Increase possibility of reuse

- Smaller implementations are easier to reuse
- Example from MiniDraw

Drawing

- Be a collection of figures.
- Allow figures to be added and removed.
- Maintain a temporary, possibly empty, subset of all figures, called a *selection*.

  – Sub responsibility

```
// === Delegate to the figure collection
Henrik Bærbak Christensen
@Override
public Figure add(Figure figure) { return figureCollection.add(figure); }

Henrik Bærbak Christensen
@Override
public Figure remove(Figure figure) { return figureCollection.remove(figure); }
```

- Allow compositional reuse of FigureCollection in ***all present and future impl. of Drawing!***

Liabilities

- Increased number of abstractions and objects ☹

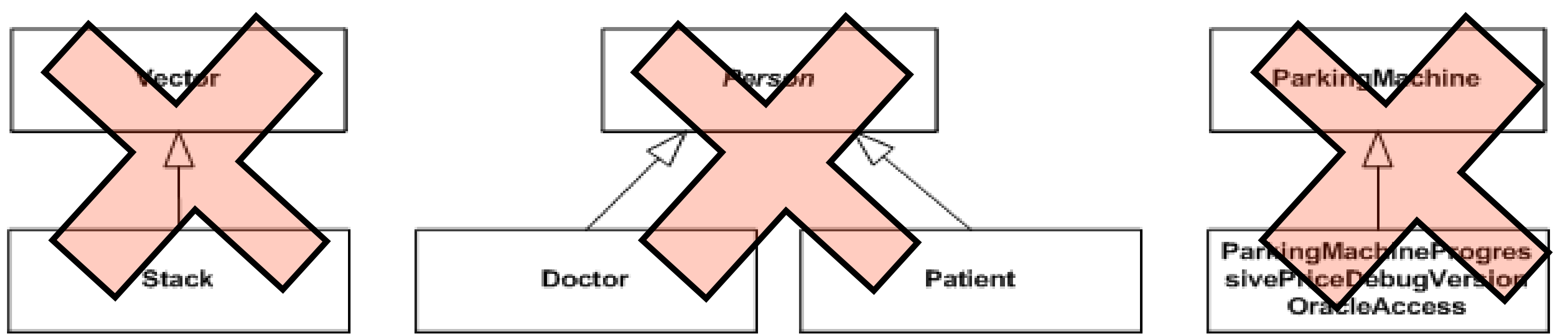
```
public CompositionalDrawing() {
  selectionHandler = new StandardSelectionHandler();
  listenerHandler = new StandardDrawingChangeListenerHandler();
  figureChangeListener = new ForwardingFigureChangeHandler( source: this, listenerHandler);
  figureCollection = new StandardFigureCollection(figureChangeListener);
}
```

- Delegation requires more boiler-plate code ☹

```
// === Delegate to the figure collection
Henrik Bærbak Christensen
@Override
public Figure add(Figure figure) { return figureCollection.add(figure); }

Henrik Bærbak Christensen
@Override
public Figure remove(Figure figure) { return figureCollection.remove(figure); }
```

(what *is* he saying???)

- Inheritance is an interesting construct, but
  - It often leads to lesser designs ☹
- It does not elegantly handle
  - ad hoc reuse
  - modelling roles
  - variance of behavior



When to use Inheritance?

- My rule of thumb
  - Iff there is *behavioral differences* between subclasses
    - Not just parameters and constants; it must be different algorithms
  - Iff you are absolutely sure there will be only one dimension of variability and a shallow inheritance tree…
- One example where I use it is in the HotStone GUI

```
public abstract class HotStoneActorFigure extends CompositeFigure
        implements HotStoneFigure {
```

```
public class CardFigure extends HotStoneActorFigure
```

```
public class MinionFigure extends HotStoneActorFigure {
```

"Same" except how the Gfx is rendered and updated…

Third Principle

GoF's 3rd principle

- *Consider what should be variable in your design*
  - [GoF §1.8, p.29]

- Another way of expressing the 3rd principle:
  - *Encapsulate the behavior that varies*

Analysis

- This statement is closely linked to the shorter
    - *Change by addition, not by modification*
- That is – you identify
  - the design/code that should remain *stable*
  - the design/code that may vary
- and use techniques that ensure that the stable part – well – remain stable

- These techniques are 1st and 2nd principle
  - most of the time ☺

The Principles In Action

Principles in action

- Applying the principles lead to basically the same structure of most patterns:
  - New requirement to our client code

Principles in action

- Applying the principles lead to basically the same structure of most patterns:

- ③ Consider what should be variable

Principles in action

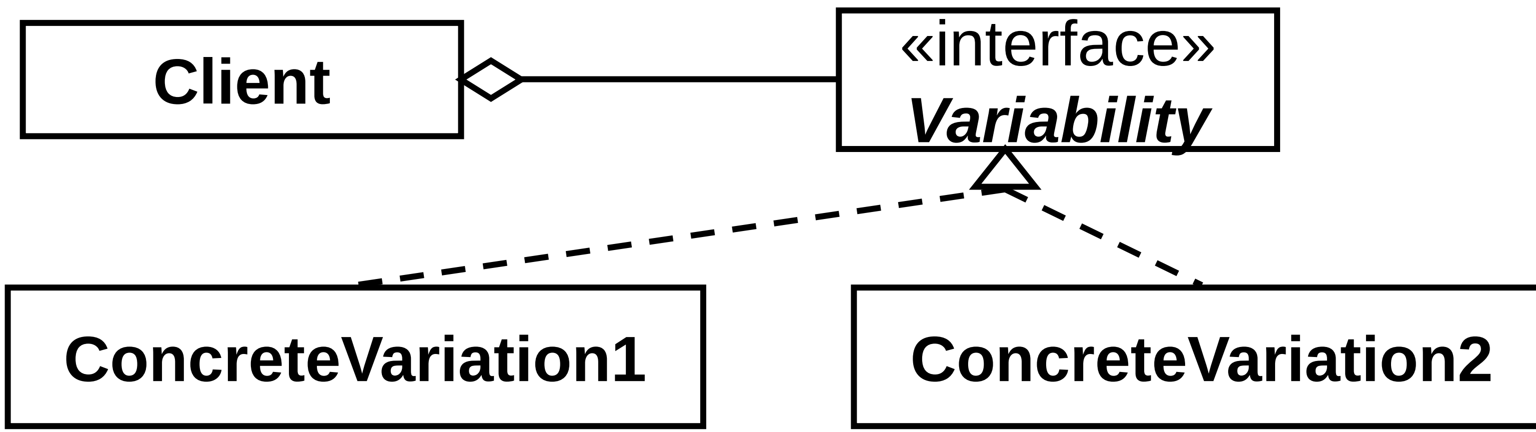
- Applying the principles lead to basically the same structure of most patterns:

- ① Program to an interface

Principles in action

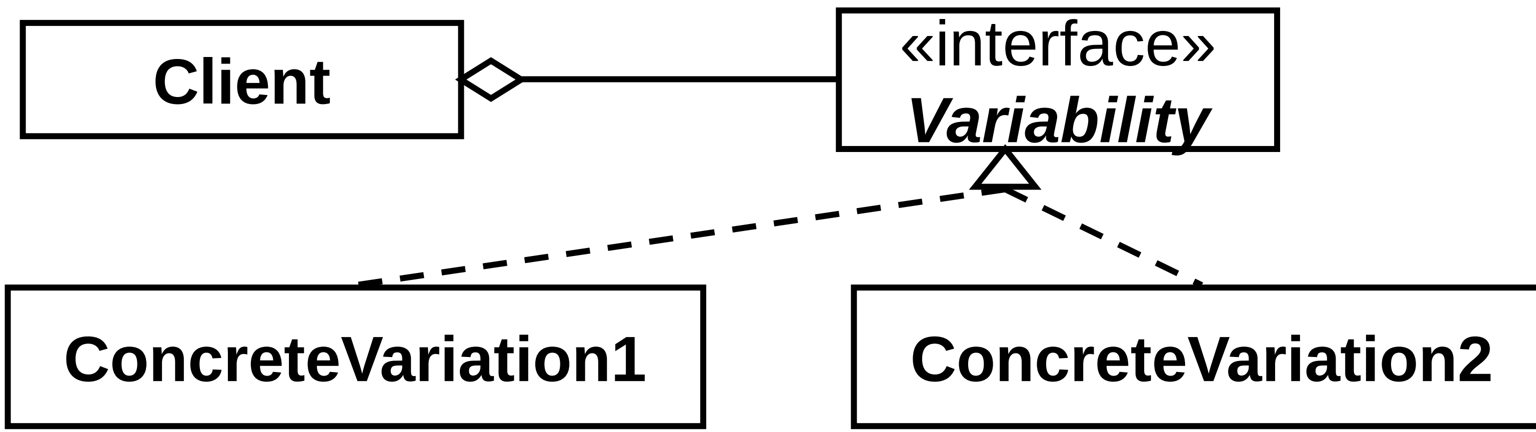
- Applying the principles lead to basically the same structure of most patterns:

- ② Favor object composition



And that is why…

- … most patterns follows this structure exactly
  - They encapsulate variability and favor composition



Summary

- ③ We *identified some behaviour* that was *likely to change*…
- ① We stated a *well defined responsibility* that covers this behaviour and expressed it in an *interface*
- ② Instead of performing behaviour ourselves we *delegated* to an object implementing the interface

- ③ *Consider what should be variable in your design*
- ① *Program to an interface, not an implementation*
- ② *Favor object composition over class inheritance*

SOLID

- A more well known set of principles than③①②, but states more or less the same…

S The single-responsibility principle: "There should never be more than one reason for a class to change." That is, encapsulate behavior in well-defined and fine-grained roles; encapsulate what varies.

O The open–closed principle: "Software entities …should be open for extension, but closed for modification." That is, favor change by addition.

L The Liskov substitution principle: "Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it." That is, program to an interface.

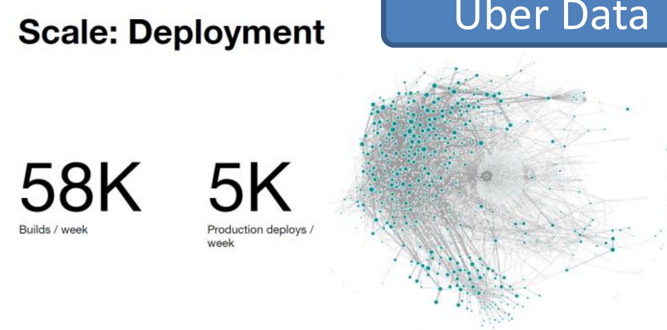
I The interface segregation principle: "Many client-specific interfaces are better than one general-purpose interface." That is, express behavior using fine-grained roles.

D The dependency inversion principle: "Depend upon abstractions, [not] concretions." That is, program to an interface, and favor object composition by dependency injection.

SOLID *is* Solid

- An architectural style for *large systems:* Microservices
  - Key architecture for Uber, Google, NetFlix, …
- Lots of tooling, lots of architectural tactics, lots of design doctrines to follow, but…

Uber Data



- At the core, it is..
  - *Design with high cohesion and low coupling*
  - *Design according to SOLID*
  - ***Program to an interface, favor object composition***