Compositional Design Principles

The “GoF” principles

Or

Principles of Flexible Design

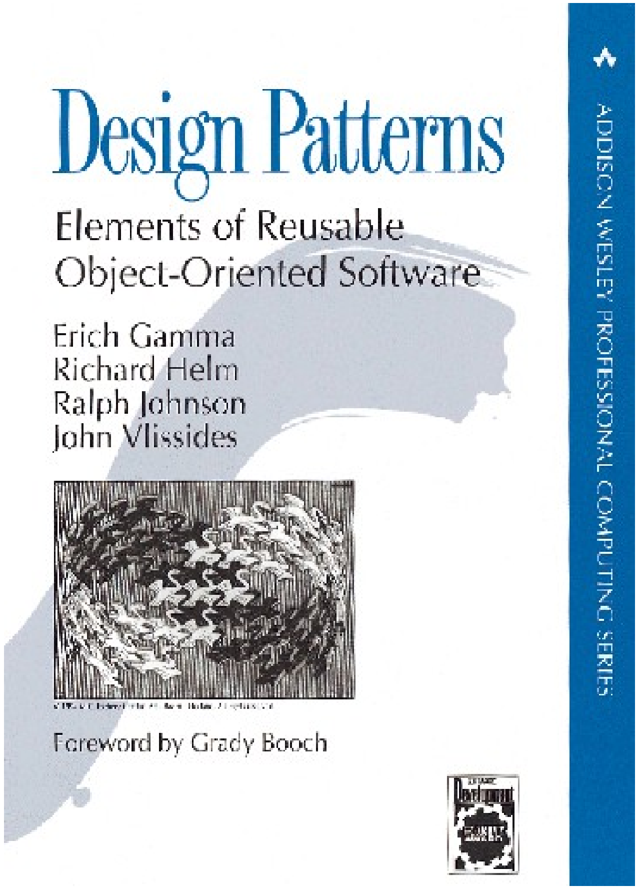
Gang of Four (GoF)

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.


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON WESLEY PROFESSIONAL COMPUTING SERIES

The most important chapter

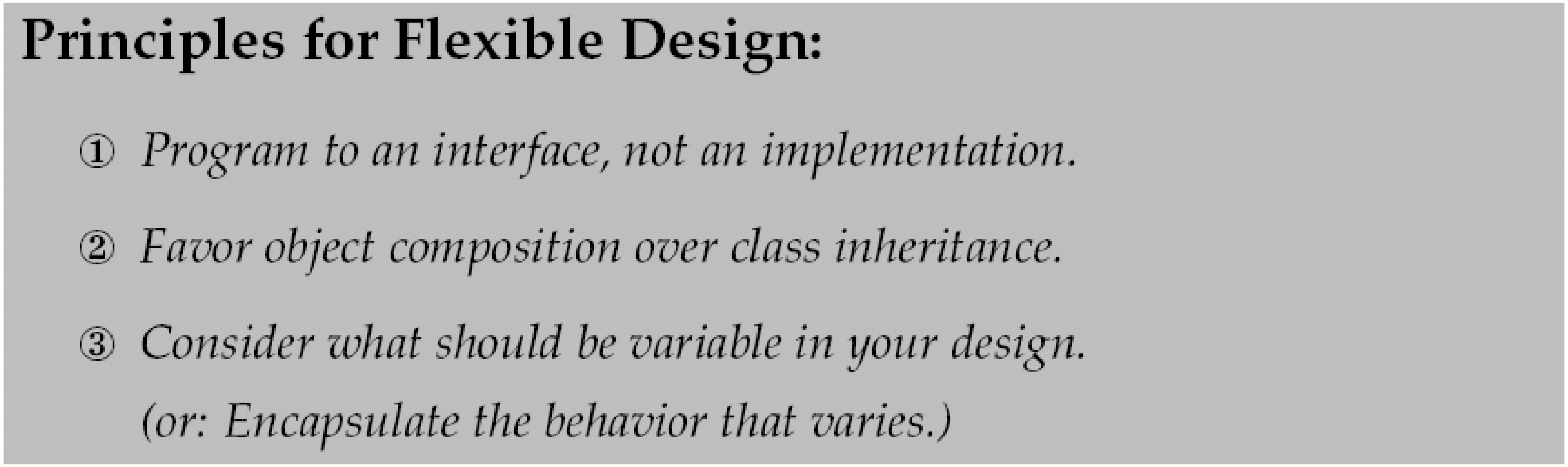
Section 1.6 of GoF has a section called:

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



Principles for Flexible Design:

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

② *Favor object composition over class inheritance.*

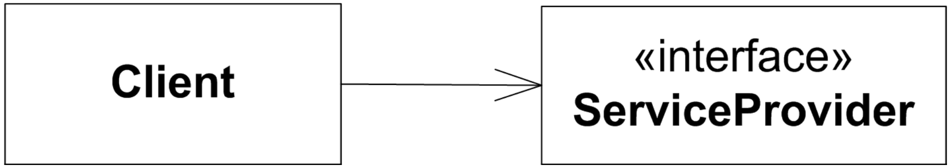
③ *Consider what should be variable in your design.*

(or: Encapsulate the behavior that varies.)

First Principle

GoF's 1st principle

Program to an interface, not an implementation



In other words

Assume only the contract

(the responsibilities)

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

First Principle

Program to an interface because

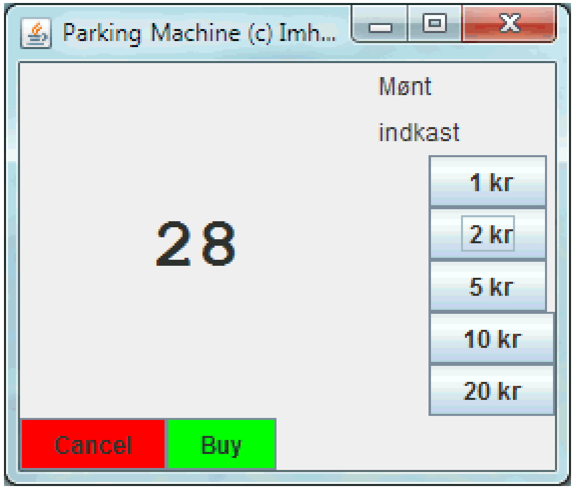
- You are *free* to use *any* service provider class!
- You do not delimit other developers for providing *their* service provider class!
- You avoid binding others to a particular inheritance hierarchy
  - Which you would do if you use (abstract) classes…

Example

Early pay station GUI used JPanel for visual output

```
public class ParkingMachineGUI extends JFrame {

    JLabel display;
    ParkingMachine parkingMachine;
```

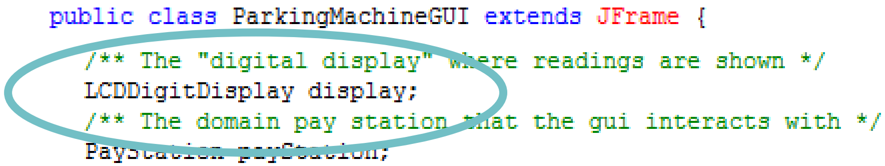


I only use method: ’setText’

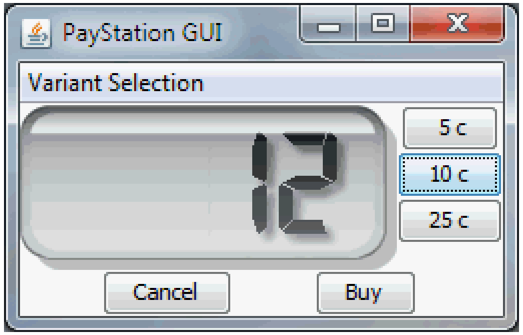
```
  public void updateDisplay() {
    display.setText( ""+parkingMachine.readDisplay() );
  }
```

Example

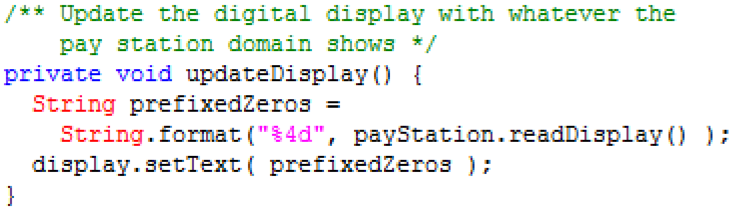
The I found SoftCollection's number display, got permission to use it, but...



```
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

I 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 JPanel!!!

Thus instead of *dependency injection* and *change by addition* I get

Change by modification

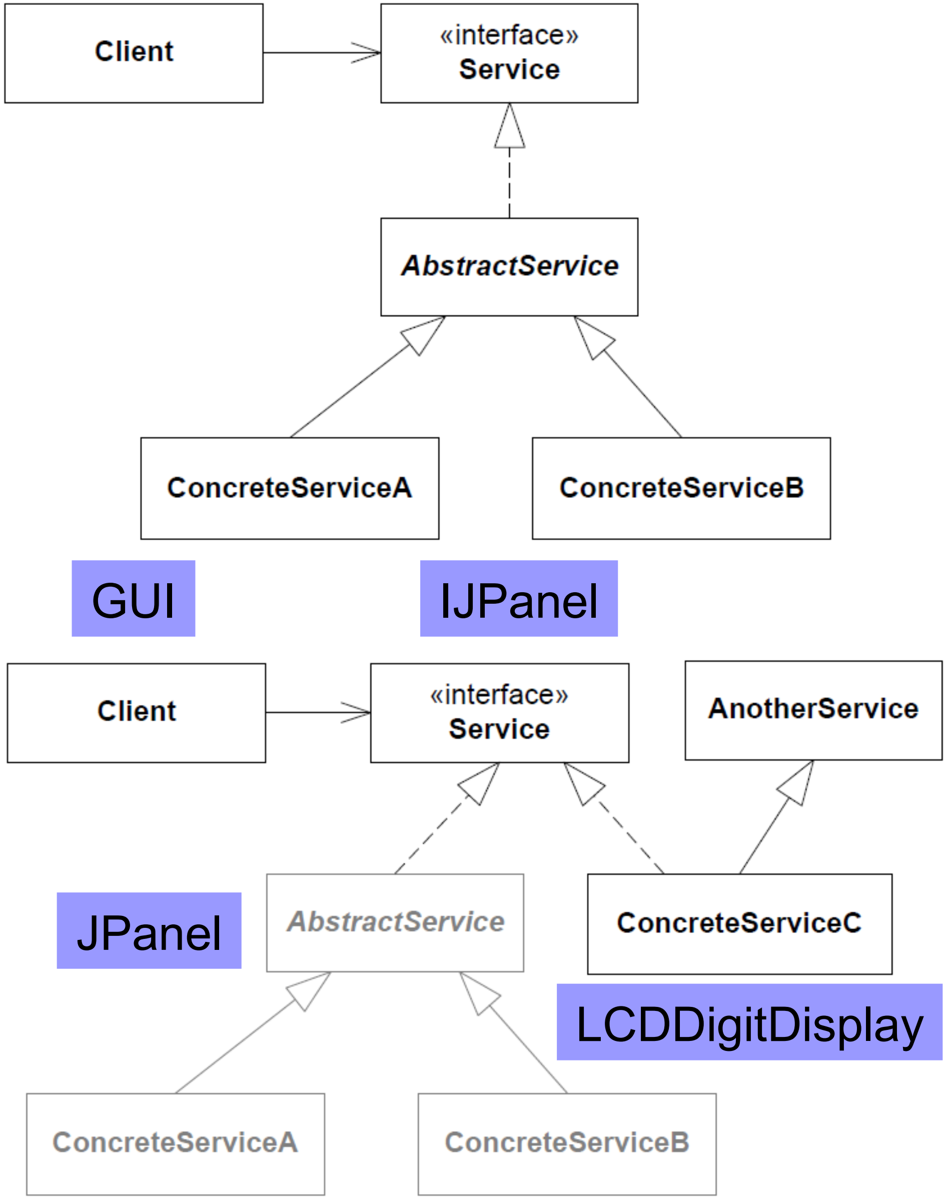
– I have to start eclipse just to change one declaration!

– I can never get a framework out of this!

If JPanel was an interface instead!

- setText(String s);

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



Interfaces allow fine-grained behavioural abstractions

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

Example:

- Collections.sort(List l)
- can sort a list of *any* type of object if each object implements the interface Comparable
- i.e. must implement method CompareTo(Object 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 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 StandardSelectionHandler implements SelectionHandler {...}
```

Umbrella responsibility

class Car extends Umbrella ?

class Umbrella extends Car ?

NONSENSE!

class Car implements UmbrellaRole

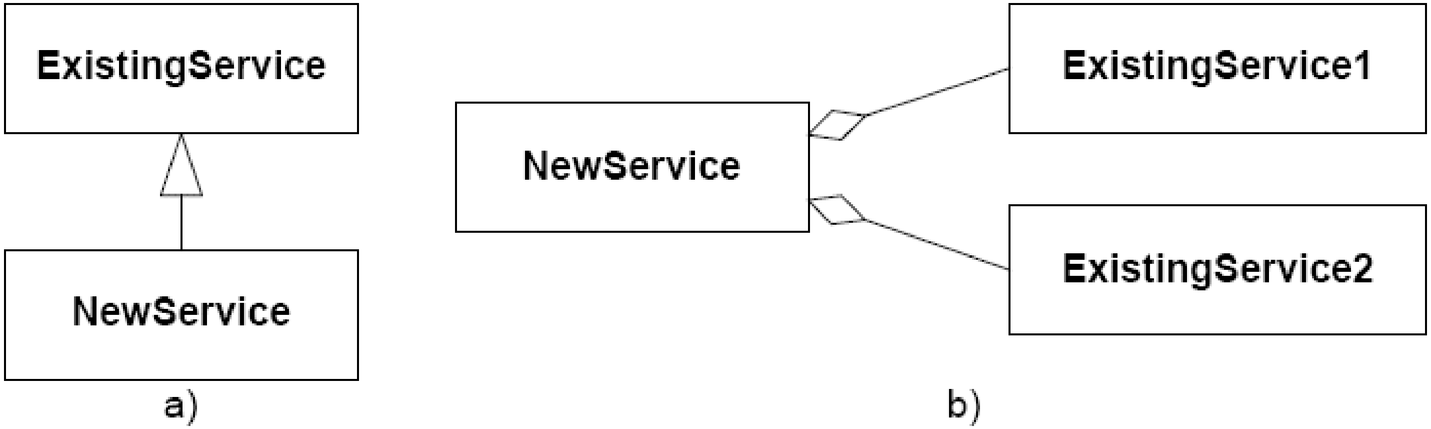
Sensible

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



Analysis

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 (Early Java)

- Java.util.Stack extends java.util.Vector (ArrayList)
  - A stack should only support *push* and *pop*
  - But it of course also support addAt(i) and remove(j)

Exercise:

- What is the proper compositional relation between a stack and an ArrayList?

Compile time binding

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

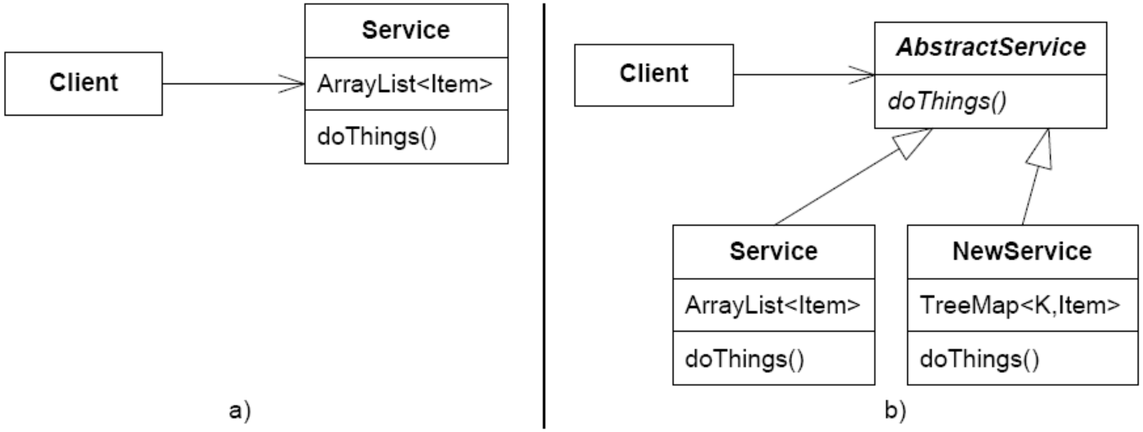
Recurring modifications

Constantly bubling of behaviour 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

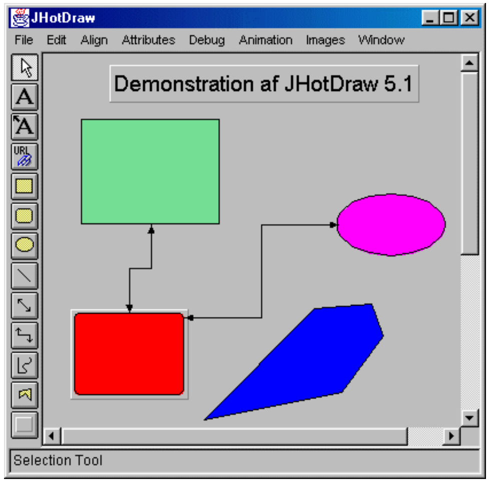
Often, small and well focused abstractions are easier to test than large classes

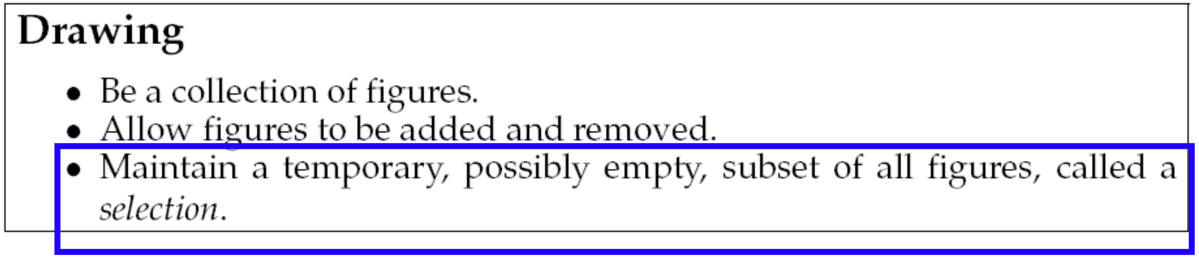
- However, often requires test stubs

Increase possibility of reuse

Smaller abstractions are easier to reuse

Example (from MiniDraw)

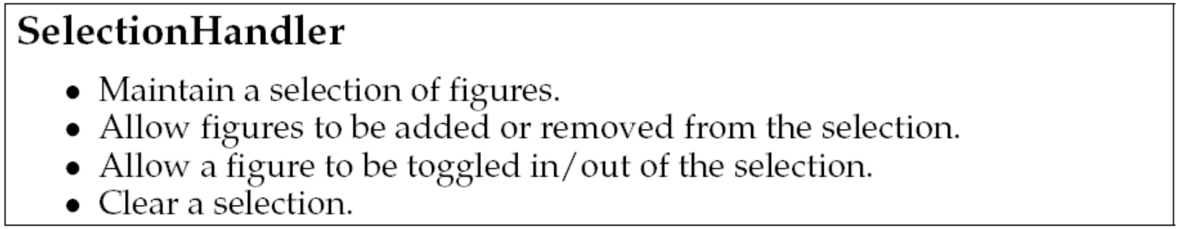




Drawing

- 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



SelectionHandler

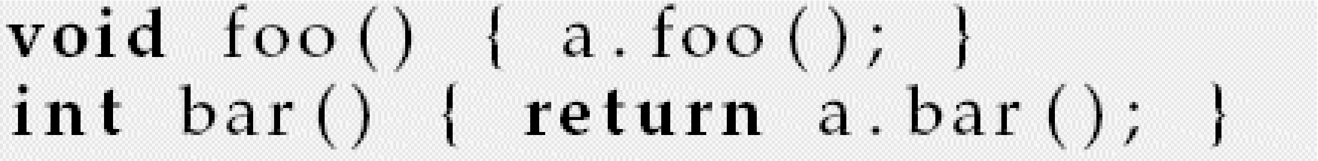
- Maintain a selection of figures.
- Allow figures to be added or removed from the selection.
- Allow a figure to be toggled in/out of the selection.
- Clear a selection.

Allow compositional reuse of selection handler in ***all present and future impl. of Drawing!***

Liabilities

Increased number of abstractions and objects ☹

Delegation requires more boiler-plate code ☹



```
void foo() { a.foo(); }
int bar() { return a.bar(); }
```

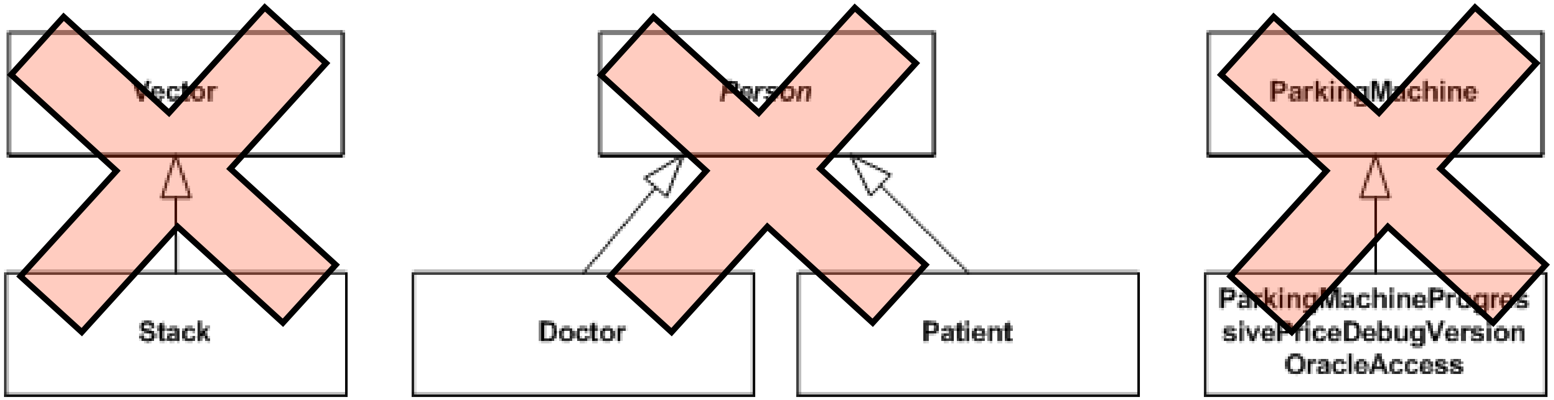
(what *is* he saying???)

Inheritance is OK but you must use it for what it handles really nice

- Modelling of concepts in a generalisation hierarchy

NOT for handling

- ad hoc reuse
- modelling roles
- variance of behaviour



Exercise

How does the three X designs on the former slide look if I apply the 2nd principle?

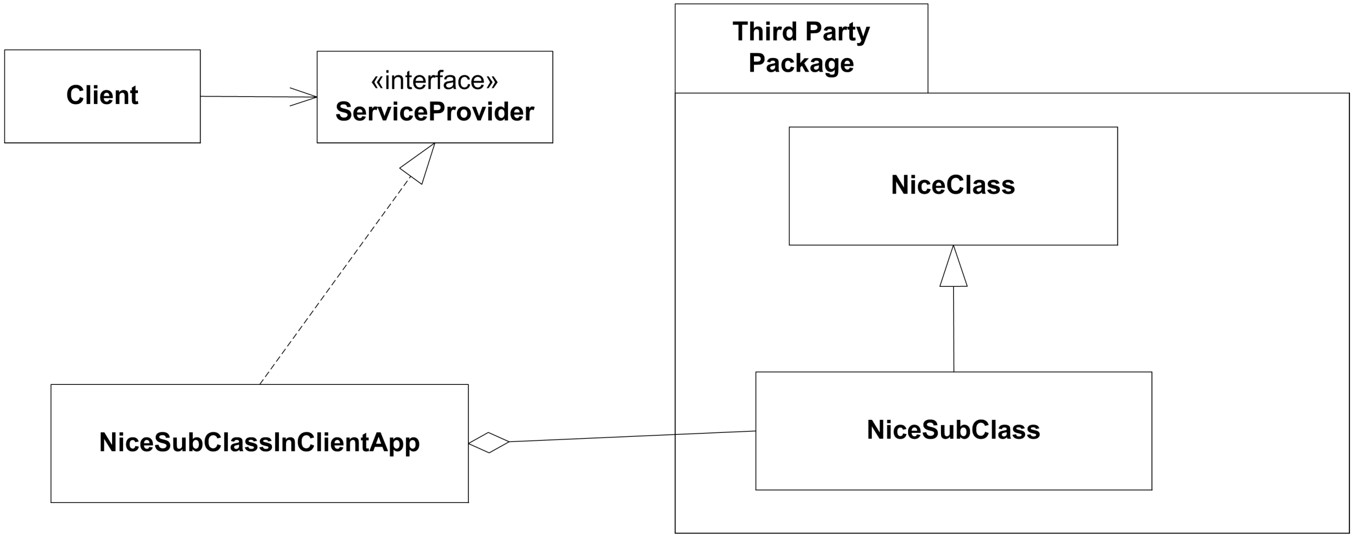
Exercise

Combining principle 1 and 2

- Combining behaviour from own subclass and third party class

It is also a design pattern!

- which one?



Third Principle

GoF's 3rd principle

Consider what should be variable in your design

[GoF §1.8, p.29]

This approach is the opposite of focusing on the causes of redesign. Instead of considering what might *force* a change to a design, consider what you want to be *able* to change without redesign. The focus here is on *encapsulating the concept that varies*, a theme of many design patterns.

Alternative formulation

Another way of expressing the 3rd principle:

Encapsulate the behaviour 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 ☺

Example

The pay station

- new price model???
  - maybe this will vary in the future
- new receipt types ???
  - maybe ...
- new display output ???
- testing often force detailed control...

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

Client

Principles in action

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

③ Consider what should be variable

Client
Variability

Principles in action

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

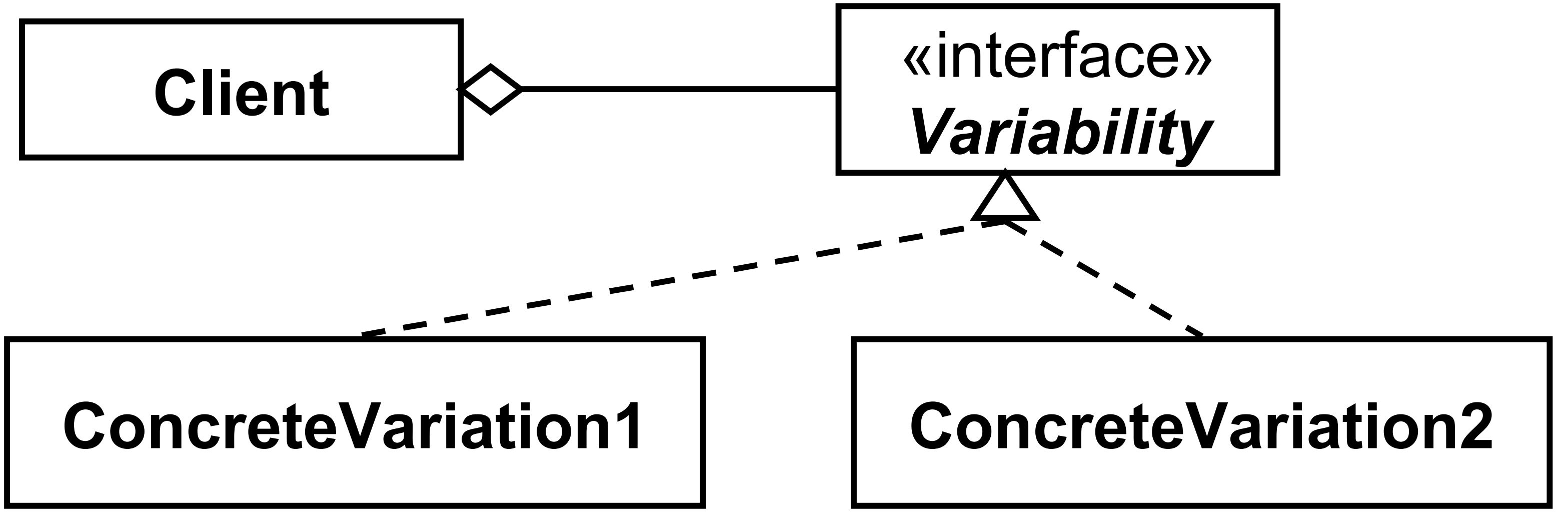
① Program to an interface

Client
«interface»
Variability

Principles in action

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

② Favour object 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*

Consideration

Beware – it is not a process to follow blindly

- Often the key point is principle 2: look over how you may **compose** the resulting behavior most reasonable

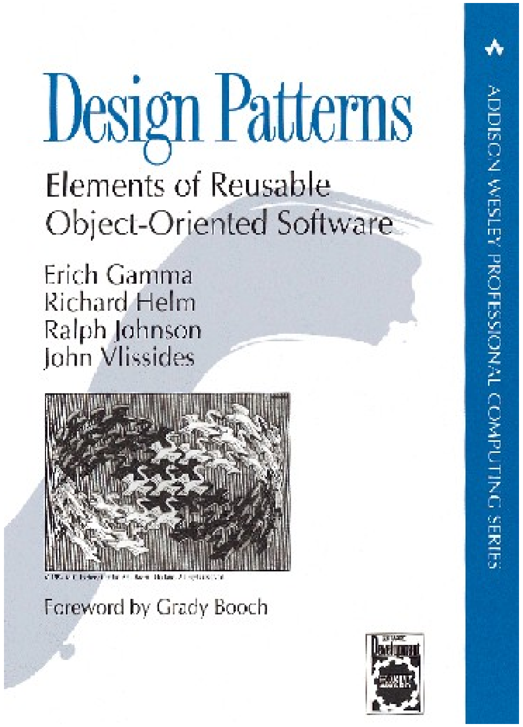
- Examples
  - Abstract Factory: We did not make a ReceiptIssuer specifically for receipts but found a more general concept
  - Decorator + Proxy: Sometimes the ‘encapsulation of what varies’ can be the whole abstraction and the solution relies on composition of ‘large’ objects.

Summary

GoF list 23 patterns – but

they also list three principles that are essential...

... elements of reusable object-oriented software...


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON WESLEY PROFESSIONAL COMPUTING SERIES