AARHUS UNIVERSITET

Software Engineering and Architecture

Compositional Design Principles

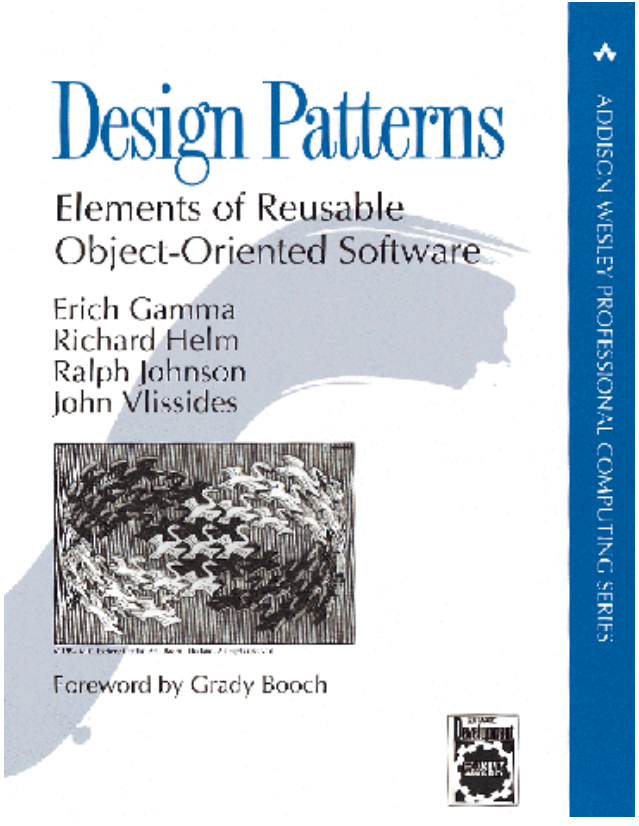
Gang of Four (GoF)

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.



AARHUS UNIVERSITET

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.)

AARHUS UNIVERSITET

As the 3-1-2 process

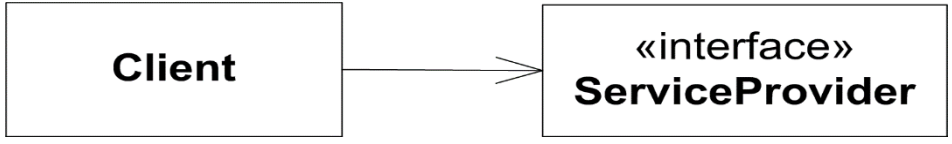
③ I identified some behavior that was likely to change…	=	③ *Consider what should be variable in your design.*
① I stated a well-defined responsibility that covers this behavior and expressed it in an interface…	=	① *Program to an interface, not an implementation.*
② Instead of implementing the behavior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

First Principle

AARHUS UNIVERSITET

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 behaviour

AARHUS UNIVERSITET

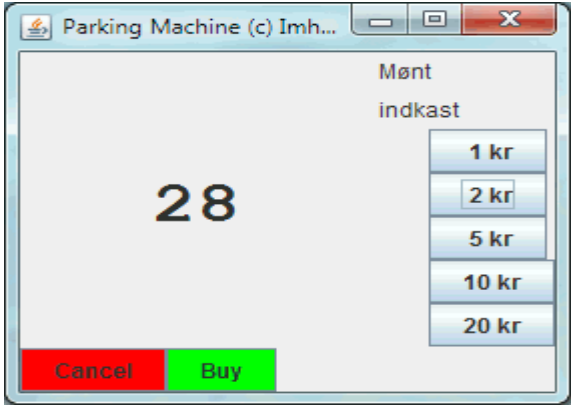
First Principle

- *Program to an interface* because
  - You only collaborate with the **role** – not an individual object
  - You are *free* to use *any* service provider class!
  - 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;
```



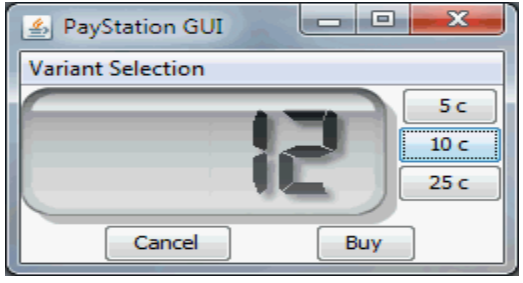
- I only use method: ’setText’

```
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

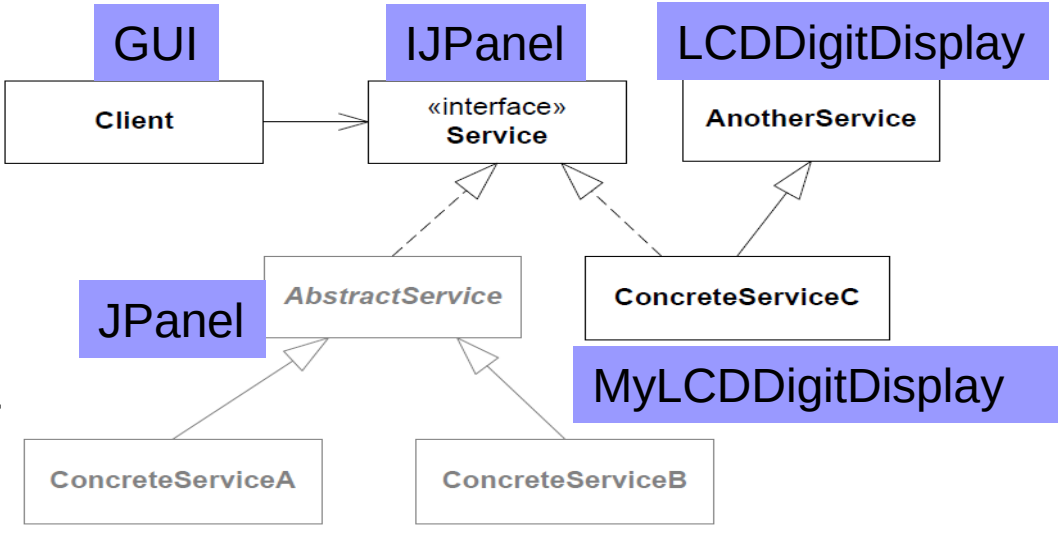
- It would have been easy to make the code completely identical, and thus support full reuse, in which I simply configure PayStationGUI with the proper ’text panel’ to use.
- ***But I cannot!***
  - Because LCDDigitDisplay does not inherit JPanel!!!
- Thus instead of *dependency injection* and *change 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!

AARHUS UNIVERSITET

Could have been solved…

- If JPanel was an *interface* instead!
  - setText(String s);

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



AARHUS UNIVERSITET

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!**

AARHUS UNIVERSITET

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 {...}
```

AARHUS UNIVERSITET

Umbrella responsibility

- class Car extends Umbrella ?
- class Umbrella extends Car ?

NONSENSE!

- class Car implements UmbrellaRole

Sensible

Second Principle

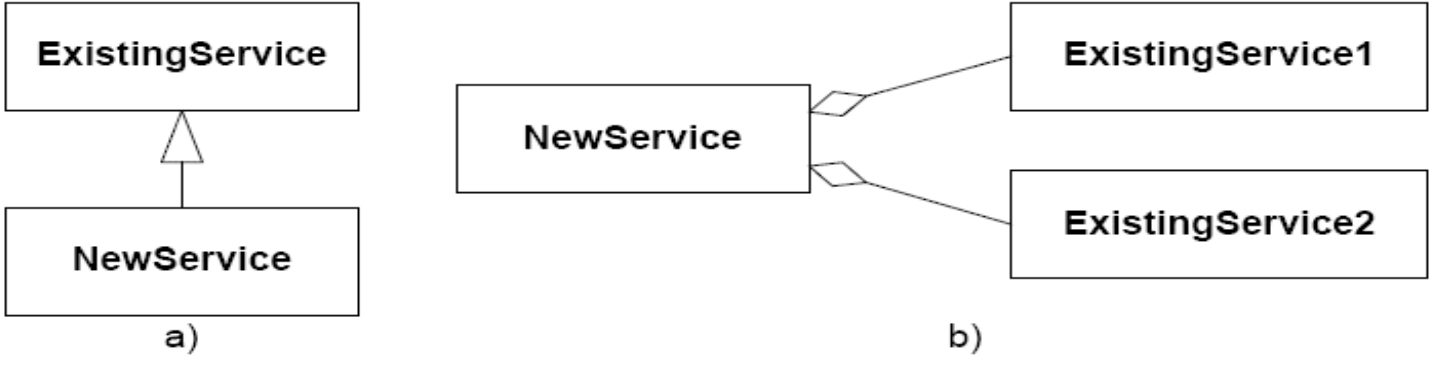
AARHUS UNIVERSITET

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!**



AARHUS UNIVERSITET

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 ☹

AARHUS UNIVERSITET

Only add responsibilities, never remove

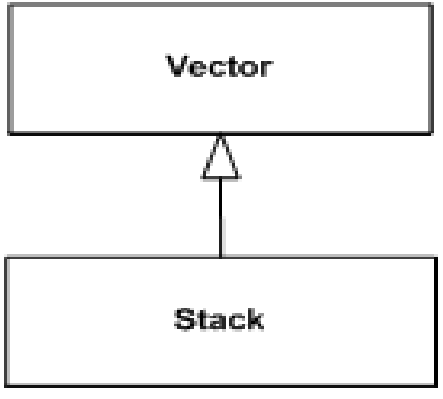
- You buy the full package!
  - All methods, all data structures
  - Even those that are irrelevant or down right wrong!

Example

- Java utility library version 1.1

- Vector (= modern ArrayList)
  - void add(int index, E element)

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



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

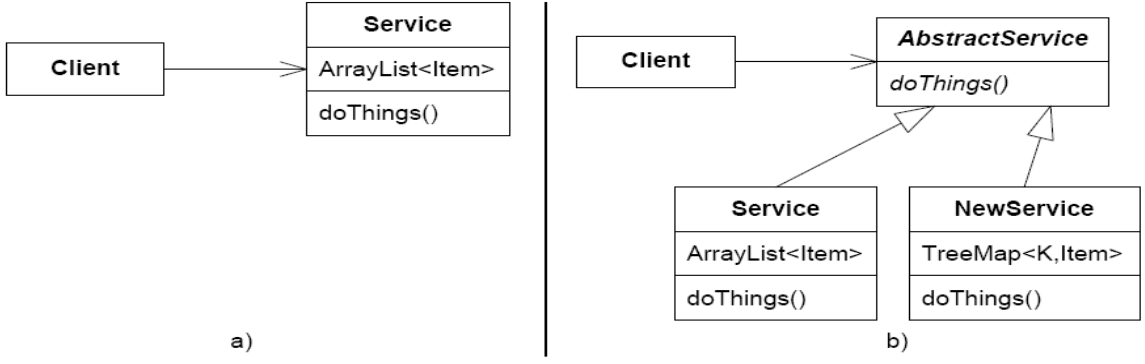
AARHUS UNIVERSITET

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

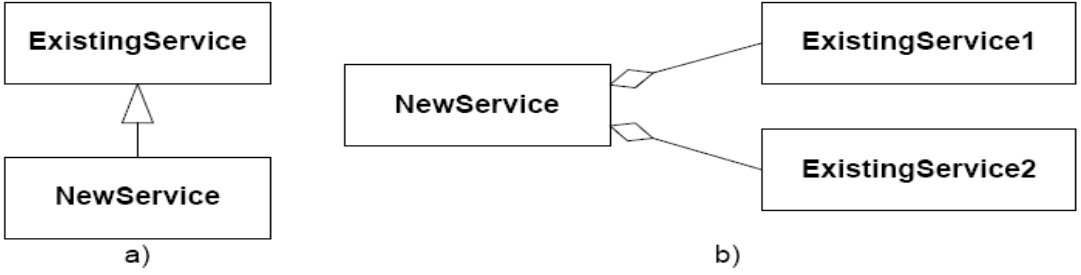
Recurring modifications

- 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



  - 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

Increase possibility of reuse

- Smaller abstractions are easier to reuse
- Example (from MiniDraw)

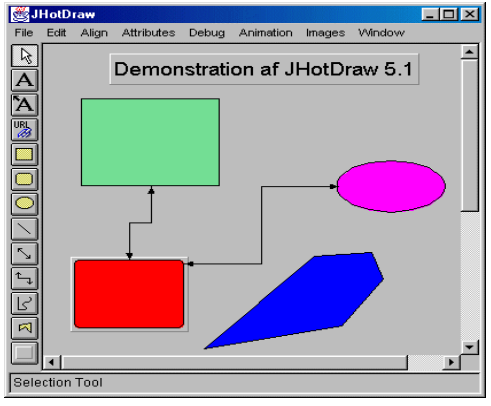
Drawing

- 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 ☹

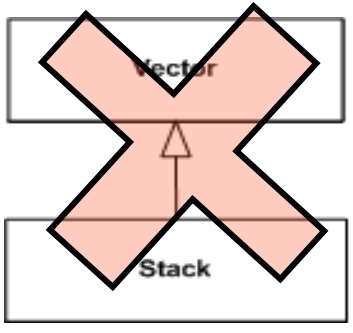
- Delegation requires more boiler-plate code ☹

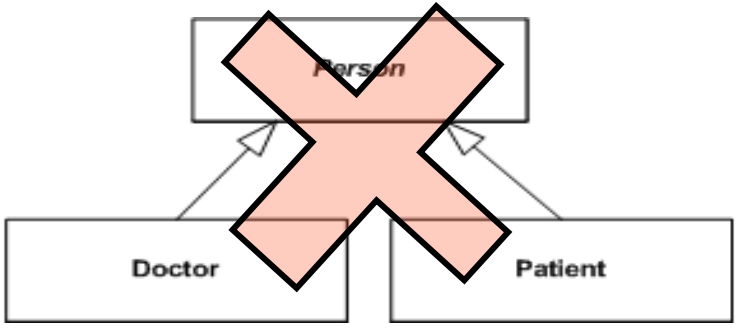
```
void foo() { a.foo(); }
int bar() { return a.bar(); }
```

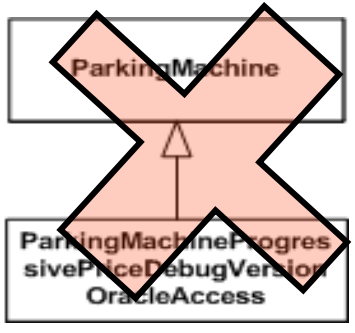
AARHUS UNIVERSITET

(what *is* he saying???)

- Inheritance is an interesting construct, but
  - I haven't seen any good examples ☹
- It does not elegantly handle
  - ad hoc reuse
  - modelling roles
  - variance of behaviour







Third Principle

AARHUS UNIVERSITET

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 behaviour that varies*

AARHUS UNIVERSITET

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

AARHUS UNIVERSITET

Principles in action

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

AARHUS UNIVERSITET

Principles in action

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

- ③ Consider what should be variable

Client

Variability

AARHUS UNIVERSITET

Principles in action

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

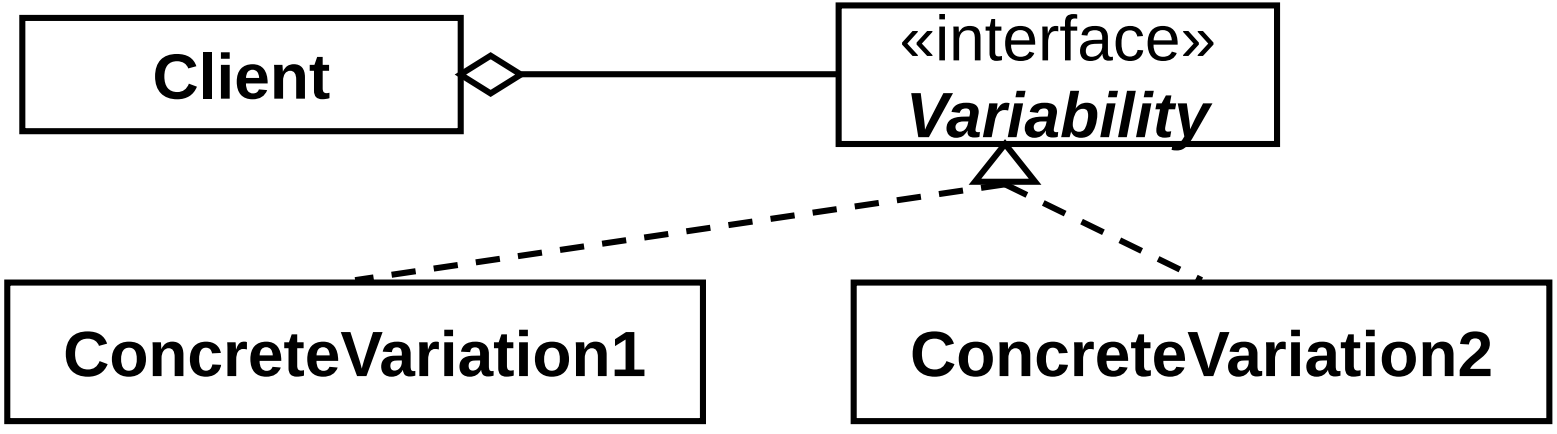
- ① Program to an interface

AARHUS UNIVERSITET

Principles in action

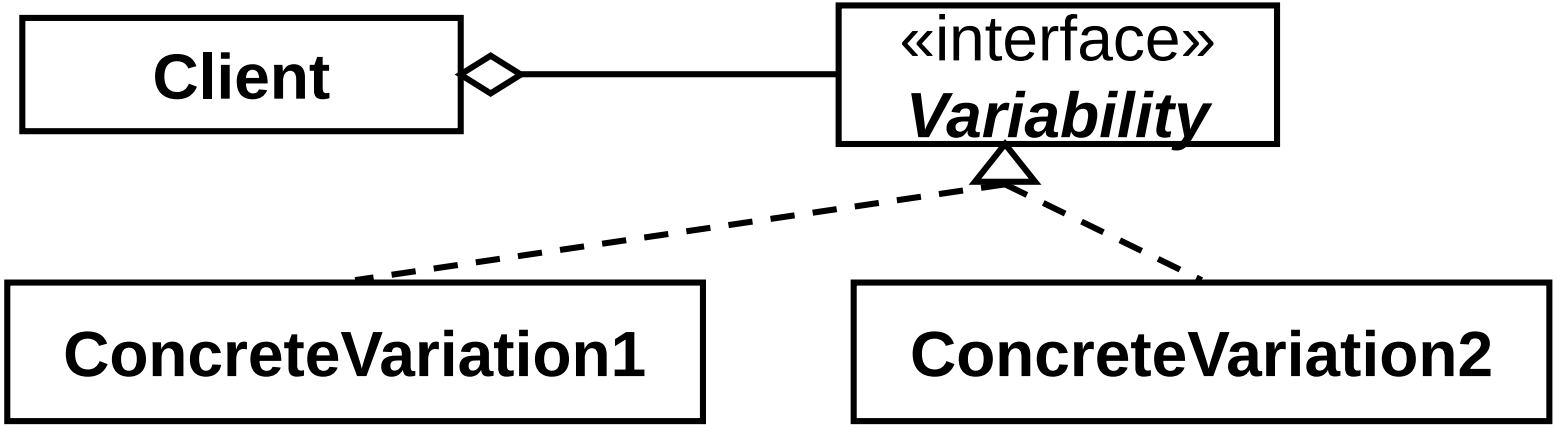
- Applying the principles lead to basically the same structure of most patterns:

- ② Favour object composition



And that is why…

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



AARHUS UNIVERSITET

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*

AARHUS UNIVERSITET

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...*