AARHUS UNIVERSITET

Deriving Strategy Pattern

... derived from the principles

Last – Alphatown county

Customer – *Alphatown* county:

The pay station must:

- accept coins for payment
- show time bought
- print parking time receipts
- US: 2 minutes cost 5 cent
- handle buy and cancel
- maintenance (empty it)

The nightmare: Success!

The Alphatown county is very satisfied!

Success is terrible!

It means:

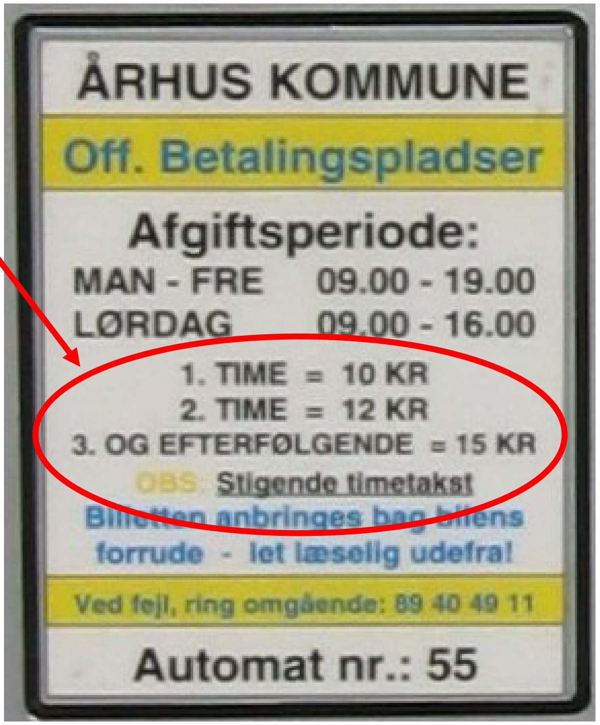
New requirements, add-ons, special cases and "wouldn't it be nice if..."

So – our parking machine software is now required by the Betatown county – but with a twist ☹

New requirement

Betatown: "New **progressive** price model"

1. First hour: \$1.50 (5 cent gives 2 minutes)
2. Second hour: \$2.00 (5 cent gives 1.5 minutes)
3. Third and following hours: \$3.00 per hour (5 cent gives 1 minute)



ÅRHUS KOMMUNE

Off. Betalingspladser

Afgiftsperiode:

MAN - FRE 09.00 - 19.00

LØRDAG 09.00 - 16.00

1. TIME = 10 KR

2. TIME = 12 KR

3. OG EFTERFØLGENDE = 15 KR

OBS: Stigende timetakst

Billetten anbringes bag bilens forrude - let læselig udefra!

Ved fejl, ring omgående: 89 40 49 11

Automat nr.: 55

Maybe we will see future changes in pricing models ???

How can we handle these two products?

The present code

This is the spot where things may change: **variability point**

```
public void addPayment( int coinValue ) throws IllegalCoinException {
  switch ( coinValue ) {
  case 5:
  case 10:
  case 25: break;
  default:
    throw new IllegalCoinException("Invalid coin: "+coinValue+" cent.");
  }
  insertedSoFar += coinValue;
  timeBought = insertedSoFar / 5 * 2;
}
```

Exercise !

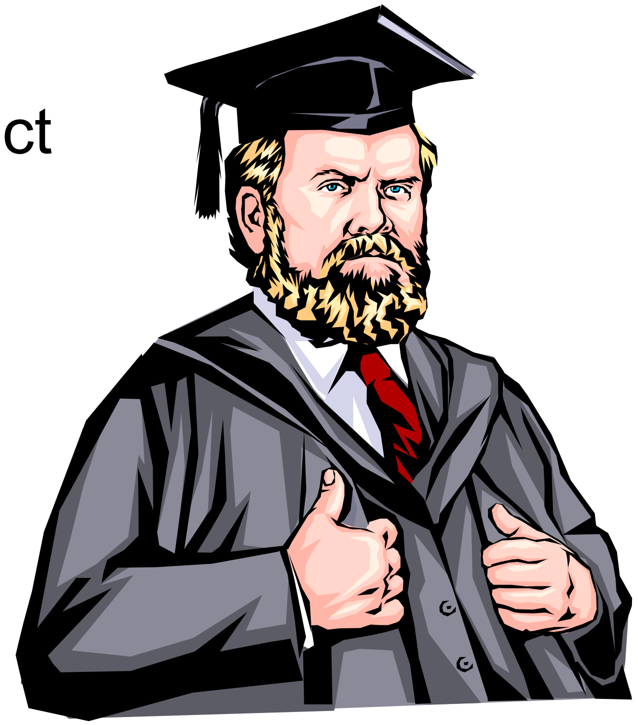
Propose some models to handle this.

Consider:

- Most of the code is the same in the two products
- What about real success? 20 product variants?

Focus:

- Sketch several models, not just an "optimal" one.
- Read the book? Find a fifth model!



Analysis

Model 1:

- Make a copy of the source tree
- Throw away some code, add some new code
- Throw away unit tests, add new unit tests

Model 2:

- Parameterization: Throw in some ‘if’-statements

Model 3:

- Polymorphic proposal: Variation through inheritance

Model 4:

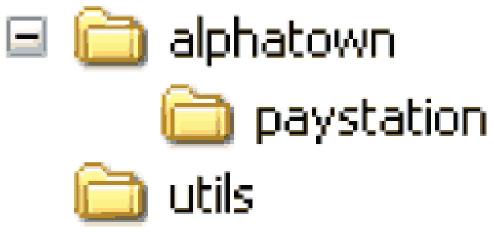
- Compositional proposal: Factor out rate model responsibility

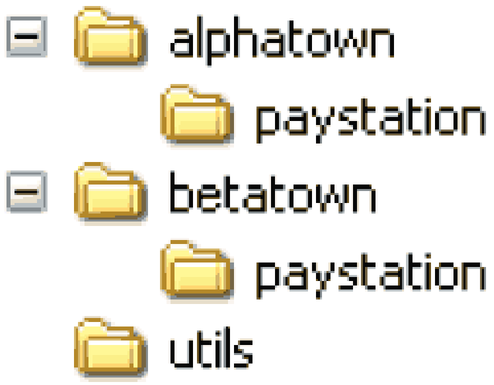
Model 1: Source Code Copy

Widely used: Next generation software

Model 1: Source tree copying

Idea: Deep copy production code source tree





Code the new variant by replacing the code at the variability point.

Benefits

Benefits:

- It is simple!
  - no special skill set required in developer team
  - easy to explain idea to new developers
- It is fast!
  - < 5 minutes?
- It provides perfect variant decoupling
  - defects introduced in variant 2 does **not** reduce reliability of variant 1
  - easy to distinguish variants (consult folder hierarchy)

Liabilities

Liabilities:

Multiple maintenance problem 💣

– Changes in common code must be propagated to all copies

– Usually manual process (or tedious SCM operation)

Example:

– 4 pay station variants (different rate policies)

– request: pay station keeps track of earning ☹

Experience: Variants drift apart, becoming different products instead of variants...

AARHUS UNIVERSITET

Liabilities

If you have many copies you easily get mixed up

– thus the benefit of easily identifying which variant you are working on is actually not true

Example:

– Fixing the same bug in 5 nearly identical SAVOS production code bases at the same time ☹

Model 2: Parametric Solution

Widely used: Debug variant

Parametric

Idea:

- It is only a single “behavioural unit” in the addPayment method that varies

```
[...]
timeBought = insertedSoFar * 2 / 5;
```

- I can simply make a conditional statement there

A) Introduce a parameter (Which town)

B) Switch on the parameter each time town specific behaviour is needed.

AARHUS UNIVERSITET

Code View

```
public class PayStationImpl implements PayStation {
  [...]
  public enum Town { ALPHATOWN, BETATOWN }
  private Town town;

  public PayStationImpl( Town town ) {
    this.town = town;
  }
  [...]
}
```

```
public void addPayment( int coinValue ) throws IllegalCoinException {
  switch ( coinValue ) {
  case 5:
  case 10:
  case 25: break;
  default:
    throw new IllegalCoinException("Invalid␣coin:␣"+coinValue+"␣cent.");
  }
  insertedSoFar += coinValue;
  if ( town == Town.ALPHATOWN ) {
    timeBought = insertedSoFar * 2 / 5;
  } else if ( town == Town.BETATOWN ) {
    [the progressive rate policy code]
  }
}
```

Instantiation

```
PayStation ps =
  new PayStationImpl( Town.ALPHATOWN );
```

Benefits

Benefits:

- Simple
  - A conditional statement is one of the first aspects learned by programmers, used widely, and thus easy to understand for any skill level developer team
- Avoid multiple maintenance problem
  - Yeah!!! Common defects/requirements are handled once and for all.

Liabilities

Liabilities:

- Reliability concerns
- Readability concerns
- Responsibility erosion
- Composition problem

Analysis

Reliability/quality problem

- Each time we must add a new rate model (sell in a new town) we must add code to the existing PayStationImpl class.

```
[...]
if ( town == Town.ALPHATOWN ) {
  timeBought = insertedSoFar * 2 / 5;
} else if ( town == Town.BETATOWN ) {
  [BetaTown implementation]
} else if ( town == Town.GAMMATOWN ) {
  [GammaTown implementation]
}
```

- This means potential of introducing errors in old code.
- This means complete regression testing (and test case review) of all product variants!
- *Change by modification* is costly !!!

Analysis

Reliability/quality problem

- Actually our pay station case is the *easiest one!*
- Consider a big system in which there are 83 places where we switch on the town parameter
  - Or – was it **84** places ???

```
[...]
if ( town == Town.ALPHATOWN ) {
  timeBought = insertedSoFar * 2 / 5;
} else if ( town == Town.BETATOWN ) {
  [BetaTown implementation]
} else if ( town == Town.GAMMATOWN ) {
  [GammaTown implementation]
}
```

- *Change by modification* is costly !!!

Analysis

Readability: Code bloat

- If we must handle 43 different price models, then the switching code becomes long and winding, and the original algorithm almost drowns...

Switch creep

- Throwing in “if” often leads to more “if”

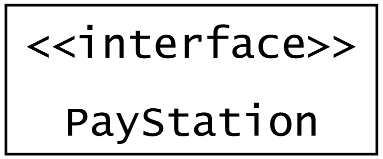
```
if ( Town == ALPHATOWN ) {
   if ( databaseServer == ORACLE && OptimizingOn ){
      if ( DEBUG ) { Console.WriteLine( "..." ); }
      ...
   } else { if ( IsMobilePayment() ) {
      discountFactor = 0.9;
      XXX
} else { ... }
```

Tell me what options are set in the XXX code ? Difficult, huh?

Analysis

Responsibility erosion (“feature creep”)

– Let us review what the responsibilities of the pay station really are now:



Responsibility

1. **Accept payment**
2. **Handle transactions**
3. **Know time bought**
4. **Print recipt**
5. **Handle variations for Alphatown and Betatown**

Wait a few month and the machine is also responsible for parsing XML files, printing debug statements in the console, updating a database, and handle transactions over mobile SMS network !

Composition problem

- A rate model that is a combination of existing ones leads to code duplication [can be avoided by making private methods in the class]
- Example of much worse situation will be dealt with later...

Conditional compilation

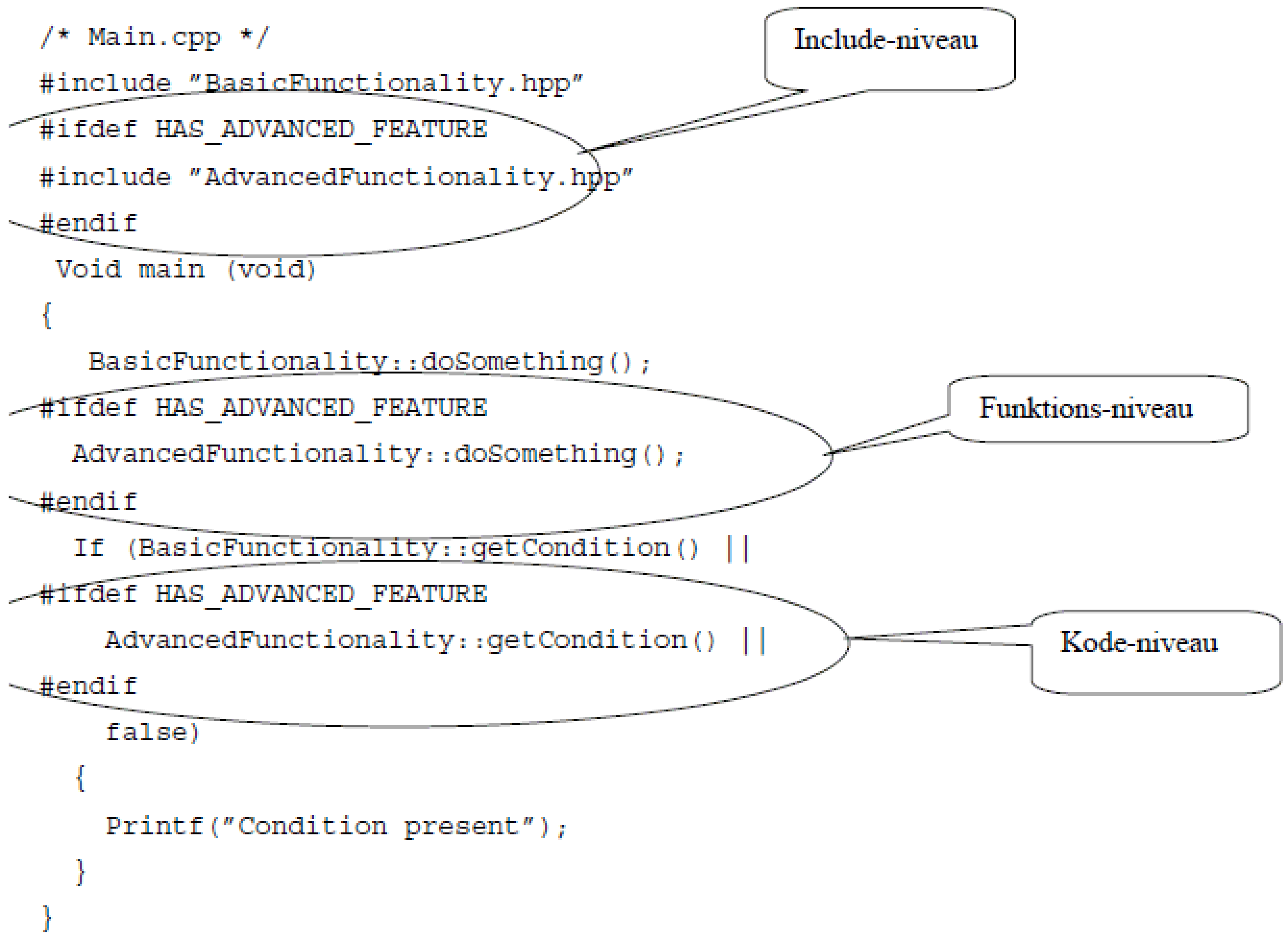
In C and C++ you may alternatively use #ifdef’s

The analysis is basically the same as for parameterization, except that there is no performance penalty --- but choice of which model to be used cannot be made at run-time.

Note: Embedded software where memory footprint of code is important this may be the solution far superior to a pattern based solution!

AARHUS UNIVERSITET

Example



```
/* Main.cpp */
#include "BasicFunctionality.hpp"
#ifdef HAS_ADVANCED_FEATURE
#include "AdvancedFunctionality.hpp"
#endif
 Void main (void)
{
   BasicFunctionality::doSomething();
#ifdef HAS_ADVANCED_FEATURE
  AdvancedFunctionality::doSomething();
#endif
  If (BasicFunctionality::getCondition() ||
#ifdef HAS_ADVANCED_FEATURE
    AdvancedFunctionality::getCondition() ||
#endif
    false)
  {
    Printf("Condition present");
  }
}
```

Example

Data from ”reality”

- 600.000 lines of C++
  - 1.300 classes
  - 2.400 files
- 60.000 staff-days for development
- 3 sites of development

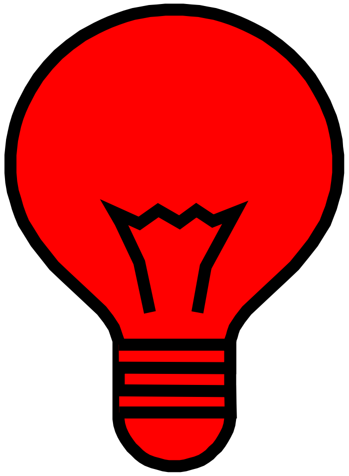
432 parameters (”compile-flags”) must be set to determine the specific variant of the product

- All defined in a make-file (~ build.xml)

Model 2: Summary

It is tempting!

- it is easy - ½ minute in the editor, compile, done!
- the first ‘if’ is easy to overview, understand, and get correct
- but it should turn on the **alarm bell !**



Model 3: Polymorphic Solution

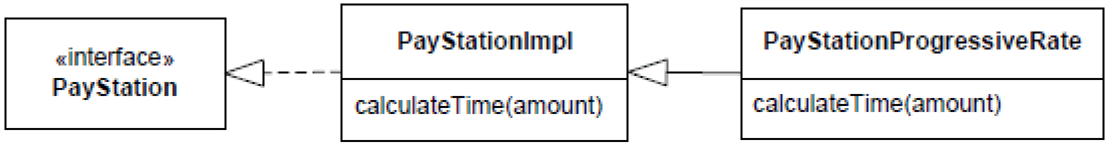
The ‘better’ solution

for many years…

Give a man a hammer and the world will seem to consist purely of nails…

Model 3: Polymorphic proposal

Subclass and override!



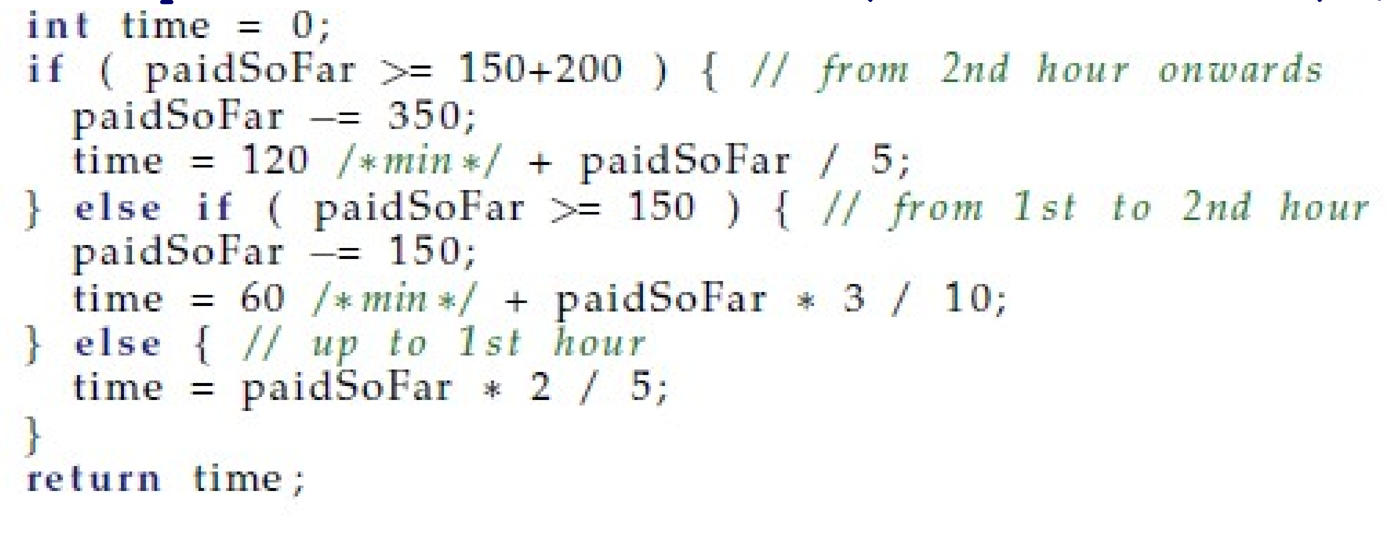
```
public void addPayment( int coinValue )
        throws IllegalCoinException {
  switch ( coinValue ) {
  case 5:
  case 10:
  case 25: break;
  default:
    throw new IllegalCoinException("Invalid coin: "+coinValue);
  }
  insertedSoFar += coinValue;
  timeBought = calculateTime(insertedSoFar);
}
/** calculate the parking time equivalent to the amount of
    cents paid so far
    @param paidSoFar the amount of cents paid so far
    @return the parking time this amount qualifies for
*/
protected int calculateTime(int paidSoFar) {
  return paidSoFar * 2 / 5;
}
```

Proposal 3: Instantiation

Warning: C# ahead

Functionality selection + implementation:

```
 public class PayStationProgressivePrice : PayStationImpl {
   public PayStationProgressivePrice() : base() {}

   override protected int calculateTime(int insertedSoFar) {
```



```
      int time = 0;
      if ( paidSoFar >= 150+200 ) { // from 2nd hour onwards
        paidSoFar -= 350;
        time = 120 /*min*/ + paidSoFar / 5;
      } else if ( paidSoFar >= 150 ) { // from 1st to 2nd hour
        paidSoFar -= 150;
        time = 60 /*min*/ + paidSoFar * 3 / 10;
      } else { // up to 1st hour
        time = paidSoFar * 2 / 5;
      }
      return time;
    }

  }
```

Instantiation:

```
PayStation ps_subclass =
        new PayStationProgressivePrice();
```

Analysis

Benefits
- Avoid multiple maintenance
- Reliability concern
- Code readability

Liabilities
- Increased number of classes
- Inheritance relation spent on single variation type
- Reuse across variants difficult
- Compile-time binding

Benefits

- ☺ Reliability concern
  - – The first time I add a new rate policy I *change by modification!*
    - I have to refactor the code to introduce the new private method *calculateTime*

But

- – All following new requirements regarding rate policies can be handled by **adding** new subclasses, not by **modifying** existing classes.
- – Thus, no fear of introducing defects in existing software; no regression testing, no reviews.

Change by addition, not by modification

Benefits

☺ Readability

- There is no code bloating from introduction conditional statements
- I simply add new classes instead

Liabilities

☹ Increased number of classes

– I have to add one new class for each rate policy variant

– thus instead of 43 if statements in one class I get 43 subclasses to overview

Liabilities

- ☹ Spent inheritance on single variation type
  - – You have "wasted" your single implementation-inheritance capability on one type of variation!
    - The name is odd – isn't it? "PayStationProgressivePrice"
    - What is next:
    - "PayStationProgressivePriceButLiniarInWeekendsWithOracleDataBaseAccessDebuggingVersionAndBothCoinAndMobilePhonePaymentOptions" ???
  - – We will discuss this problem in detail later...

Liabilities

☹ Inheritance is a compile time binding

- Inheritance is a compile time binding !!!
  - you literally write "extends / :" in your editor !!!
- Thus you cannot change rate model except by rewriting code!
  - Sorts of similar to "change by modification ☺"
- And it is completely impossible to dynamically change rate policy at run-time or at start-up time.

Liabilities

☹ Reuse across variants is difficult

- Gammatown
  - “We want a rate policy similar to Alphatown during weekdays but similar to Betatown during weekends.”
- but some code is in one superclass and some in another subclass...
- combining them will lead to a pretty odd design
- or I have to refactor into an abstract superclass that contains the rate policies... But what do they do there?

Model 5: Generative Solution

The masked 'source code copy' approach

Weaving

Source code divided into

- Template code with “holes”
- Code fragments that fit the holes
  - A set defined by the fragments that define a variant

Weaving

- Merge(template, fragment set) => source

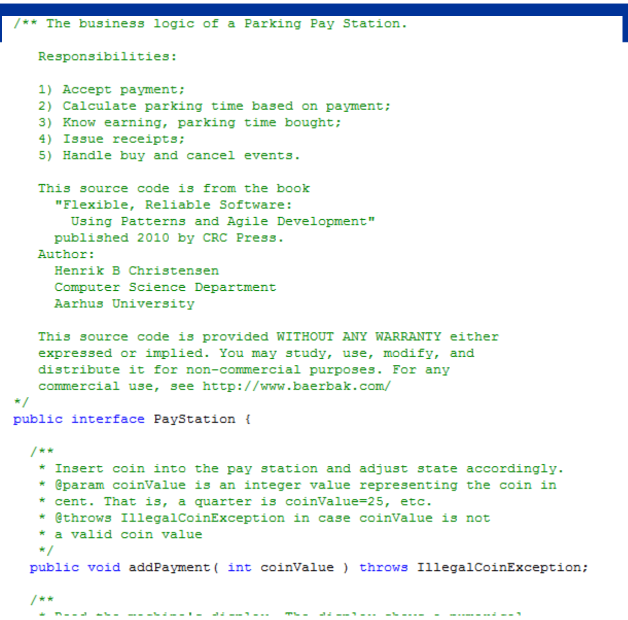
Now you can compile the variant source code.

Example: FMPP used in generating source code for the book in two variants: download or in-book listings

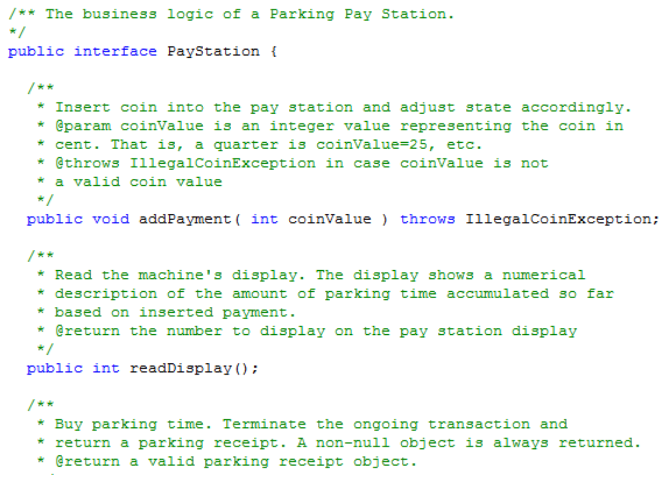
Example: PayStation.java

A ”hole”

```
/** ${paystationClassHeadline}
<#if type == "code">

<#include "/data/paystation/class-responsibilities.txt">

<#include "/data/author.txt">
</#if>
*/
public interface PayStation {

  /**
<#include "/data/paystation/addPayment-specification.txt">
   */
  public void addPayment( int coinValue ) throws IllegalCoinException;

  /**
<#include "/data/paystation/readDisplay-specification.txt">
   */
  public int readDisplay();

  /**
<#include "/data/paystation/buy-specification.txt">
   */
  public Receipt buy();

  /**
<#include "/data/paystation/cancel-specification.txt">
   */
  public void cancel();
}
```



```
/** The business logic of a Parking Pay Station.

   Responsibilities:

   1) Accept payment;
   2) Calculate parking time based on payment;
   3) Know earning, parking time bought;
   4) Issue receipts;
   5) Handle buy and cancel events.

   This source code is from the book
     "Flexible, Reliable Software:
       Using Patterns and Agile Development"
     published 2010 by CRC Press.
   Author:
     Henrik B Christensen
     Computer Science Department
     Aarhus University

   This source code is provided WITHOUT ANY WARRANTY either
   expressed or implied. You may study, use, modify, and
   distribute it for non-commercial purposes. For any
   commercial use, see http://www.baerbak.com/
*/
public interface PayStation {

  /**
   * Insert coin into the pay station and adjust state accordingly.
   * @param coinValue is an integer value representing the coin in
   * cent. That is, a quarter is coinValue=25, etc.
   * @throws IllegalCoinException in case coinValue is not
   * a valid coin value
   */
  public void addPayment( int coinValue ) throws IllegalCoinException;

  /**
```



```
/** The business logic of a Parking Pay Station.
*/
public interface PayStation {

  /**
   * Insert coin into the pay station and adjust state accordingly.
   * @param coinValue is an integer value representing the coin in
   * cent. That is, a quarter is coinValue=25, etc.
   * @throws IllegalCoinException in case coinValue is not
   * a valid coin value
   */
  public void addPayment( int coinValue ) throws IllegalCoinException;

  /**
   * Read the machine's display. The display shows a numerical
   * description of the amount of parking time accumulated so far
   * based on inserted payment.
   * @return the number to display on the pay station display
   */
  public int readDisplay();

  /**
   * Buy parking time. Terminate the ongoing transaction and
   * return a parking receipt. A non-null object is always returned.
   * @return a valid parking receipt object.
```

Weaving

Examples

- Maven archetype
- AspectJ – aspect oriented programming
- FMPP that handles aspects of my book’s code
- And for those with white beards ☺
    - Beta fragment system
    - Trine’s slots

My experience

This type systems pops up again and again

- Maven archtype is the newest I know of…

It is basically *source-code-copy* over again

- But with some tooling support to avoid multiple maintenance problem

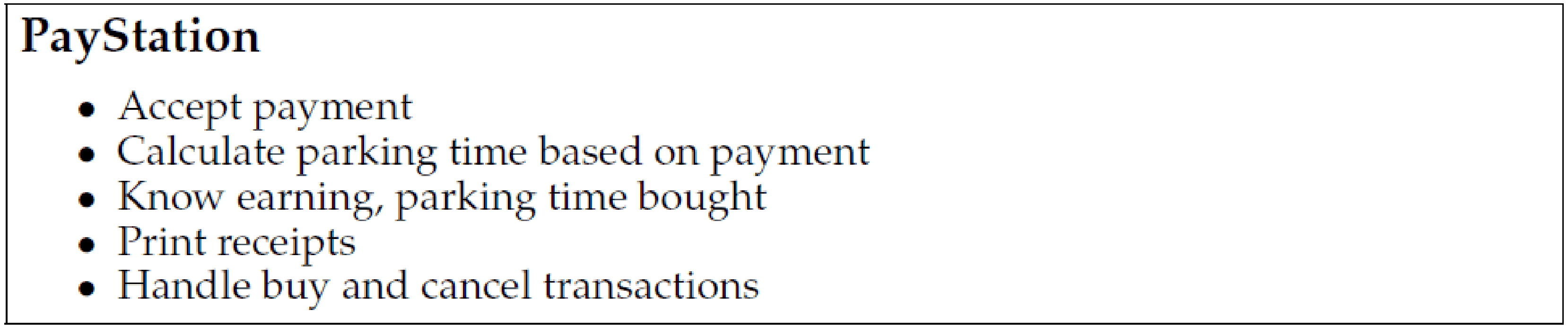
However

- It stinks! Why?
- **Because the executing code differs from what I see in my editor!**
  - **We short-circuit our power of reasoning => BUGS!**
- Morale: Avoid it if possible…
- But it is not always possible.
  - I use it for my book’s code – I have no other option (except *manual* source code copy – yikes…)

Model 4: Compositional Solution

A fresh and new look at the problem

Proposal 4: Composition



PayStation

- Accept payment
- Calculate parking time based on payment
- Know earning, parking time bought
- Print receipts
- Handle buy and cancel transactions

Golden rule: *No abstraction should have too many responsibilities. Max 3 is a good rule of thumb…*

(Facade objects are an exception)

Serving too many responsibilities

The reason that we have to *modify code* to handle the new requirement instead of *adding code* is because

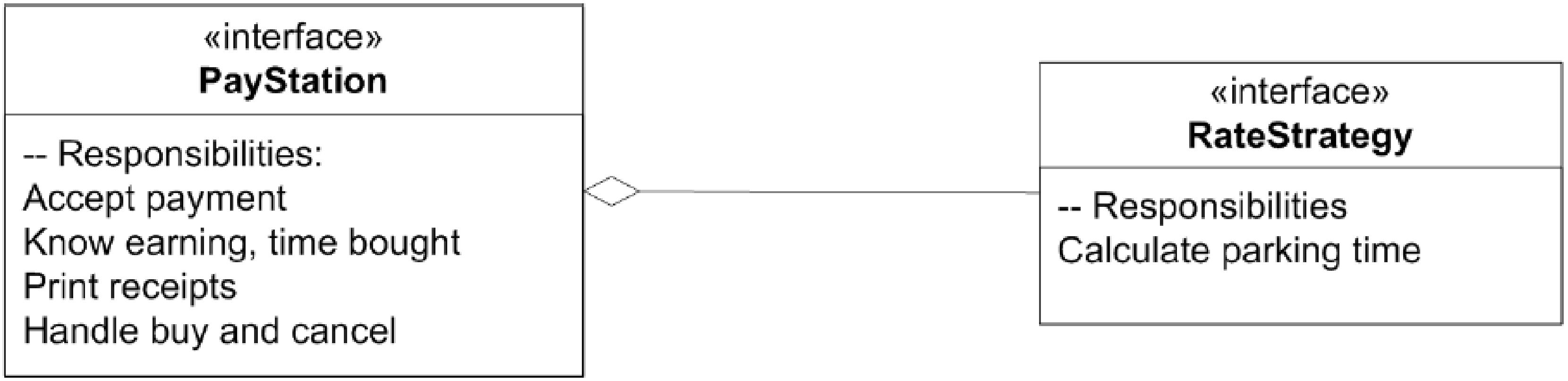
The change revolves around a responsibility (calculate parking time) that is buried within an abstraction and mixed up with many other responsibilities (print receipt, handle buy, etc.) !!!

So: What do we do???

Divide responsibilities - compose them

A proposal is simply to

Put the responsibility in its own abstraction / object

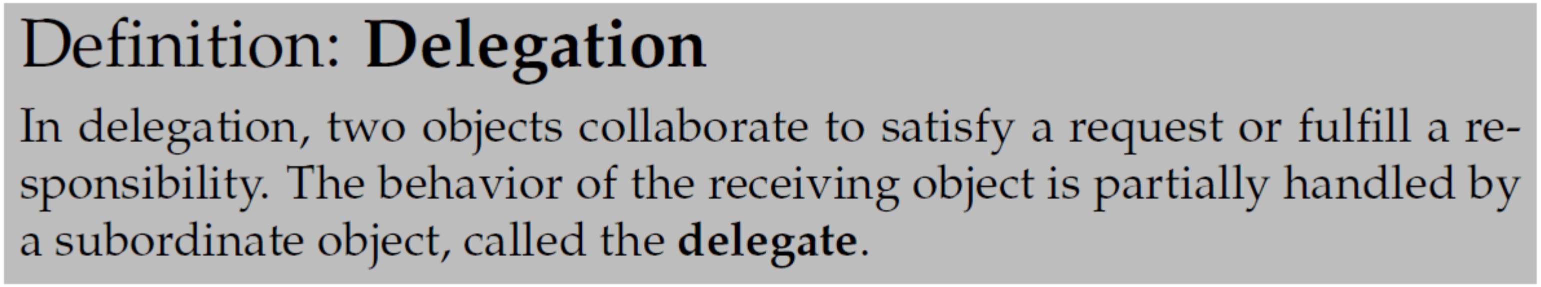


Delegation

The basic principle is simple but powerful:

- Instead of one object doing it all by itself, it *asks* another object to help out. Some of the job is handled by another “actor” – the *delegate*

This principle has a name:

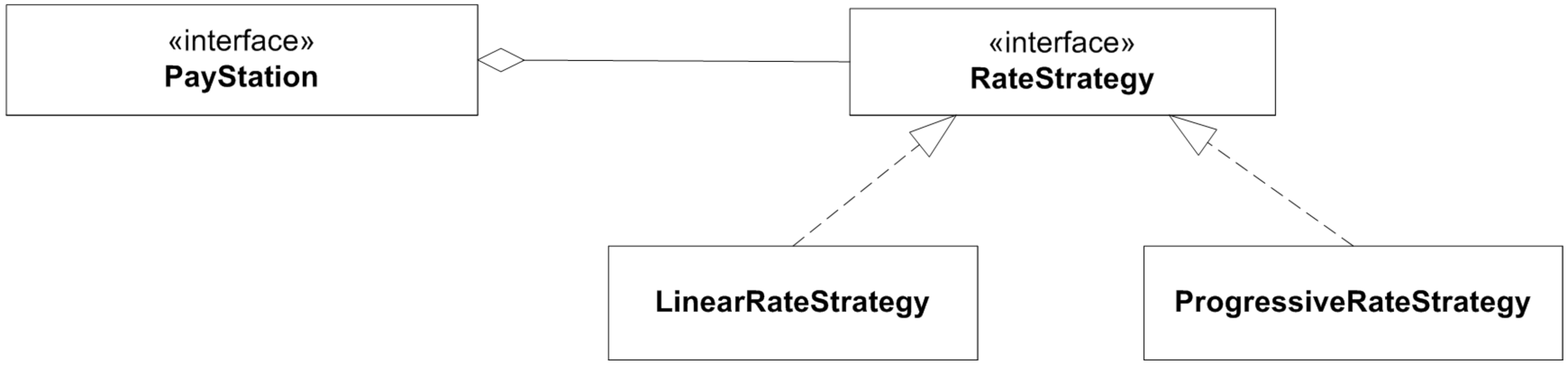


Definition: **Delegation**

In delegation, two objects collaborate to satisfy a request or fulfill a responsibility. The behavior of the receiving object is partially handled by a subordinate object, called the **delegate**.

Concrete behaviours

Responsibilities must be served by concrete behaviour in objects...



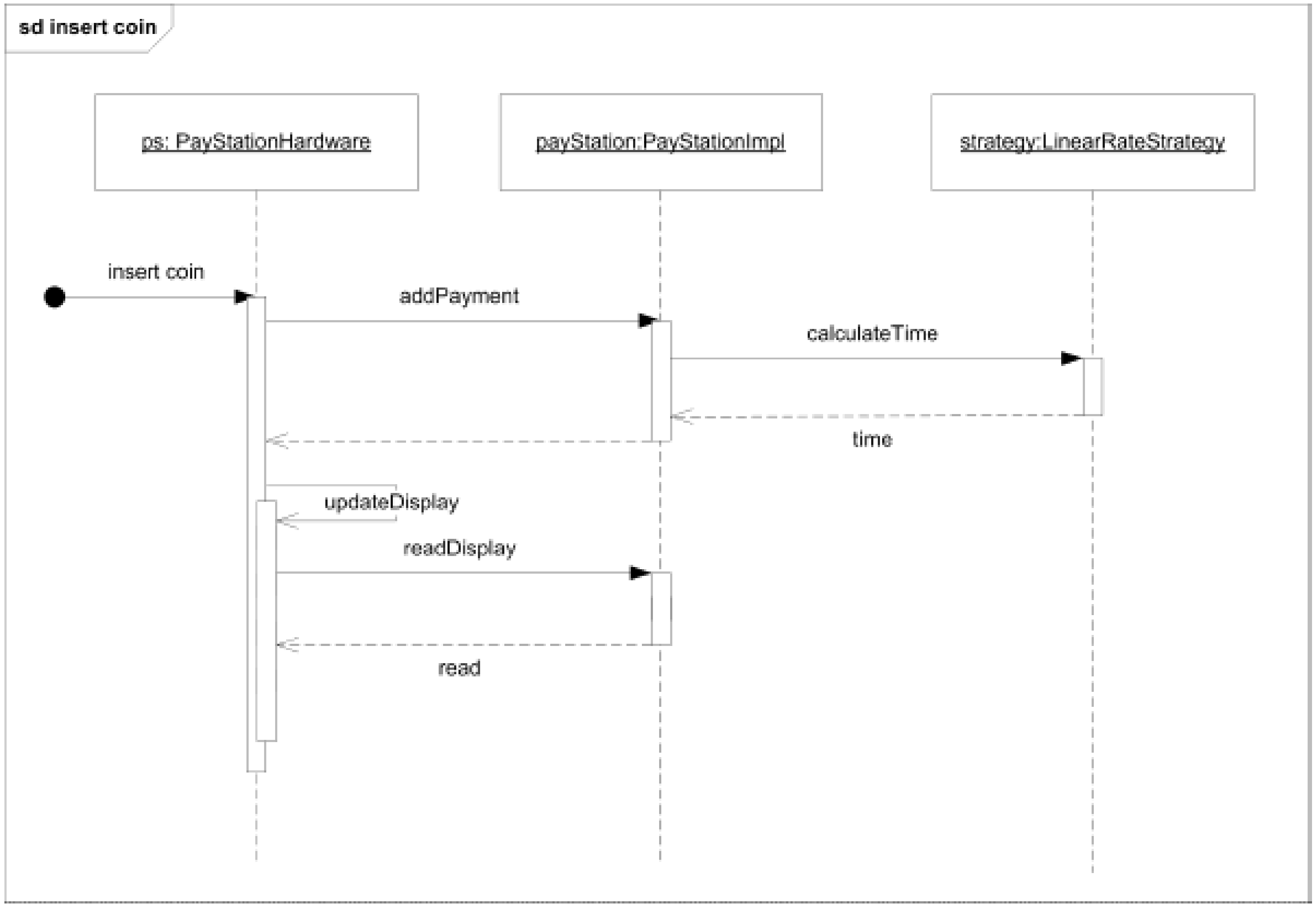
Code View

```
public class PayStationImpl implements PayStation {
  private int insertedSoFar;
  private int timeBought;

  /** the strategy for rate calculations */
  private RateStrategy rateStrategy;
  ...
```

and modify the addPayment method:

```
  public void addPayment( int coinValue ) throws IllegalCoinException {
    switch ( coinValue ) {
    case 5:
    case 10:
    case 25: break;
    default:
      throw new IllegalCoinException("Invalid␣coin:␣"+coinValue+"␣cent.");
    }
    insertedSoFar += coinValue;
    timeBought = rateStrategy.calculateTime(insertedSoFar);
  }
```

Behaviour


sd insert coin
ps: PayStationHardware
payStation:PayStationImpl
strategy:LinearRateStrategy
insert coin
addPayment
calculateTime
time
updateDisplay
readDisplay
read

Exercise

The pay station needs to know which rate strategy object to use, of course!

How do we tell it ???

Choosing pricing

Warning: C# ahead

Several possibilities

- Constructor
- Set-method
- Creational patterns (later on☺)

Functionality selection:

```
// The rate calculation strategy used
RateStrategy rateStrategy;

public PayStationImpl( RateStrategy rs ) : base() {
   rateStrategy = rs;
}
```

Exercise

What are the benefits and liabilities of

- Using the constructor to define the strategy?

- Using a set-method to define the strategy?

Analysis

Constructor

- Compiler will tell you that you have forgotten to make it!
  - Much less cost than letting the customer find out !!!
- Early binding that cannot be changed at run-time

Set-method

- You **will** forget to set it !!!
- ... but you can change your mind at run-time !
  - (at least for stateless objects like strategy objects...)

Analysis

Benefits

- Readability
- Run-time binding
- Separation of responsibilities
- Variant selection is localized
- Combinatorial

Liabilities

- Increased number of interfaces, objects
- Clients must be aware of strategies

Analysis

- ☺ Readability
  – no code bloat of conditional statements

- ☺ Run-time binding
  – I can actually change the rate policy while the system is running. Leads to lower maintenance costs as no shut down required

Benefits

☺ Responsibilities clearly stated in interfaces

- Leads to No Odd Naming:
- **PayStation** and **RateStrategy**: The responsibilities
- LinearRateStrategy ect: Concrete behaviour fulfilling responsibilities

- The pay station has “lost some fat”
  - by separating responsibilities the cohesion of the code within each abstraction is higher

  - Note though that from the GUI/hardware’s perspective, the pay station *still* has the ‘formal’ responsibility to calculate rates!

Benefits

- ☺ Variant selection localized
  - There is only one place in the code where I decide which rate policy to take
    - namely in the configuration/main code where I instantiate the pay station!
  - contrast to the parametric solution where selection and decision code is smeared all over the place
  - **No variant handling code at all** in the pay station code !

Benefits

☺ Combinatorial

- I have not used inheritance – we can still subclass it to provide new behavior on other aspects – without interfering with the rate calculation!
- But – much more on that later...

The 3-1-2 process

So – Pizza from the ingredients ☺

AARHUS UNIVERSITET

③ We have *identified some behaviour* that is *likely to change*…

- rate policies

① We have clearly stated a responsibility that covers this behaviour and expressed it in an interface:

```
<<interface>>
RateStrategy
-- Calculate Parkingtime
```

② The parking machine now perform rate calculations by letting a *delegate* object do it: the RateStrategy object.

- time = rateStrategy.calculateTime(amount);

The 3-1-2 process

I call this “mini-process” of handling variability for the 3-1-2 process

The reason for the odd numbering is its relations to the *principles of flexible design* that are put forward in the Design Pattern book (GoF) by Gamma et al.

The number refer to the sequence in the GoF book that the principle is mentioned.

Transferring responsibilities

Actually this is a common thing in everyday life

Some years ago I transferred the responsibility to empty the garbage can to my eldest son

- (not without some heated arguments though ☺)

Project leaders' main responsibility is – to delegate responsibility to other people

And why? Because A) we cannot do everything ourselves and B) too many responsibilities leads to stress and errors!

Key technique: Delegation

In software this simple technique *"let someone else do the dirty job"* is called **delegation**.

Instead of an object doing it itself:

- time = this.calculateTime(amount);
- this.takeGarbageToGarbageCan();

we let some *specialist* object do it for us:

- time = priceStrategy.calculateTime(amount);
- son.takeGarbageToGarbageCan();

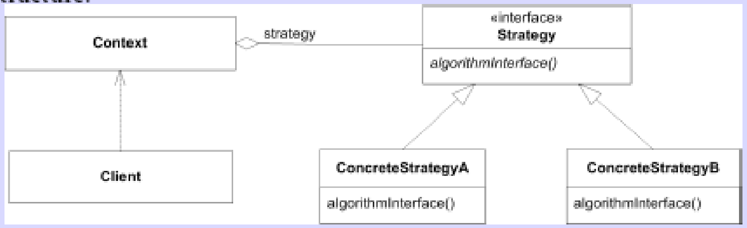
Conclusion

We have *derived* the strategy pattern by analysing our problem in a certain way!

[3.1] Design Pattern: Strategy

Intent	Define a family of business rules or algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithms vary independently from clients that use it.
Problem	Your product must support variable algorithms or business rules and you a flexible and reliable way of controling the variability.
Solution	Separate selection of algorithm from its implementation by expressing the algorithms responsibilities in an interface and let each implementation of the algorithms realize this interface.

Structure:



Roles	**Strategy** specifies the responsibility and interface of the algorithm. **ConcreteStrategies** defines concrete behaviour fulfilling the responsibility. **Context** performs its work for **Client** by delegating to an instance of type **Strategy**.
Cost - Benefit	The benefits are: *Strategies eliminate conditional statements*. It is an *alternative to subclassing*. It facilitates *separate testing* of **Context** and **ConcreteStrategy**. The liabilities are: *Increased number of objects*. *Clients must be aware of strategies*.

Consequences (GoF)

Families of related algorithms. Hierarchies of Strategy classes define a family of algorithms or behaviors for contexts to reuse. Inheritance can help factor out common functionality of the algorithms.

An alternative to subclassing. Inheritance offers another way to support a variety of algorithms or behaviors. You can subclass a Context class directly to give it different behaviors. But this hard-wires the behavior into Context. It mixes the algorithm implementation with Context's, making Context harder to understand, maintain, and extend. And you can't vary the algorithm dynamically. You wind up with many related classes whose only difference is the algorithm or behavior they employ. Encapsulating the algorithm in separate Strategy classes lets you vary the algorithm independently of its context, making it easier to switch, understand, and extend.

Strategies eliminate conditional statements. The Strategy pattern offers an alternative to conditional statements for selecting desired behavior. When different behaviors are lumped into one class, it's hard to avoid using conditional statements to select the right behavior. Encapsulating the behavior in separate Strategy classes eliminates these conditional statements. Code containing many conditional statements often indicates the need to apply the Strategy pattern.

AARHUS UNIVERSITET

Consequences (GoF)

A choice of implementations. Strategies can provide different implementations of the *same* behavior. The client can choose among strategies with different time and space trade-offs.

Clients must be aware of different Strategies. The pattern has a potential drawback in that a client must understand how Strategies differ before it can select the appropriate one. Clients might be exposed to implementation issues. Therefore you should use the Strategy pattern only when the variation in behavior is relevant to clients.

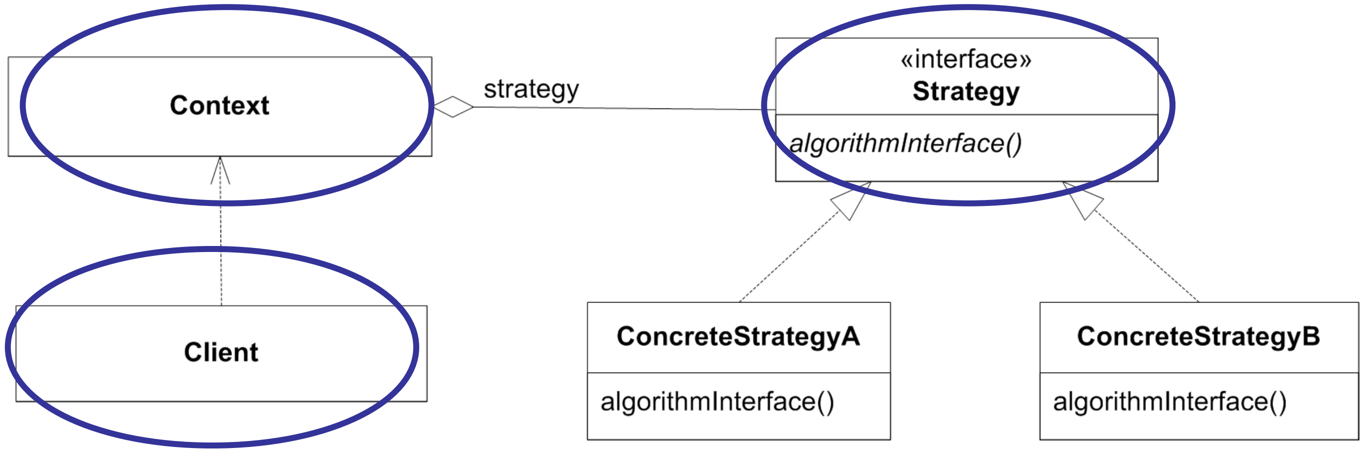
Communication overhead between Strategy and Context. The Strategy interface is shared by all ConcreteStrategy classes whether the algorithms they implement are trivial or complex. Hence it's likely that some ConcreteStrategies won't use all the information passed to them through this interface; simple ConcreteStrategies may use none of it! That means there will be times when the context creates and initializes parameters that never get used. If this is an issue, then you'll need tighter coupling between Strategy and Context.

Increased number of objects. Strategies increase the number of objects in an application. Sometimes you can reduce this overhead by implementing strategies as stateless objects that contexts can share. Any residual state is maintained by the context, which passes it in each request to the Strategy object. Shared strategies should not maintain state across invocations. The Flyweight (195) pattern describes this approach in more detail.

Terminology

Strategy defines three roles:

Client, **Context** and **Strategy**



Summary

From the ingredients

- ③ identified behaviour likely to change
- ① express responsibility for behaviour as interfaces
- ② use delegation to support behaviour

we have derived a pattern *automagically* ☺

This is the nuts and bolts for most (behavioural) patterns !