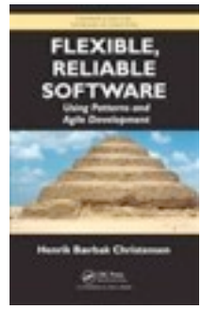
Theme 2
Program Design

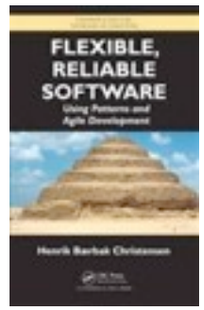
Deriving Abstract Factory
Pattern Fragility

Learning Objectives



- to establish the ABSTRACT FACTORY as a solution to the problem of creating variable types of objects
- to show how this pattern also comes naturally from a compositional design philosophy
  - but not completely...
- to highlight the importance of getting the implementation right as
  - demonstrate how even small errors may cripple the advantages a pattern was supposed to have.

The Receipt class revisited



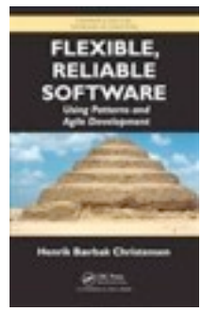
- add responsibility:
  - know its value in minutes parking time
  - *print itself*

- so, will introduce a single method
  public void print(PrintStream stream);

- Result:

```
----------------------------------------
-------  P A R K I N G   R E C E I P T  -------
              Value 049 minutes.
            Car parked at 08:06
----------------------------------------
```

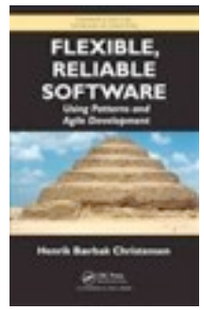
New requirements

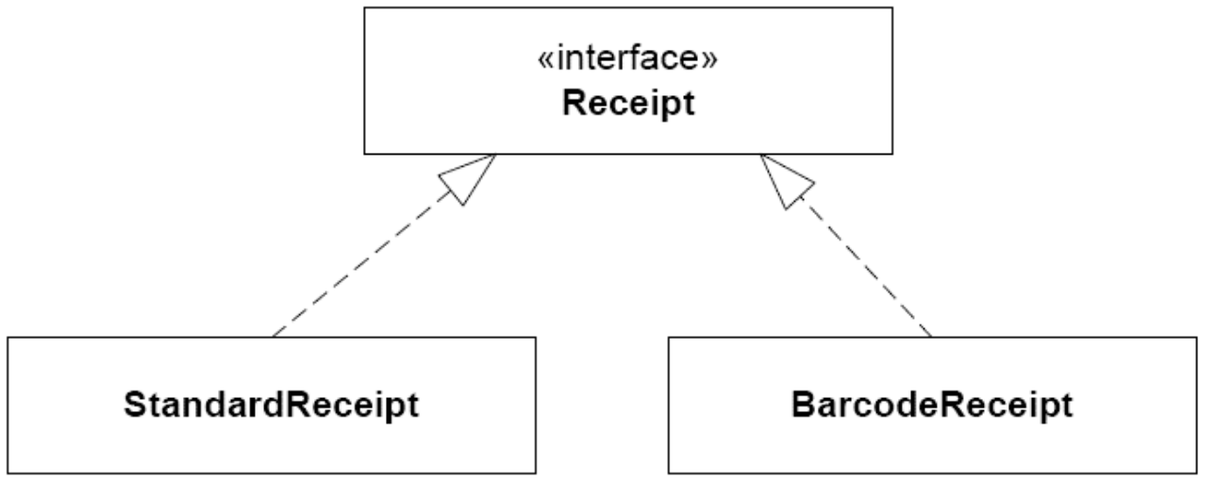


- *Change is the only constant in software development.*
- Betatown wants receipts with bar code for easy scanning

```
----------------------------------------
-------  P A R K I N G   R E C E I P T  -------
              Value 049 minutes.
            Car parked at 08:06
||  ||||| | || ||| || ||  ||| | || |||| | || ||||
----------------------------------------
```

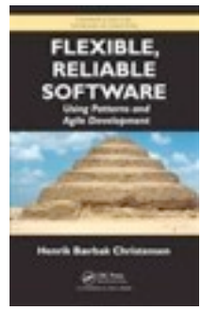
Variability Point



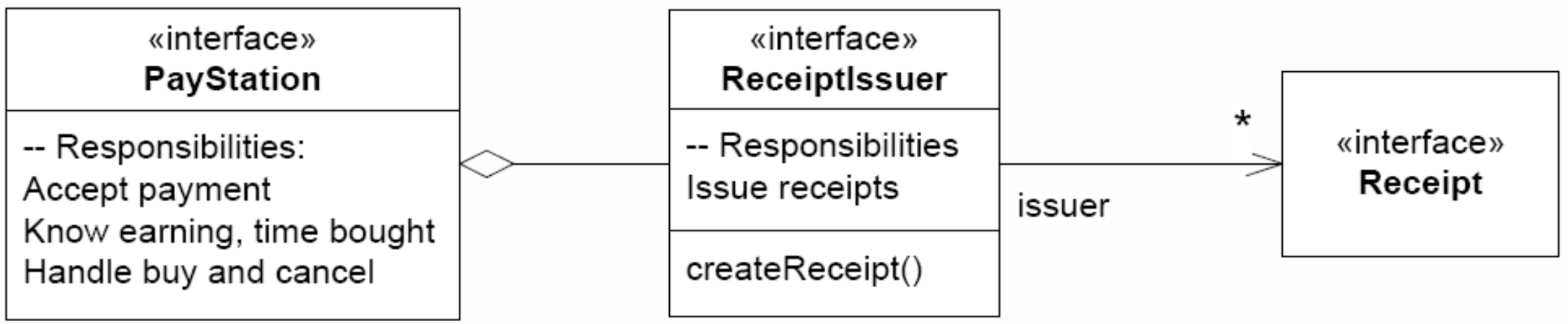


	Variability points	
Product	**Rate**	**Receipt**
Alphatown	Linear	Standard
Betatown	Progressive	Barcode
Gammatown	Alternating	Standard

Compositional Approach

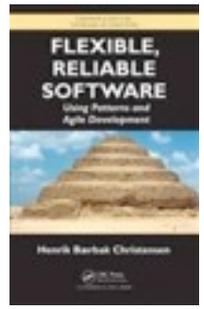


- Identify what varies: *instantiation of receipts*
- Interface expressing responsibility: *ReceiptIssuer*
- Compose behaviour: *delegate to ReceiptIssuer*



Question: do we really need this additional *ReceiptIssuer* abstraction?

Try the *ReceiptIssuer*



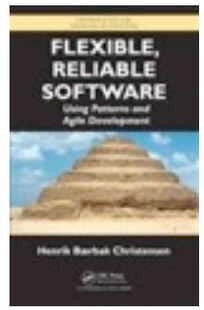
- TDD tells us:
  - refactor to introduce ReceiptIssuer
  - add bar code receipts to BetaTown

```
PayStation ps;
/** Fixture for pay station testing. */
@Before
public void setUp() {
  ps = new PayStationImpl( new One2OneRateStrategy() );
}
```

```
PayStation ps;
/** Fixture for pay station testing. */
@Before
public void setUp() {
  ps = new PayStationImpl( new One2OneRateStrategy(),
                           new StandardReceiptIssuer() );
}
```

Fact: configuration responsibility is assigned to two different objects

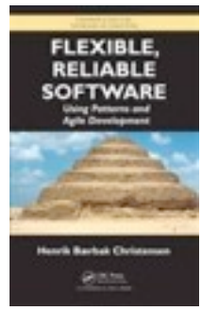
Rethink the design

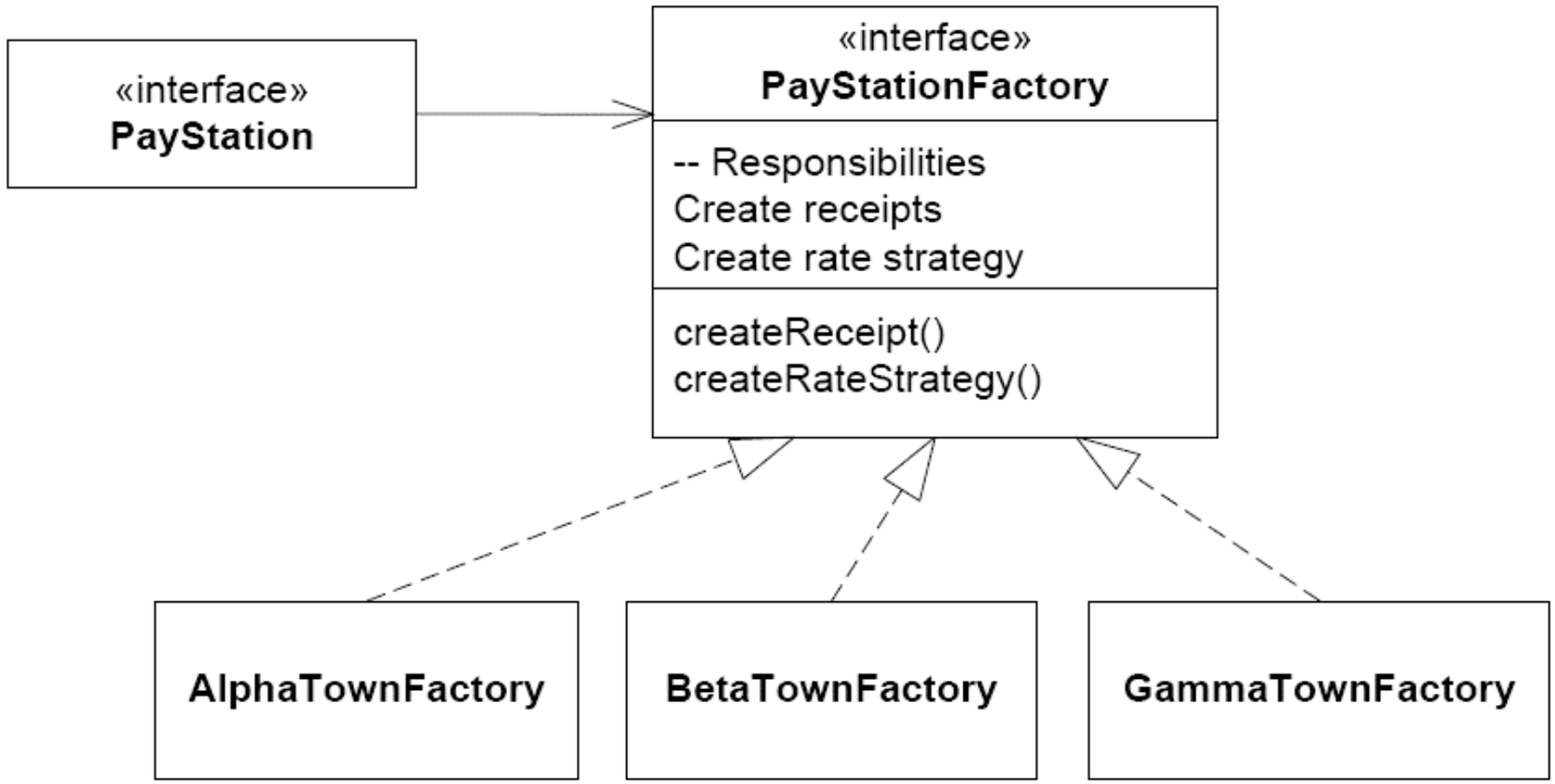


- Rethink the responsibilities:
  - one object alone is responsible for creating all objects that are related to the *paystation configuration*.
  - define responsibility of creation of objects in a single place, often called a *factory*.

- PayStationFactory:
  - create receipts
  - create rate strategies

Factories


FLEXIBLE,
RELIABLE
SOFTWARE
Using Patterns and
Agile Development
Henrik Bærbak Christensen


«interface»
PayStation
«interface»
PayStationFactory
-- Responsibilities
Create receipts
Create rate strategy
createReceipt()
createRateStrategy()
AlphaTownFactory
BetaTownFactory
GammaTownFactory

Use TDD


```
PayStation ps;
/** Fixture for pay station testing. */
@Before
public void setUp() {
  ps = new PayStationImpl( new TestTownFactory() );
```

```
public interface PayStationFactory {
  /** Create an instance of the rate strategy to use. */
  public RateStrategy createRateStrategy();

  /** Create an instance of the receipt.
   * @param the number of minutes the receipt represents. */
  public Receipt createReceipt( int parkingTime );
}
```

```
public class PayStationImpl implements PayStation {
  [...]
  /** the strategy for rate calculations */
  private RateStrategy rateStrategy;
  /** the factory that defines strategies */
  private PayStationFactory factory;

  /** Construct a pay station.
      @param factory the factory to produce strategies
  */
  public PayStationImpl( PayStationFactory factory ) {
    this.factory = factory;
    this.rateStrategy = factory.createRateStrategy();
    reset();
  }
  [...]
  public Receipt buy() {
    Receipt r = factory.createReceipt(timeBought);
    reset();
    return r;
  }
  [...]
}
```

does not compile

introduce the factory interface and implement the TestTownFactory class

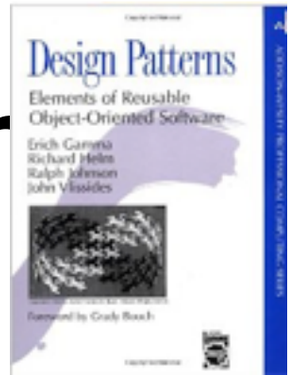
refactor the PayStationImpl to use the factory

go on with making the proper factories for the different towns.

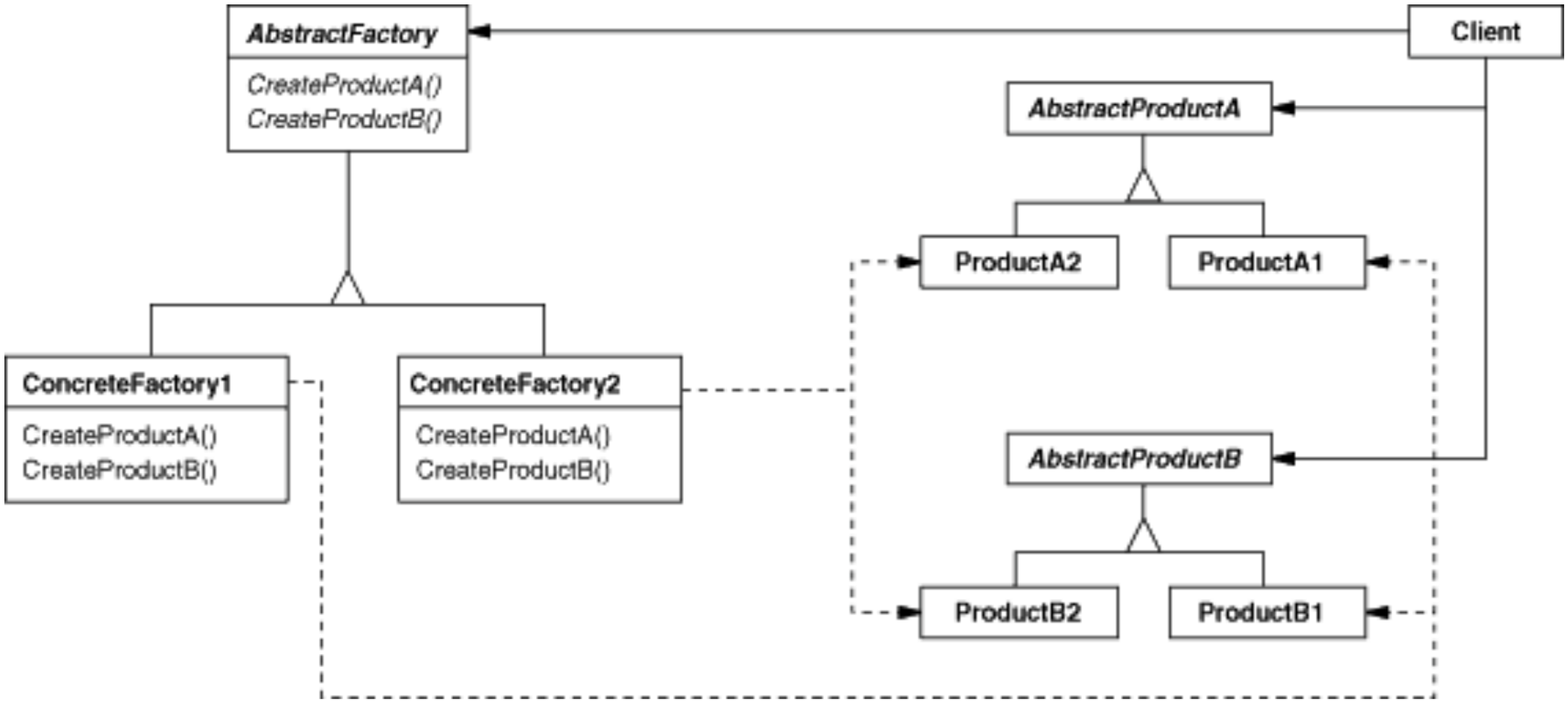
The Compositional Process Again

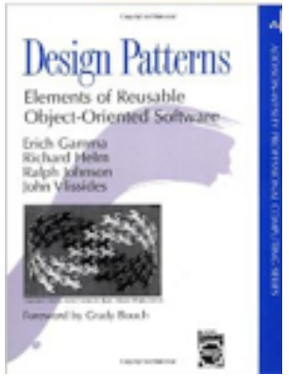
- Identify some behaviour that *varies* between different products:
  - creating objects
- Express the *responsibility* of creating objects in an *interface*
  - *PayStationFactory* expresses this responsibility
- Let the pay station *delegate all creation of objects* it needs to the *delegate object*

Abstract Factory Patterr



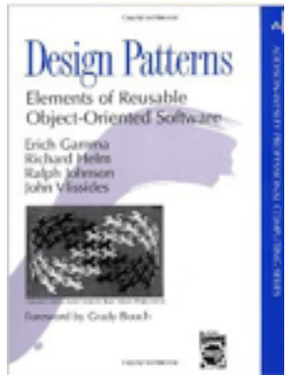
- **Intent**:
  - **provide an interface for creating families of related or dependent objects without specifying their concrete classes.**
- **Applicability**:
  - a system should be independent of how its products are created, composed, and presented
  - a system should be configured with one of multiple families of products
  - a family of related product objects is designed to be used together, and enforcement of this constraint is needed
  - you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.





AbstractFactory defines a common interface for object creation. **ProductA** defines the interface of an object, **ProductA1**, required by the client.
ConcreteFactory1 is responsible for creating products belonging to the variant 1.

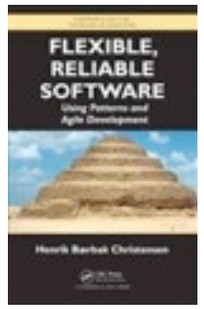
Consequences



- + Low coupling between client and product
- + Isolates concrete classes
- + Makes exchanging product families easy
- + Promotes consistency between products
- +- Supporting new types of products is difficult

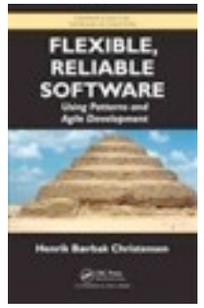
Pattern Fragility

Why Patterns?

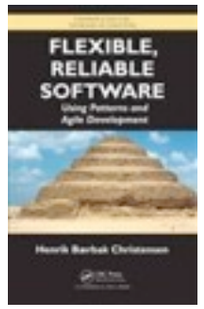


- Design patterns organize and structure code in a particular way.
  - Static: arrangement of classes/interfaces
  - Dynamic: assignment of responsibility, interaction patterns
- Why:
  - Because I get some benefits from doing so
- Bottom line:
  - *Patterns are means to a goal, not the goal itself*

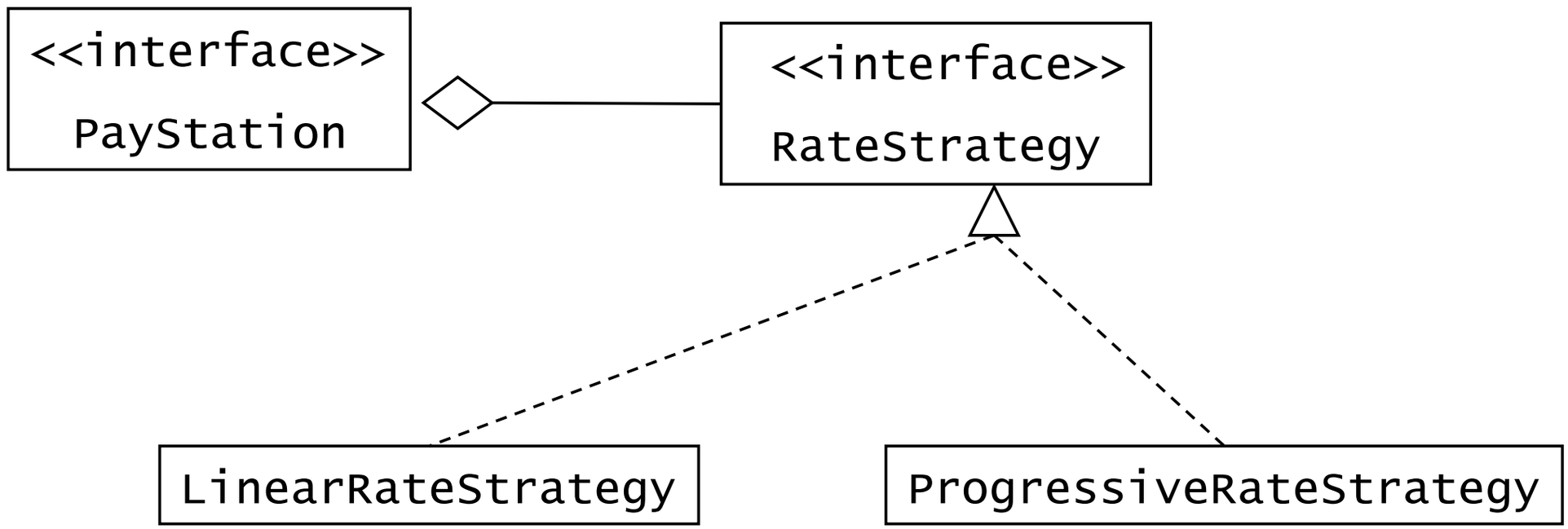
Patterns are encoded



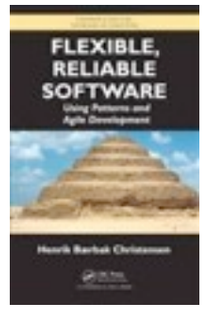
- **Pattern Fragility**
  Pattern fragility is the property of design patterns that their benefits can only be fully utilized if the pattern's object structure and interaction patterns are implemented correctly.



Example: Strategy



Pitfalls: Declaration of delegates



Do not use class names in declarations!

Why is the following change a disaster?

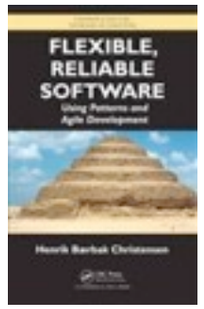
```
public class PayStationImpl implements PayStation {
  [...]

  /** the strategy for rate calculations */
  private ProgressiveRateStrategy rateStrategy;

  [...]
}
```

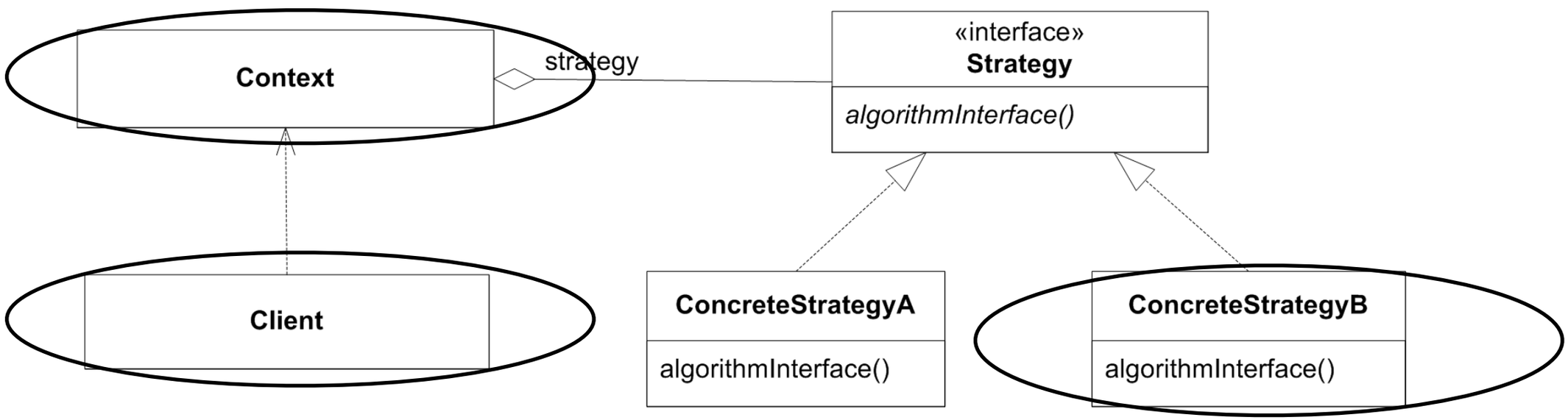
Declare object references that play part in a design pattern by their interface type, never by their concrete class type.

Pitfalls: Binding in the right place

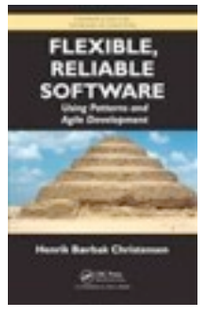


- Loose coupling is fine, but we have to couple the objects together eventually.
- **It is important that the binding is made**
  - **in the right place**
  - **as few places as possible (optimally 1!)**

- Many possibilities for Strategy:



Pitfalls: Binding in the right place

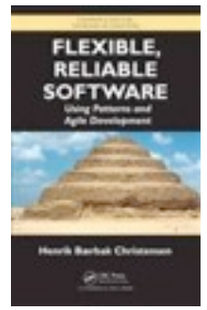



```
public class PayStationImpl implements PayStation {
  [...]
  public void addPayment( int coinValue ) throws IllegalCoinException {
    switch ( coinValue ) {
    case 5:
    case 10:
    case 25: break;
    default:
      throw new IllegalCoinException("Invalid coin: "+coinValue+" cent.");
    }
    insertedSoFar += coinValue;
    RateStrategy rateStrategy = new LinearRateStrategy();
    timeBought = rateStrategy.calculateTime(insertedSoFar);
  }
  [...]
}
```

Binding in the context object:

all the patterns liabilities
none of the patterns benefits!!

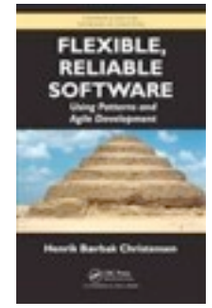
Pitfalls: Binding in the right place



- **Object should be created and coupled in the production code units whose responsibility are explicitly configuration and binding**

- In Strategy, this is normally the Client role
- Abstract Factory's purpose is to define bindings.
  - the factory is often the right place to make bindings.
- In State it is actually often the ConcreteState objects that define the 'next state'

Pitfalls: Concealed Parametrisation



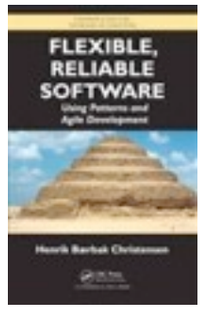
Assume: Previous binding survived.

Later: *"Why does Betatown not work any more?*
I need to fix it, and fix it fast!"

```
public class PayStationImpl implements PayStation {
  [...]
  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;
    RateStrategy rateStrategy;
    if ( town == Town.ALPHATOWN ) {
      rateStrategy = new LinearRateStrategy();
    } else if ( town == Town.BETATOWN ) {
      rateStrategy = new ProgressiveRateStrategy();
    }
    timeBought = rateStrategy.calculateTime(insertedSoFar);
  }
  [...]
}
```

Decide on a design strategy to handle a given variability and stick to it.

Pitfalls: Responsibility Erosion



Software changes its own requirement.

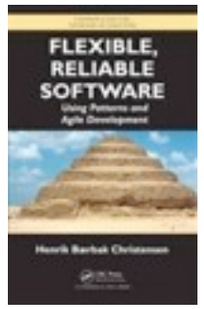
New (weird) request:

Gammatown: explain rate policy.

```
public class AlternatingRateStrategy implements RateStrategy {
  [...]
  public int calculateTime( int amount ) {
    if ( decisionStrategy.isWeekend() ) {
      currentState = weekendStrategy;
    } else {
      currentState = weekdayStrategy;
    }
    return currentState.calculateTime( amount );
  }

  public String explanationText() {
    if ( currentState == weekdayStrategy ) {
      return [the explanation for weekday];
    } else {
      return [the explanation for weekend];
    }
  }
}
```

Pitfalls: Responsibility Erosion



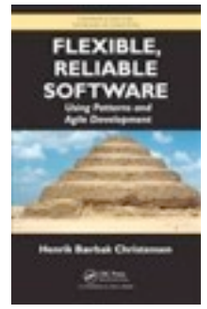
However, this strategy does not conform to the contract by the interface.
So, need to type check:

```
if ( rateStrategy instanceof AlternatingRateStrategy ) {
  AlternatingRateStrategy rs =
    (AlternatingRateStrategy) rateStrategy;
  String theExplanation = rs.explanationText();
  [use it somehow]
}
```

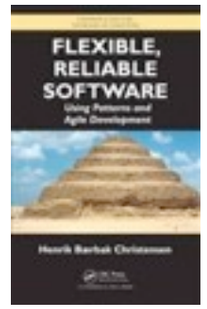
Possible solution: Move the method up into the *RateStrategy* interface.

But: I have now added a new responsibility.
One that may not be very cohesive.

Pitfalls: Responsibility Erosion



Carefully analyse new requirements to avoid responsibility erosion and bloating interfaces with in-cohesive methods.



Conclusion

- Take care at the implementation level!!!
  - It only takes a few "slip-ups" to completely destroy the intended benefits of a pattern!

You do not learn patterns by reading a book or listening to me!

DO IT: CODE! and reflect!