Adding New Requirement to PayStation Example

PayStation Example

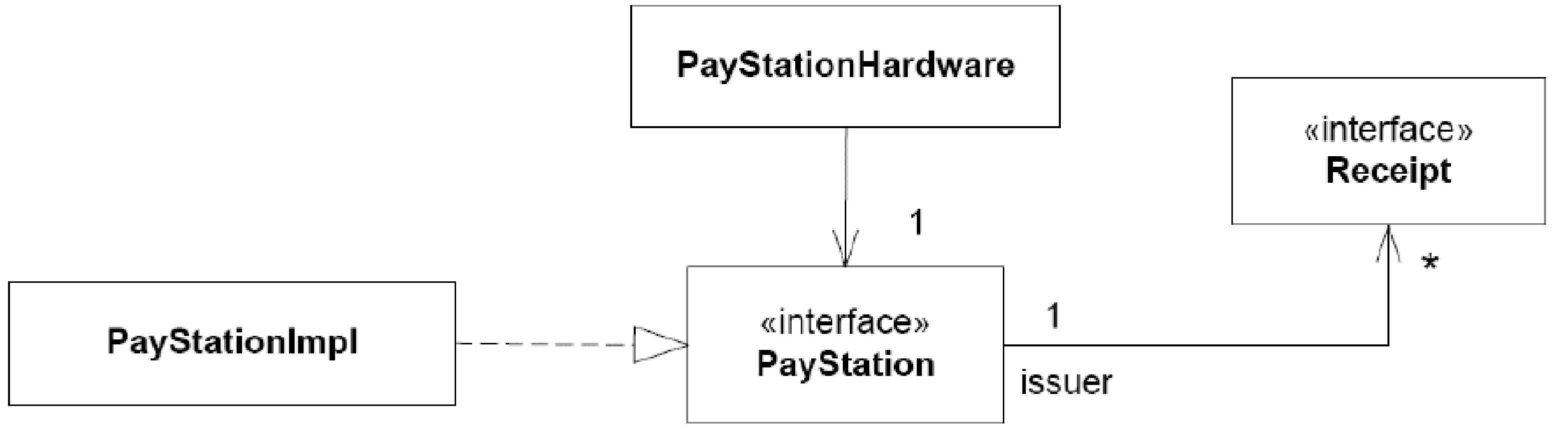
Story 1: Buy a parking ticket. A car driver walks to the pay station to buy parking time. He enters several valid coins (5, 10, and 25 cents) as payment. For each payment of 5 cents he receives 2 minutes parking time. On the pay station's display he can see how much parking time he has bought so far. Once he is satisfied with the amount of time, he presses the button marked "Buy". He receives a printed receipt, stating the number of minutes parking time he has bought. The display is cleared to prepare for another transaction.

Story 2: Cancel a transaction. A driver has entered several coins but realize that the accumulated parking time shown in the display exceeds what she needs. She presses the button marked "Cancel" and her coins are returned. The display is cleared to prepare for another transaction.

Story 3: Reject illegal coin. A driver has entered 50 cents total and the display reads "20". By mistake, he enters a 1 euro coin which is not a recognized coin. The pay station rejects the coin and the display is not updated.

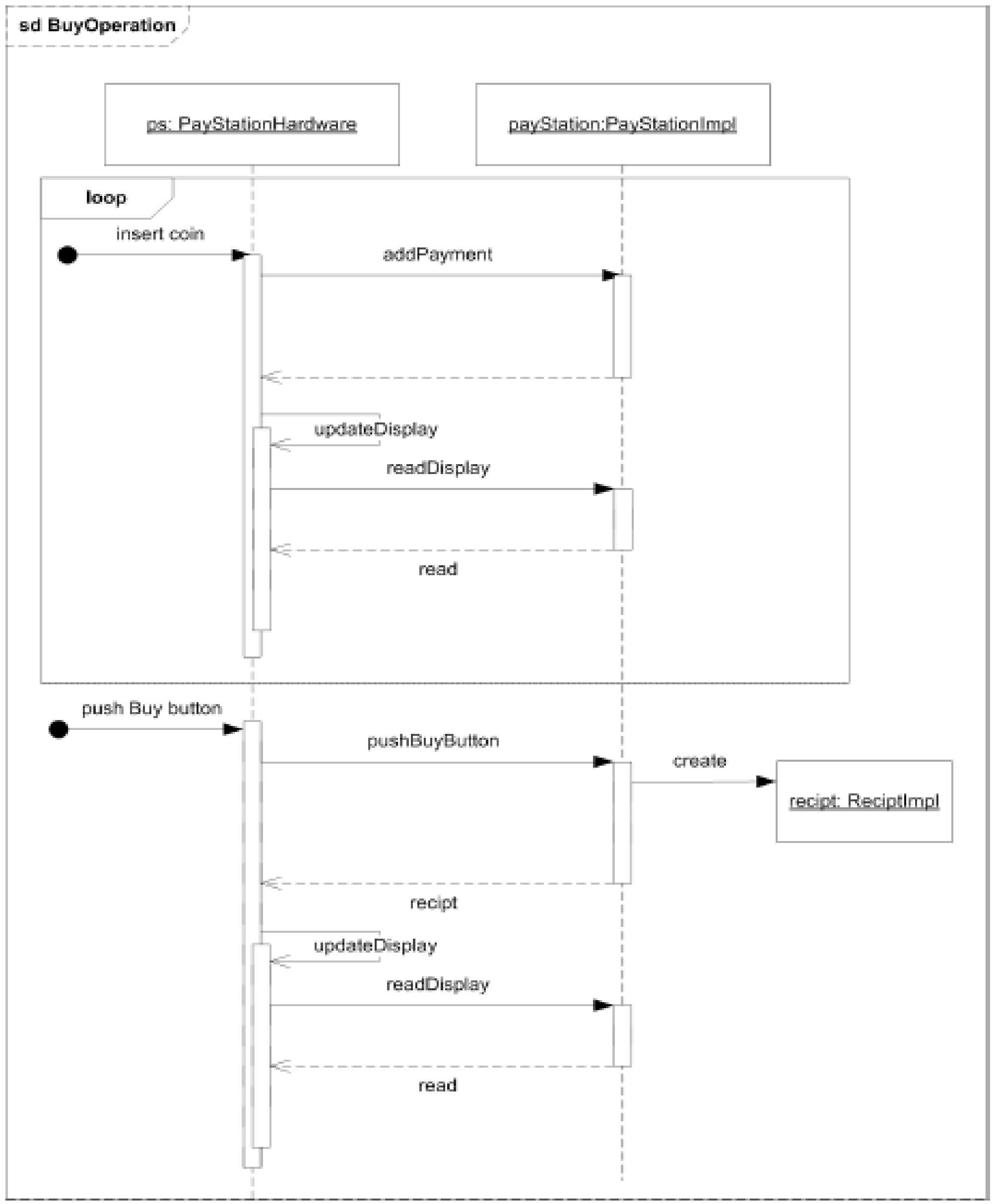
*Example extracted from: Flexible, Reliable Software: Using Patterns and Agile Development. Henrik B. Christensen

PayStation Example



*Example extracted from: Flexible, Reliable Software: Using Patterns and Agile Development. Henrik B. Christensen

PayStation Example



*Example extracted from: Flexible, Reliable Software: Using Patterns and Agile Development.
Henrik B. Christensen

Adding New Requirement to PayStation

- A new customer, Boston city, wants to buy the PayStation system but has another requirement on how rates are calculated.

- Boston city requires a progressive rate according to the following scheme:
  - First hour: $1,5 (5 cents gives 2 minutes)
  - Second hour: $2 (5 cents gives 1,5 minutes)
  - Third and following hours: $3 (5 cents gives 1 minute)

Adding New Requirement to PayStation

- **Problem formulation:** We need to develop and maintain two or more variants of the software system in a way that introduces the least cost and defects.

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

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

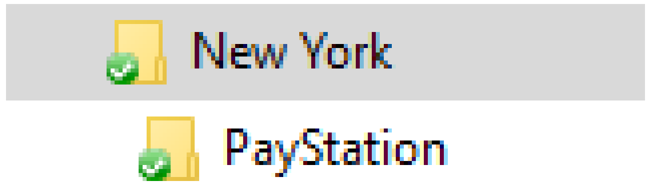
Adding New Requirement to PayStation:

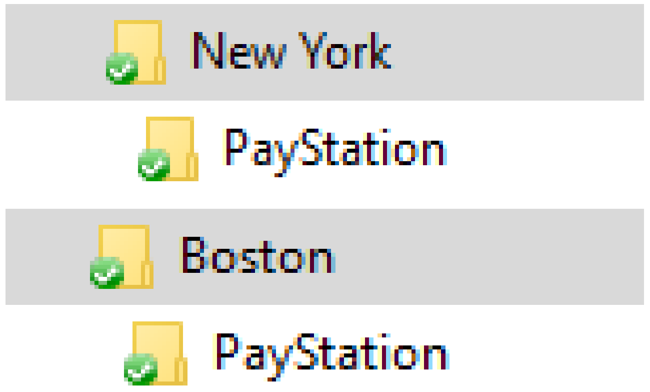
Four Proposals

- **Source tree copy proposal:** Making a copy of all the source code.
- **Parametric proposal:** Introducing a parameter that defines the town the software is running in. Only one copy of the production code but at appropriate places the code branches on this parameter to give the behavior required.
- **Polymorphic proposal:** Subclassing the PayStationImpl and override a method that calculates the rate.
- **Compositional proposal:** Describing the variable behavior, rate calculation, in an interface and have classes implementing each concrete rate calculation policy. Then, pay station call such a rate calculation object instead of performing the calculation itself.

Adding New Requirement to PayStation:
Source Tree Copy Proposal

- Copy the source code (in a source code folder tree), rename the root of the tree to Boston city and overwrite the rate calculation with the proper algorithm for the progressive rate.





- Most of the test cases have to be rewritten because almost of them rely on the rate policy

Adding New Requirement to PayStation:

Source Tree Copy Proposal Benefits

- **Speed:** It is quick! It is a matter of a single copy operation, modify the test cases and let the production code changes, and you are ready to ship it to the customer.
- **Simple:** The idea is simple and easy to explain to colleagues and new developers.
- **No implementation interference:** The two implementations do not interfere with each other. For instance, if we introduce a defect in one implementation it is guaranteed that it will have no implications on the other implementation.

Adding New Requirement to PayStation:
Source Tree Copy Proposal Limitations

- **Multiple maintenance problem:**
  - If we want to introduce a next generation pay station that can keep track of its earning (a new method amountEarned) we have to implement the same production code and the same test cases for each source tree.
  - Then, if a defect is detected in this behavior, the defect also has to be removed in each code tree.

Adding New Requirement to PayStation:

Parametric Proposal

- Use an if statement that selects the proper code block to execute based upon which city code is running in.

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

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

 public void addPayment (int coinValue ) throws IllegalCoinException {
  ...
  default:
   throw new IllegalCoinException ("Invalid coin: "+coinValue); }
  insertedSoFar += coinValue;
  if (town == Town.NEWYORK) { timeBought = insertedSoFar / 5 * 2; }
  else if (town == Town.BOSTON) { [the progressive rate policy code] }
 }
 ...
```

Adding New Requirement to PayStation:
Parametric Proposal Benefits

- **Simple:** Conditionals are easy to understand and often one of the first language constructs introduced to new learners of programming. Thus the parametric idea is easy to describe to other developers.

- **Avoids the multiple maintenance problem:** There is only one code base to maintain for both towns.
  - New features have to be introduced once in one production code base.
  - Defects in common behavior have to be fixed once and for all towns.

Adding New Requirement to PayStation:
Parametric Proposal Limitations

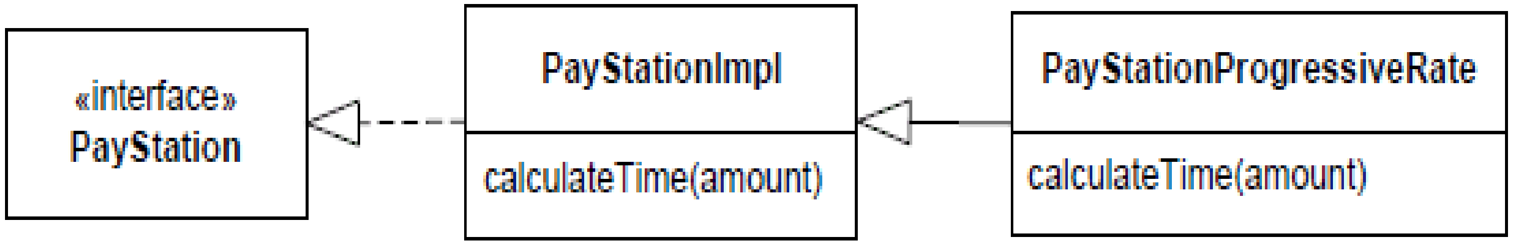
- **Reliability concerns:** The new requirement is handled by *change by modification* that has a risk of introducing new defects, even for the NEWYORK town.

- **Analyzability concerns:** As more and more requirements are handled by parameter switching, the production code becomes less easy to analyze (difficult to read, long, or slow).

- **Responsibility erosion:** The pay station has, without much notice, been given an extra responsibility. The responsibility “Handle variants of the product” has been added.

Adding New Requirement to PayStation:
Polymorphic Proposal

- Use polymorphism and subclassing.



```
public void addPayment (int coinValue ) throws IllegalCoinException {
  ...
  default:
    throw new IllegalCoinException("Invalid coin: "+coinValue); }
  insertedSoFar += coinValue;
  timeBought = calculateTime (insertedSoFar); }

protected int calculateTime (int paidSoFar ) { return paidSoFar * 5 / 2; }
```

Adding New Requirement to PayStation:
Polymorphic Proposal Benefits

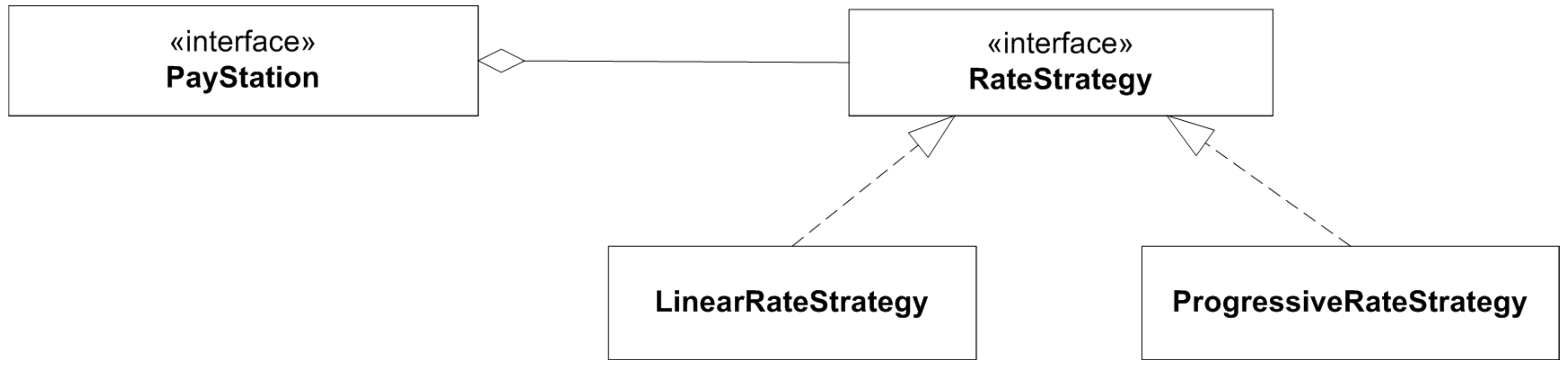
- **Avoid multiple maintenance problem:** One code base which is a good property.
- **Reliability concern:** The production code has once and for all been prepared for any new rate policy to be required later.
- **Code analyzability:** This proposal does not suffer from code bloat, you have to read less code to understand a variant.

Adding New Requirement to PayStation:
Polymorphic Proposal Limitations

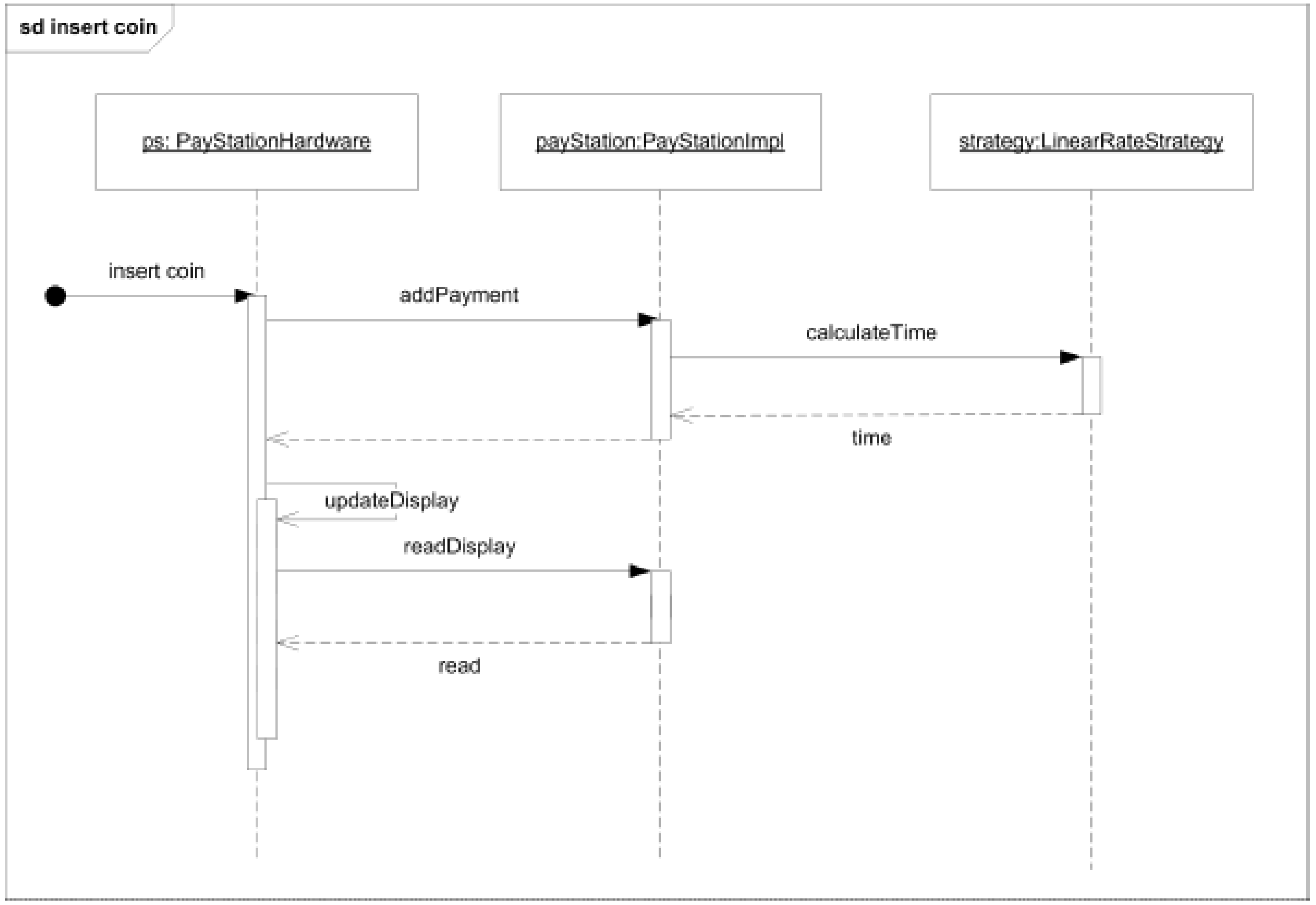
- **Increased number of classes:** Every new rate policy will introduce a new subclass.
- **Inheritance relation spent on single type of variation:** I cannot use the inheritance relation to handle other types of variation.
- **Reuse across variants difficult:** It turns out to be cumbersome to reuse the algorithms defined by one variant in another.
- **Compile-time binding:** The binding between the particular rate policy and the pay station is defined at compile-time.

Adding New Requirement to PayStation:
Compositional Proposal

- Let objects collaborate so their combined effort provide the required behavior. That is, move the responsibility Calculate parking time based on payment away from the pay station abstraction and put it into a new abstraction.



Adding New Requirement to PayStation:
Compositional Proposal



Adding New Requirement to PayStation:

Compositional Proposal Benefits

- **Reliability:** The pay station has been refactored once and for all with respect to rate calculations, and new rate structures can be handled only by adding more classes. Change by addition instead of change by modification.
- **Run-time binding:** The binding between the pay station and its associate rate calculation can be changed at run-time.
- **Separation of responsibilities:** Responsibilities are clearly separated and assigned to easily identifiable abstractions in the design.

Adding New Requirement to PayStation:

Compositional Proposal Benefits

- **Separation of testing:** As the responsibilities have been separated, rate calculations and core pay station functionalities may be tested independently.

- **Variant selection is localized:** The code that decides what particular variant of rate calculation to use is in one spot only (during the initialization process).

- **Combinatorial:** Other types of variability may be introduced without interfering with the rate calculation variability.

Adding New Requirement to PayStation:
Compositional Proposal Limitations

- **Increased number of classes and interfaces:** The number of classes and interfaces has grown in comparison to the original design.

- **Client must be aware of strategies:** The selection of which rate policy to use is no longer in the pay station but still someone has to make the decision (typically the object that instantiates the pay station object). Variant selection is moved to the client objects.

References

- *Flexible, Reliable Software: Using Patterns and Agile Development*
  Henrik B. Christensen
  CRC Press