Multi Dimensional Variance: How to make ultra flexible software!

Goal and means to an end?

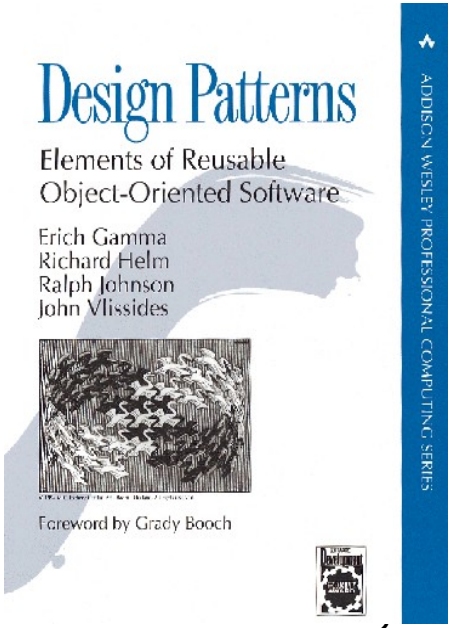
Patterns:

Goal in itself or just the means to an end?

Patterns are interesting as *means* to achieve some specific quality in our software:

– elements of **Reusable** ...

A key aspect is handling **variance**



Variance

Factoring out in roles and delegating to objects that play roles is a very strong technique to handle **multiple dimensions of variance!**

- that is – a piece of software that must handle different types of context
  - work on both Oracle and MS database
  - work in both debug and production environment
  - work both with real hardware attached or simulated environment
  - work with variations for four different customers

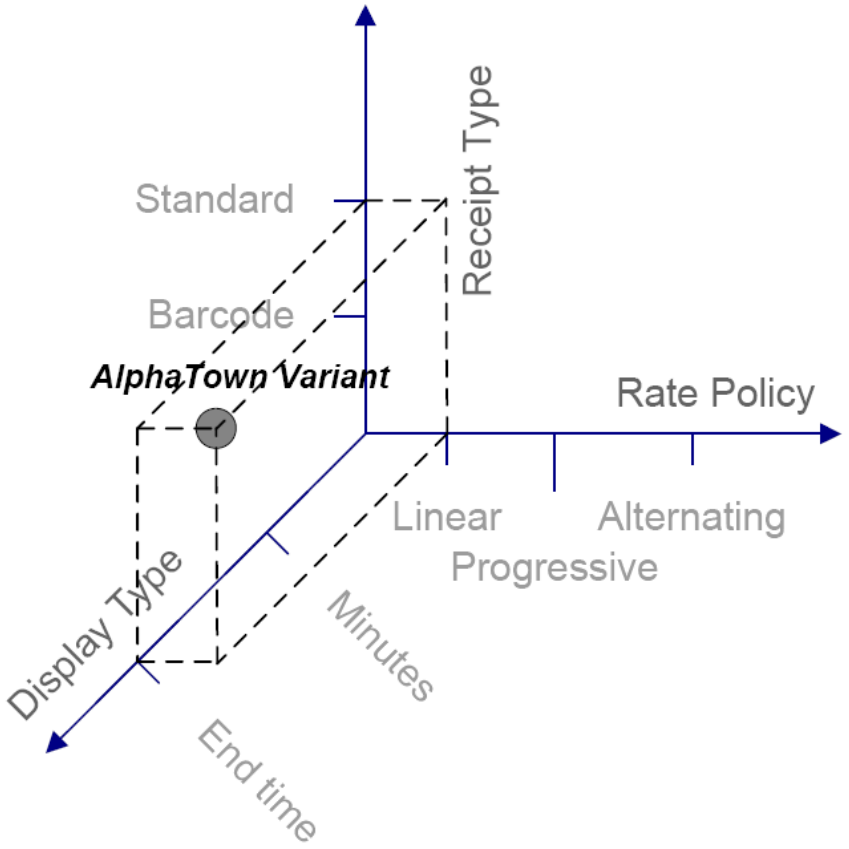
Here all types of combinations are viable !

New Requirements

Alphatown county wants the display to show *parking end time* instead of minutes bought!

Combinatorial explosion!

All these requirements pose a *combinatorial explosion* of variants ☹



There are 3*2*2 = 12 combinations. This may be doubled if we include overriding weekend day algorithm !

Variance by switching

Variant handling by #ifdef's or switching is well known, but the code simply bloats with conditional statements.

Example: GNU C compiler has a single statement that includes 41 macro expansions !!!

I wonder what that code does???

#ifdef (MSDOS && ORACLE || MYSQL && …)

#ifdef (DEBUG)

- quickly you loose control of what is going on…

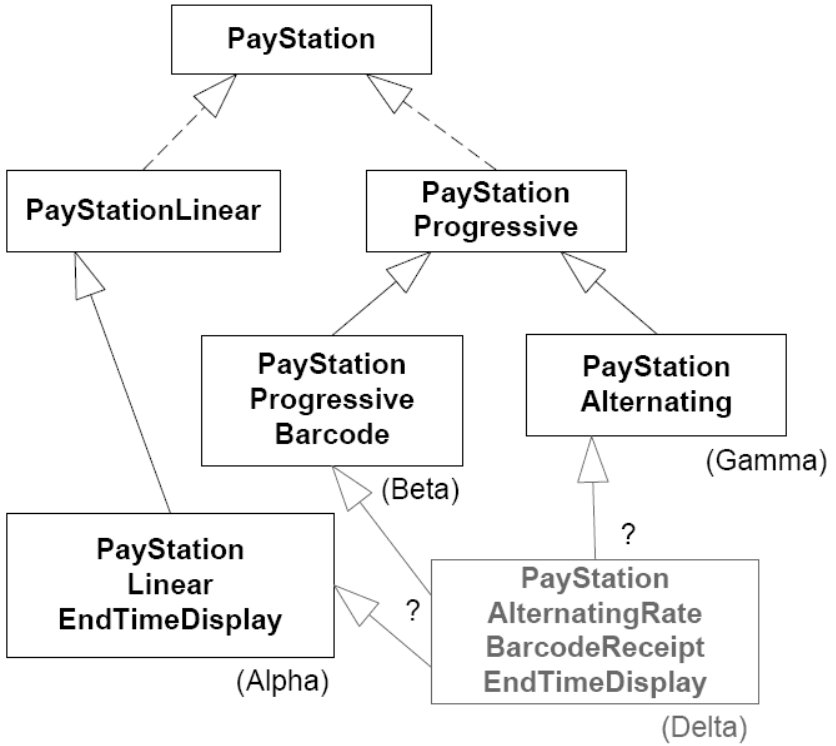
Variance by inheritance

Inheritance dies **miserably** facing this challenge!

Just look at names!

Making new variants is difficult.

And code reuse is very difficult ☹



Masking the problem

By combining inheritance and switching you may mask the problem somewhat.

I.e. handle receipt type by inheritance, and the rest by pumping the code with if's…

but … it is still an inferior way to handle multi-dimensional variance…

Compositional software

The way forward is:

Compositional software

Highly configurable and flexible software!

③ Consider what behaviour that may vary

① Express variable behaviour as a responsibility clearly defined by an interface

② Delegate to object serving the responsibility to perform behaviour

Compositional software

[Demo]

Compositional Software

[Backgammon Demo]

Compositional software

The parking machine has become a *team leader*, delegating jobs to his specialist workers:

```
public int readDisplay() {
  return displayStrategy.reading();
}
```

```
timeBought = rateStrategy.calculateTime(insertedSoFar);
```

```
public Receipt buy() {
  Receipt recipt = factory.createReceipt( this );
  resetTransaction();
  return recipt;
}
```

Note! No if's – no bloat – easy to read code leading to fewer bugs!

Compositional software

Telling the team leader which persons will serve the roles:

The factory interface

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

  /** Create an instance of the displayStrategy to use. */
  DisplayStrategy createDisplayStrategy( PayStation ps );

  /** Create an instance of the receipt.
      @param the number of minutes parking time the receipt is valid for.
  */
  Receipt createReceipt( int parkingTime );

}
```

Compositional software

Creating a parking machine:

- create the factory
- create the pay station, giving it access to the factory

```
PayStationFactory psf = new AlphaTownFactory();
PayStation pm = new PayStationImpl( psf );
```

Compositional software

... and a factory:

```
class AlphaTownFactory : PayStationFactory {
      public RateStrategy createRateStrategy() {
         return new LinearRateStrategy();
       }
      public DisplayStrategy createDisplayStrategy() {
         return new EndTimeDisplayStrategy();
       }
      public Receipt createReceipt( int parkingTime ) {
         return new StandardReceipt(parkingTime);
      }
   }
```

Analysis

Benefits

- The variability points are independent
  - we introduced new display strategy – but this did not alter any of the existing strategies !

```
public int readDisplay() {
  return displayStrategy.reading();
}
```

- Once the variability point has been introduced we can introduce as many new types of variations as we like
  - only by *adding* new classes
    - any price model; new receipt types; new display output...
- **Open-closed principle in action...**

Open/Closed principle

Open for extension

Closed for modification

Analysis

Benefits

- Any combination it is possible to “mix”
- Nonsense combinations can be delimited
  - abstract factory is the place to “mix” the cocktails
- Code readability
  - every aspect of the configuration is clearly defined in a single place
    - configuration mixing in the abstract factory
    - orchestration in the Parking Machine impl
    - each variation type in its own implementing class

Analysis

Liabilities

- Each dimension of variability (price model, receipt type, display output, etc) is *really* independent – so
- we cannot feed information from one to the other ☹
- Example:
  - The Alternating Rate policy needs to know whether it is weekend or not – but this is separated in a special strategy – thus the one needs the other...
- But – of course we can handle this – by a pattern ☺

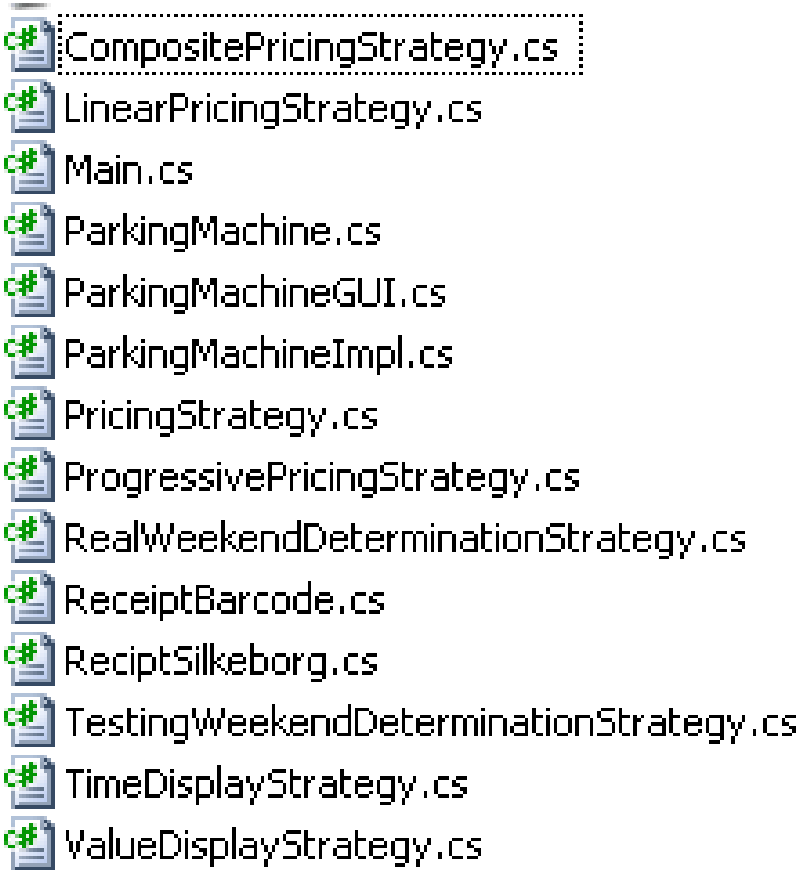
Analysis

Liabilities

The number of classes in action ☹

On the other hand:

- careful naming makes it possible to quickly identify which class to change…



Analysis

Liabilities

- Actually I have a combinatorial explosion of factories! I need a factory for each and every combination of delegates that I have

- Exercise: How can I avoid this explosion?

Summary

Handle multi-dimensional variance by compositional software designs !