1 Design Patterns

A design pattern describes

- a problem that reoccurs regularly in the domain
- the core of a solution to this problem, such that one can reuse the solution in other contexts (might not be exactly the same)

Template Method Pattern

Implements an algorithm in a manner that allows adaptation to different implementations.

- Define skeleton algorithm, but defer implementation of some concrete parts to subclasses
- Often used in frameworks and APIs

Some benefits:

- Separation of variant and invariant parts
- Avoidance of unnecessary code duplication
- Control of subclass extensions

Design Pattern Template

1. - Name: Short mnemonic to extend the design vocabulary
   - Intent: Goals and reasons why to use the pattern

---

2. - Motivation: States problem situation
   - Applicability: Context in which the pattern can be used

---

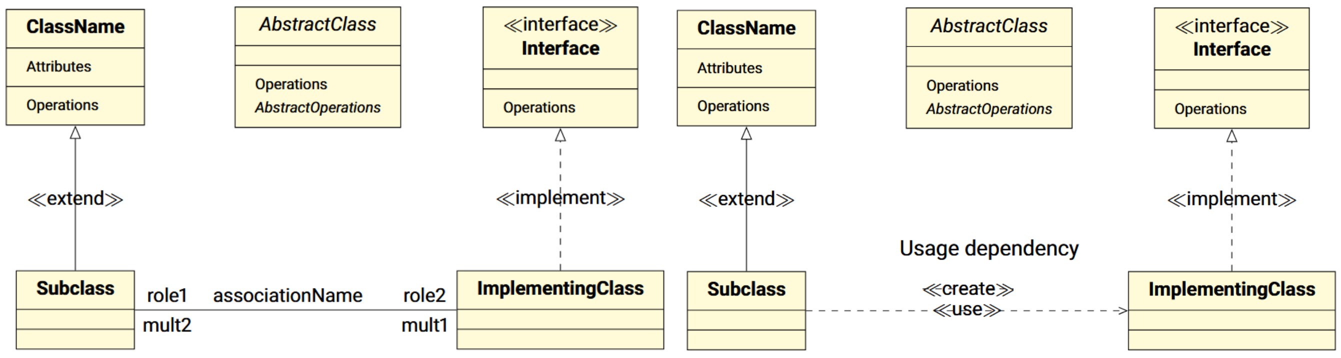
3. - Structure: Static structure of the pattern (UML Class diagram)
   - Participants: Which classes are involved
   - Collaborations: How the classes interact
   - Implementation: How to implement the pattern

---

4. - Consequences: Gains and trade-offs

---

5. - Known Uses: Examples of using the pattern
   - Related Patterns: References to and discussion of related patterns



1.1 Subject - Observer Pattern

The subject - observer pattern utilizes Object-Oriented Analysis and Design (OOAD). OOAD organizes a problem by breaking it down into managable subtasks Objects are responsible for subtasks. Collaboration might be required for complex problems.

Advantages:

- Easy to understand, implement, understand, maintain and reuse objects - Divide and Conquer approach possible
- Flexible combinations for different problems possible

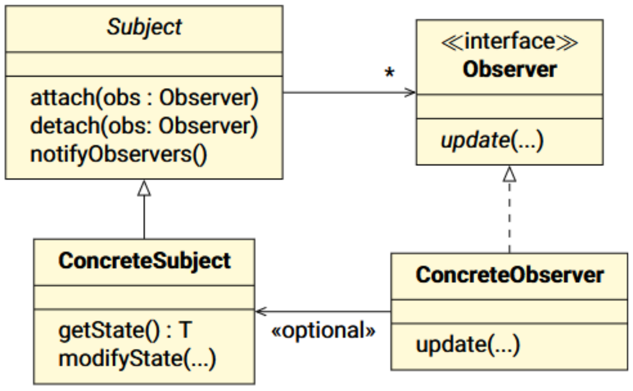
Disadvantages:

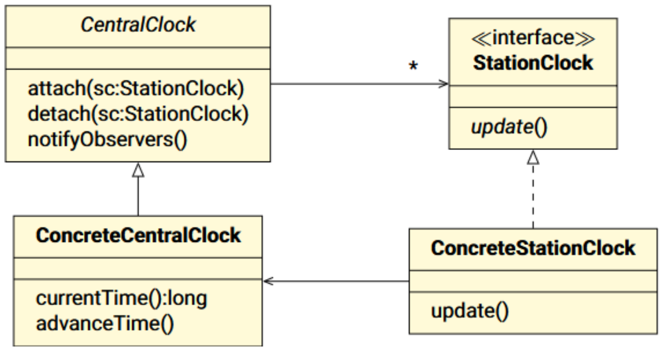
- Behaviour is distributed across objects Lack of clarity of design
- Any state change might affect others

Hereby the communication of the objects is desirable to be done with low coupling, as change in one object should not necessarily result in change in another. This way the objects can be reused in different context.

1.1.1 Subject - Observer Pattern: Pull Mode

Subject is an object with state changes that are independent of the observers. The observers get information from the subject and react accordingly.





- Subject should not need to know details about its observers
- Identity and number of observers not predetermined or fixed
- New observers can be added dynamically

Observer Pattern - Advantages

- Abstract coupling between **subject** and **observer**
- Support for broadcast communication
  - Sender does not know the type of the receiver

Observer Pattern - Disadvantages

- Danger of update cascades from observers to their dependant objects
- Update sent to all observers - might not be interesting to all observers
- No change details - Observers need to find out what has changed
- Uniform interface for all observer updates - Subject cannot send optional parameters to observers

Example: Car Inventory

```java
class CarInventoryView implements Observer {
    @Override
    public void update(Subject subject) {
        fillTable((CarInventory) subject);
    }
}
```

1.1.2 Subject - Observer Pattern: Push Mode

Instead of just passing the current state of the subject to every observer additionally also provides the change in the state. Observers can then handle their actions according to the change and do not necessarily need to access the state of the subject

Example: Car Inventory

```java
class CarInventoryView implements Observer {
    @Override
    public void update(Subject subject, Change change) {
        if(change.getKind() == Change.CarDeletion) {
            deleteRowForCar(change.getCar());
        } else(change.getKind() == Change.CarAddition) {
            ...
        }
    }
}
```

1.1.3 Subject - Observer Pattern: Interest Mode

When registering observer to subject, specify what kind of updates the observer is interested in. This way the observer doesn't have ot check whether the update is of interest or not, and can skip right to handling it. This, of course, means that the subject has to handle more as it can't just send out a pure update notification.

For example: Java Action Listeners, Mouse Listeners etc.

1.2 Factory Method

In a framework that needs to be able to present multi-format documents like PDF, HTML, Word etc. the framework should be able to do so, while offering common functionality. (open, close, save, print, etc.)

This can essentially be done by letting these different classes implement a common interface, but leave instantation of a specific class to the factory.

Example: Factory Method

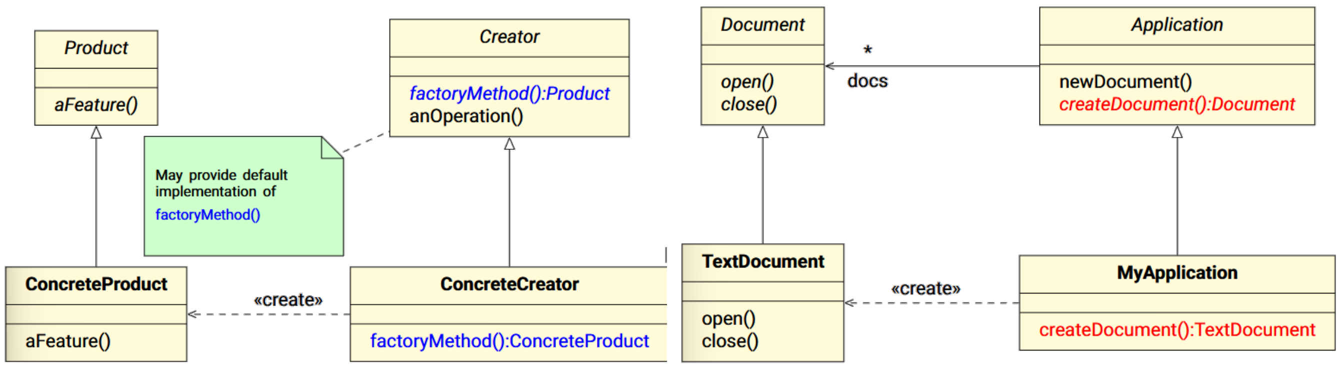
Abstract:

```java
public abstract class Document {
    public void open();
    public void close();
    ...
}

public abstract class Application {
    private List<Document> docs = new ArrayList
    <>();

    public void newDocument() {
        Document doc = createDocument();
        docs.add(doc);
        doc.open();
    }
    public abstract Document createDocument();
}
```

Concrete:

```java
public class TextDocument extends Document {
    ...
}

public class MyApplication extends Application {
    public Document createDocument() {
        return new TextDocument();
    }
}
```

The creator can also be implemented concretely, providing a reasonable default implementation.



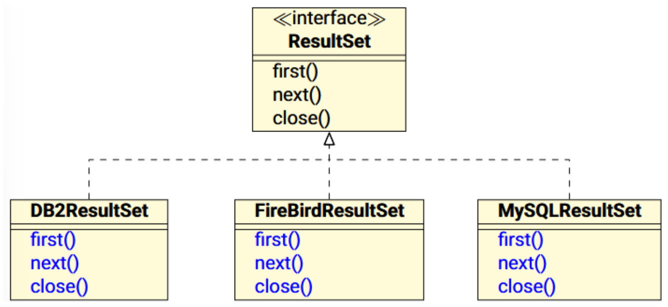
Factory Method - Consequences

- Client application code only knows product interface Works for any ConcreteProduct
- Product provides a hook for subclasses Extended version of object via hook

1.3 Abstract Factory

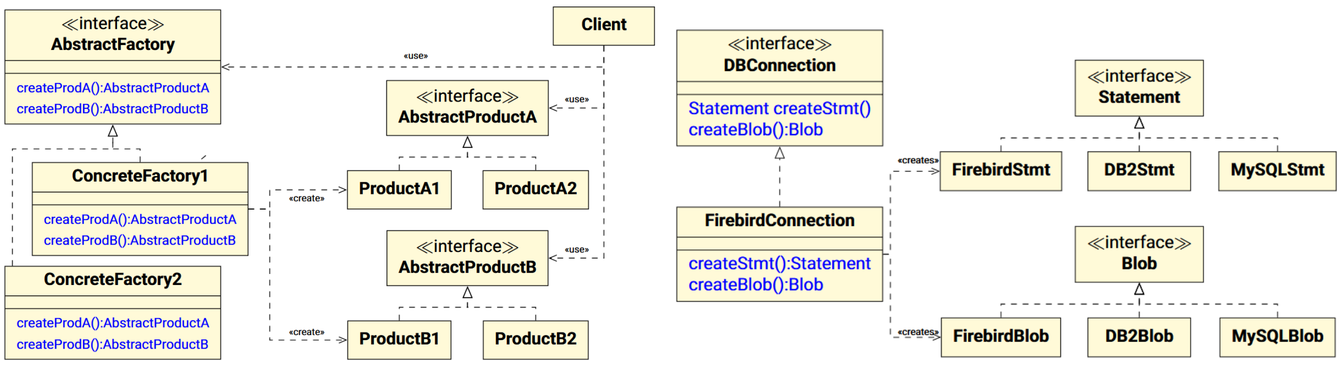
Abstract Factories provide a system to create a family of related classes that implement common interfaces.

For example, taking a search engine that needs to search for a query in multiple databases of different formats:



This creates a problem of that this usually needs to still have concrete implementations of the subclasses as the class hierarchy might differ and the interacting code might be different. Additionally, this method requires that the format is specified at the time of creation, which is undesirable.

The solution is to have an interface, the abstract factory, that is used to instantiate the concrete factories, which in turn can then be used to create the concrete classes.



Abstract Factory - Advantages

- Abstracts concrete products - Client is unaware of the concrete product they're using
- Changing formats/families is easy
- Consistency among products

Abstract Factory - Disadvantages

- Adding unforseen additional products is expensive - Abstract family and all its subclasses need to be changed
- Object creation followa non-standard pattern - Factory instead of constructor

1.4 Factory Method vs. Abstract Factory

	Factory Method	Abstract Factory
Product	Single	Family
Product declaration	Client	Startup
Product exchange	Concrete Product	Abstract Product
Creation	Local (Creator)	Selection of Factory