More Patterns

Mikael Svahnberg[*]

2023-10-06

Contents

[*] Mikael.Svahnberg@bth.se

1 Introduction

- Freeman & Robson, Chapter 4: The Factory Pattern
- Freeman & Robson, Chapter 7: The Adapter and Facade Pattern
- Freeman & Robson, Chapter 9: The Iterator and Composite Pattern
- Design Principle: *Open-Closed Principle*
- Design Principle: *Depend on Abstractions*
- Design Principle: *Encapsulation*
- Design Principle: *High Cohesion*

2 Referencing vs Creating Objects

- Somewhere in your code you are going to need to create objects
- Otherwise, you are not writing an *object*-oriented program
- `new` always expects a concrete class.
- But you may *store* it in a reference to an abstract base-class (or `interface`)
  - In fact, you *want* to use a reference to an interface
  - Referencing a concrete sub-class means we have to *modify* the code when we *extend* with a new type.

Design Principle: *Open-Closed Principle*

- A class should be *open* for extension
  - We can add or modify behaviour e.g. through inheritance
- A class should be *closed* for modification
  - modify the code → original tests no longer apply
  - modify the code → break other parts of the system

3 Referencing vs Creating Objects

- In order to *do* something, we first need to create an object

```
public Document generateDocument(String documentType) {
  Document theDocument;

  if (documentType.equals("Letter")) {
    theDocument = new LetterDocument();
  } else if (documentType.equals("Report")) {
    theDocument = new ReportDocument();
  } else if (documentType.equals("Protocol")) {
    theDocument = new ProtocolDocument();
  }

  theDocument.applyFormatting();
  theDocument.setHeaders();
  theDocument.setFooters();
  theDocument.insertSectionHeadings();
  theDocument.insertLoremIpsum();

  return theDocument;
}
```

- this method does two things:
  - create an object
  - generate a document.
- Breaks the Design Principle: *High Cohesion*
- Breaks the Design Principle: *Encapsulation*
- We *do* (probably) use an `interface Document`, which is good.

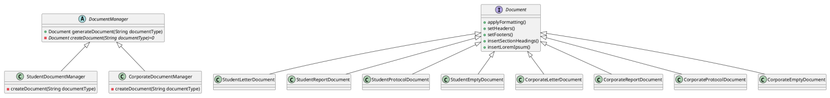
4 Factory Method

- Design Principles: *High Cohesion* and *Encapsulation*
  - Separate two tasks into two methods
  - Encapsulate the creation of objects in a separate method
  - A new documentType now only cause modification in one place: `createDocument()`
  - `generateDocument()` becomes
    - *Open for Extension* by not being aware of the sub-classes to Document.
    - *Closed for Modification* since it is no longer needed.

```
public Document createDocument(String documentType) {
  if (documentType.equals("Letter")) {
    return new LetterDocument();
  } else if (documentType.equals("Report")) {
    return new ReportDocument();
  } else if (documentType.equals("Protocol")) {
    return new ProtocolDocument();
  } else {
    return new EmptyDocument();
  }
}

public Document generateDocument(String documentType) {
  Document theDocument = createDocument(documentType);
  theDocument.applyFormatting();
  theDocument.setHeaders();
  theDocument.setFooters();
  theDocument.insertSectionHeadings();
  theDocument.insertLoremIpsum();

  return theDocument;
}
```

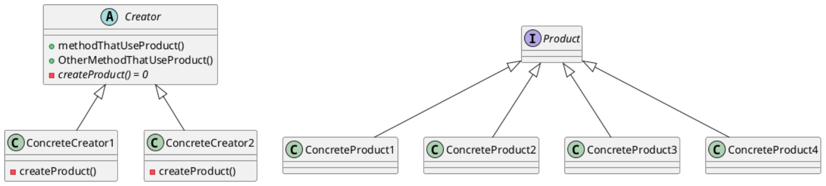
5 Using Different Factory Methods

- Sometimes, you need to add an extra layer of qualifiers.
  - For example, let's say that *Students* documents are different from *Corporate* documents.
- We could add an extra parameter to `createDocument()`
  - … but we would need to add this to `generateDocument()` as well.
  - `createDocument()` becomes more complicated
  - `createDocument()` now does two things:
    - Decide which Document type is requested
    - Decide whether it is for Students or Corporate.
  - Add another role, e.g. *Administrators*, and we need to modify `createDocument()`.
- Or, *we can trust the compiler*


A DocumentManager
Document generateDocument(String documentType)
Document createDocument(String documentType)=0
C StudentDocumentManager
createDocument(String documentType)
C CorporateDocumentManager
createDocument(String documentType)
I Document
applyFormatting()
setHeaders()
setFooters()
insertSectionHeadings()
insertLoremIpsum()
C StudentLetterDocument
C StudentReportDocument
C StudentProtocolDocument
C StudentEmptyDocument
C CorporateLetterDocument
C CorporateReportDocument
C CorporateProtocolDocument
C CorporateEmptyDocument

- Decide once which type of `DocumentManager` we want
- The concrete document managers (`StudentDocumentManager` or `CorporateDocumentManager`)
  - Knows how to *create* documents of the right type
  - Does not know how to work with the document; this is still done by the `DocumentManager`

6 Summary of Factory Method



7 Depend on Abstractions

Depend upon abstractions. Do not depend upon concrete classes.

- In other words:
  - Refer to an *interface* whenever possible
  - Keep creation of objects in specific easy-to-locate places

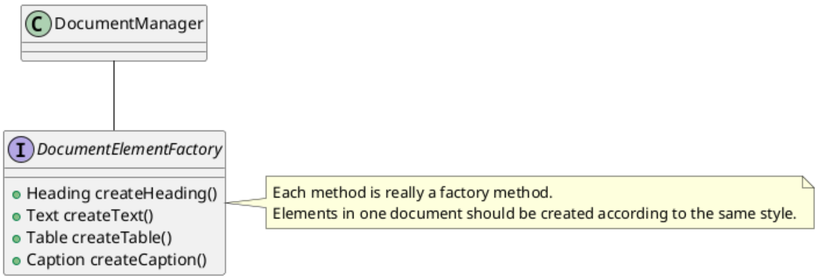
8 Factories for Related Choices: Abstract Factory

- Factory Method to create *one type* of objects (with a shared base class)
- Several types → several factory methods
- Breaks Design Principle: *High Cohesion*
  - The class has two responsibilities:
    - Whatever it is supposed to be responsible for, e.g. managing documents
    - Keeping track of many possible object types and which object type that applies in each situation
  - one single factory method was an ok compromise
  - several factory methods begin to get messy, and overshadow the real intention of the class.

So.
We delegate.

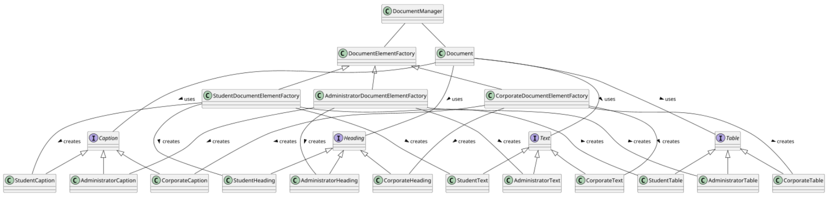
9 Example: Document Elements

- Different elements: Headings, text, tables, figure captions, etc.
- Different styles for each role: Student, Corporate, Administrator, etc.



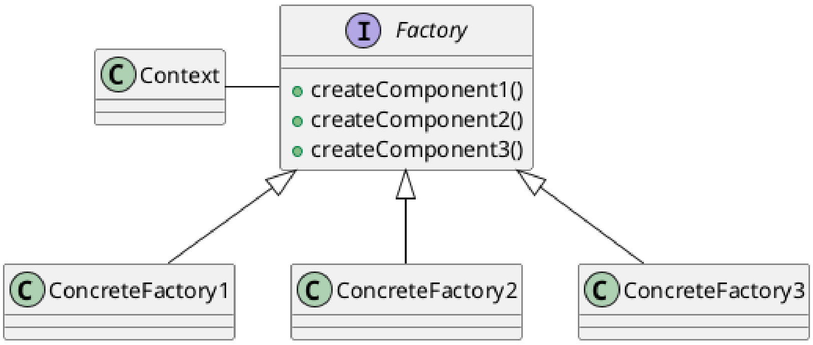
```
public class DocumentManager {
  private DocumentElementFactory docFactory;
  public void setUserRole(String userRole) {
    // This is still a factory method
    // but is only used to create the right factory

    if (userRole.equals("Student")) {
      docFactory = new StudentDocumentElementFactory();
    } else if (userRole.equals("Corporate")) {
      docFactory = new StudentDocumentElementFactory();
    } /* ... and so on for the rest of the roles */
  }


  public void insertHeading(String contents) {
    myDocument.insert(docFactory.createHeading(contents));
  }

  public void insertText(String contents) {
    myDocument.insert(docFactory.createText(contents)); // Note that we use the same docFa
                                                        // so that the text style matches
  }

  /* ... and so on for all the other document elements */
}
```

10 Inherit All the Things



- Normally, we will not draw all these relations because it gets too messy.

11 Summary of Abstract Factory



- *Encapsulate what varies*: the creation of objects
- *Program to Interfaces*: Use references to the abstract interfaces, delegate creation to factories.
- *Loose coupling*: The users of a factory do not know which concrete subclass that is used for creation.
- $\sum$ *Depend on Abstractions*: We use interfaces to create a loosely coupled design.

12 Adapter

- Sometimes, parts of the code is outside our control
  - We may, for example, use a third party library
  - Or, the code may be developed by a separate team
- *If we are lucky…*
  - there are stable and well defined interfaces to use
  - we can work together to define those interfaces
  - the interfaces fit with how we want to use the code
- Otherwise, our code will need to know to things:
  - What it is supposed to do
  - How to deal with changing or unsuitable interfaces.

For this, we use the *Adapter Design Pattern.*

- The Adapter defines a stable interface
- We can program to an interface – the adapter – and not the implementation.

13 Adapter Defined

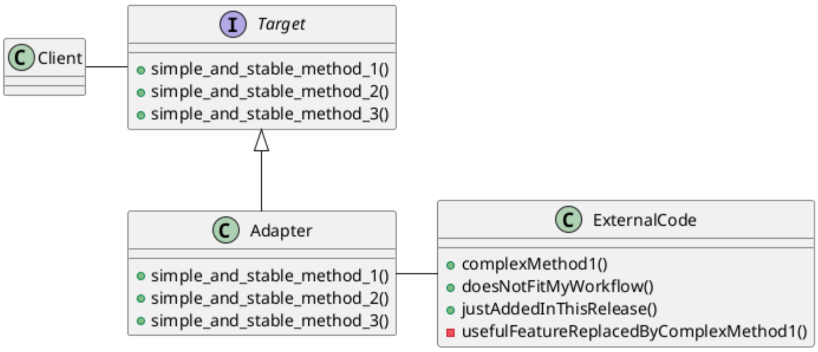
Client All of our system

Target Defines the interface we want to use

Adapter Translates from our interface `Target` to the actual interface `ExternalCode`

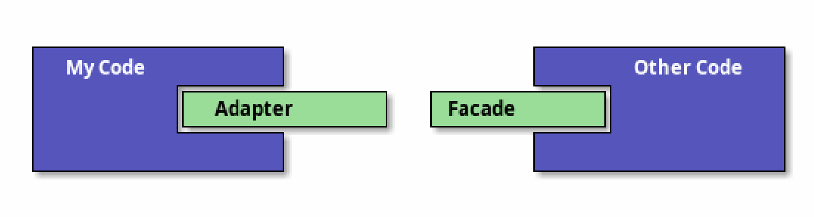
ExternalCode Whatever we wanted to adapt, e.g.

- A single class
- Several classes
- A Subsystem or a Package
- a REST API on some other server


Client
Target
simple_and_stable_method_1()
simple_and_stable_method_2()
simple_and_stable_method_3()
Adapter
simple_and_stable_method_1()
simple_and_stable_method_2()
simple_and_stable_method_3()
ExternalCode
complexMethod1()
doesNotFitMyWorkflow()
justAddedInThisRelease()
usefulFeatureReplacedByComplexMethod1()

14 Facade

- We write a adapters to make it easier to use other code
- We provide a *facade* with an easy-to-use interface into our code
  - *Encapsulate* our code structure
  - c.f. the `public` declaration in a class



A subsystem:

- Can still expose other classes if someone needs them
- May have several facades, e.g.
  - one per user role
  - one per use case
- With a good enough facade, users of our code may not need to write an adapter

15 Adapter, Facade, and Decorator

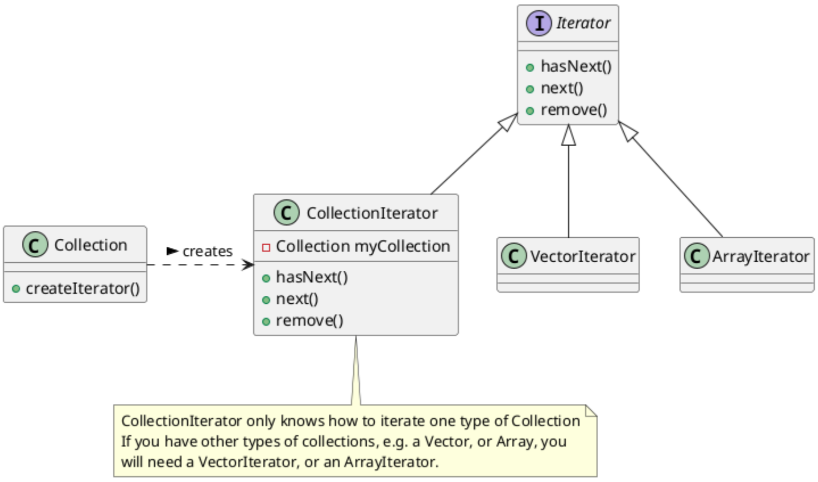
- Adapter *alters* an interface to make it stable and better fit our needs
- Facade provides a *simplified interface* to make a subsystem easier to use
- Decorator *adds functionality* to a lower-level interface
- Design Principle: *The Principle of Least Knowledge*

- “Talk only to your immediate friends”
  - Use the adapters/facades, and try not to dig deeper.
  - Avoid writing methods that expose internal design
    - e.g. return an object that is only used to find another object, etc.

```
myCustomers->findCustomer(theCustomerName)
->getTickets()->filterByTime("today")->first()->getDetails();
```

16 Iterator

- A collection should only do one thing: collect the elements.
- As a user we do not need to know *how* it collects the elements
  - ArrayList<>
  - Array
  - Vector
  - Set
  - Bag
  - Dictionary
  - Tree
  - HashMap
  - ...
- We may need to access all elements in some order
- We want to do this without knowing anything about the collection’s internal structure



CollectionIterator only knows how to iterate one type of Collection
If you have other types of collections, e.g. a Vector, or Array, you
will need a VectorIterator, or an ArrayIterator.

17 Using Iterators

```
// Lists (of objects) have an iterator
List<Integer> intList = new ArrayList<Integer>();
Iterator<Integer> iter = intList.iterator();
while(iter.hasNext()) {
  Integer element = iter.next();
}

for(Integer element : intList) { /* ... */  }

// As to other data structures, e.g. Maps
Map<String, Integer> stringHash = new HashMap<String, Integer>();
Iterator<String> stringIter = stringHash.keySet().iterator();
while(stringIter.hasNext()) { String element = stringIter.next(); }

iter = stringHash.values().iterator();
while(iter.hasNext()) { Integer element = iter.next(); }

// But what about an intArray?
// Arrays do not provide an iterator
// and especially not one for built-in datatypes.
int [] intArray = new int[10];

// We could use the streams API to convert
// our int[] to a List<Integer>
iter = IntStream
  .of(intArray)                // start with the int array
  .boxed()                     // Convert to a stream of Integers
  .collect(Collectors.toList()) // collect it as a List<Integer>
  .iterator();                 // and get the iterator.

while(iter.hasNext()) {
  Integer element = iter.next();
  System.out.print(element.toString());
}

// But if we are already using streams, why not jump full in?
IntStream.of(intArray).forEach( (elem) -> { System.out.print(elem); } );
```

18 Summary

- Design Principle: *Open-Closed Principle*
- Design Principle: *Depend on Abstractions*
- Design Principle: *Encapsulation*
- Design Principle: *High Cohesion*
- A *Strategy* for creating objects: *Abstract Factory* Design Pattern

- At the very least, *Encapsulate* what may chage: *Factory Method*
- More on *Encapsulate* what may change: *Adapter* Design Pattern
- Using *High Cohesion* to focus on one thing only: *Facade* Design Pattern
  - Focus on the domain, or
  - Focus on the software design
- *High Cohesion* to split responsibilities:
  - Maintaining a collection of objects (possibly from an inheritance hierarchy)
  - Iterating over the collection: *Iterator* Design Pattern