CS 520/620
Advanced Software Engineering
Fall 2016

September 27, 2016
Recap

● Behavioral patterns
  ○ Strategy pattern
  ○ Observer
  ○ Iterator

● MVC revisited
  ○ Design patterns commonly used in an MVC architecture
Recap: Strategy pattern

<<interface>>
Median
+median(a:double[]):double

StrategyMedian
-sortStrategy:Sorter
+median(a:double[]):double +setSorter(s:Sorter)

<<interface>>
Sorter
+sort(array:double[])
Recap: Strategy pattern

“median” delegates the sorting of the array to a “sortStrategy”, which can be configured and changed at run time.
Recap: Observer pattern and Iterator pattern

Observer pattern

- Models a one-to-many dependency: one observable subject and many observers.
- Decoupling of state and action: Notify registered observer(s) about state change.

Iterator pattern

- Provides uniform (sequential) access to a data structure.
- Does not reveal implementation details.
MVC revisited

Design patterns in an MVC architecture

- Client sees
- Client uses
- Model updates
- Model manipulates
- View
- Controller
- Composite
- Observer
- Iterator
- Strategy
Today

- THE behavioral pattern: visitor pattern
- Decoupling object creation from object uses
- Creational design patterns:
  - Factory method
  - Singleton
  - Abstract Factory
  - Builder
Visitor pattern: motivation

Scenario

- Complex data structure
- Different types of elements
- Many different operations on the data structure
Visitor pattern: motivation

Scenario
- Complex data structure
- Different types of elements
- Many different operations on the data structure

Goal
- Adding new operations should be easy
- Allow the definition of an operation independently of
  - the concrete data structure
  - the data structure traversal

Example: Operations on an abstract syntax tree
Visitor pattern
Visitor pattern

```
Visitor
  +visitElemA(elemA)
  +visitElemB(elemB)

Element
  +accept(visitor: Visitor)

Concr. Elem. A
  +opA()
  +accept(visitor: Visitor)
  ...
  visitor.visitElemA(this);
  ...

Concr. Elem. B
  +opB()
  +accept(visitor: Visitor)
  ...
  visitor.visitElemB(this);
  foreach (e : elems)
    e.accept(visitor)
  ...
```
Visitor pattern

- **Visitor**
  - «interface»
  - +visitElemA(elemA)
  - +visitElemB(elemB)

- **Concr. Visitor 1**
  - +visitElemA(elemA)
  - +visitElemB(elemB)

- **Concr. Visitor 2**
  - +visitElemA(elemA)
  - +visitElemB(elemB)

- **Concr. Element A**
  - +opA()
  - +accept(visitor: Visitor)
  - ...
  - visitor.visitElemA(this);
  - ...

- **Concr. Element B**
  - +opB()
  - +accept(visitor: Visitor)
  - ...
  - visitor.visitElemB(this);
  - foreach (e : elems)
    - e.accept(visitor)
  - ...
Visitor pattern: example

See code example (online)

- Count/print items in a shopping cart
  - The Interface Visitor defines a visit method for each type of element that can occur in the data structure.
  - The two implementations of this interface (PrintVisitor and TotalVisitor) perform their operations on the data structure without knowledge about the data structure traversal (note how the traversal is implemented in the Visitable elements Cart, Customer, and Item).

Use ant to compile and run the code:

$ant -p => list all targets
$ant compile => compile the code
$ant run => run the Main class
Visitor pattern: summary

Decoupling of data structure and operations

Pros

● Adding new operations (visitors) is easy

Cons

● Adding a new visitable element is hard
Object creation challenges
Object creation challenges

```java
public class MyClass {
    public void do() {
        Product product = new Product();
        product.getPrice();
    }
}
```
Object creation challenges

```java
public class MyClass {
    public void do() {
        Product product = new Product();
        product.getPrice();
    }
}
```

What if MyClass should not have control over how product is instantiated?

How to decouple the creation of an object from its uses?
Factory method pattern

```
... product = factoryMethod();
...
return new SubProduct();
```
Factory method pattern

“Alternative constructor” or “virtual constructor”
  • Can be implemented as a class or object method
  • Singleton pattern is a special case

Pros
  • Flexibility: no hard-coded constructor calls
  • Readability: allows descriptive names

Cons
  • Needs proper documentation (e.g., naming conventions: getInstance, createXYZ, constructXYZ, …)
Factory method pattern

“Alternative constructor” or “virtual constructor”
- Can be implemented as a class or object method
- Singleton pattern is a special case

Pros
- Flexibility: no hard-coded constructor calls
- Readability: allows descriptive names

Cons
- Needs proper documentation (e.g., naming conventions: getInstance, createXYZ, constructXYZ, …)

What about factory methods for related objects?
Abstract factory pattern
Abstract factory pattern: the products
Abstract factory pattern: the factory

```
AbstractFactory
    +newWindow(): Window
    +newPanel(): Panel

UIFactory
    +newWindow(): Window
    +newPanel(): Panel

WebFactory

SwingFactory
```
Abstract factory pattern
Abstract factory pattern: summary

Generalization of factory methods

Pros

● Adding a new family of products is easy
● Consistency: clients use products of a single family

Cons

● Adding a new product is hard
Object creation challenges cont.

Example application: typesetting/text conversion

Scenario
- Fixed set of types of content (header, paragraph, ...)
- Multiple output formats (html, latex, ...)

Goal
- Different representations of the same input

How to separate the internal representation from the output format?
Builder pattern

```java
interface Builder
{
    +convertTitle(title: String)
    +convertText(text: String)
}

class HtmlBuilder extends Builder
{
    +convertTitle(title: String)
    +convertText(text: String)
    +getHtml(): String
}

class LatexBuilder extends Builder
{
    +convertTitle(title: String)
    +convertText(text: String)
    +getLatex(): String
}

class Director
{
    +build()
}

... builder.convertTitle("Title");
... builder.convertText("...");
... builder.convertText("...");
...
Builder pattern: example

See code example (online)

- Typesetting/text conversion
  - The Director defines how the document should be typeset but it doesn’t encode a specific output format. Note how a new output format can be defined without changing the Director.
  - The interface Builder provides a method for each typesetting feature.
  - The two implementations of this interface (HtmlBuilder and LatexBuilder) encode the output format for the typesetting features.

Use ant to compile and run the code:

```bash
$ant -p => list all targets
$ant compile => compile the code
$ant run => run the Main class
```
Builder pattern: summary

Decoupling of internal representation and output format

Pros
  ● Adding a new format is easy

Cons
  ● Adding a new type (internal representation) is hard
Design patterns: summary

Pros

● solve a recurring, common design problem
● provide a generalizable solution
● provide common terminology
● “toolbox” for novice developers

Cons

● risk of over-engineering
● impact on complexity and system performance
Design patterns: categories

1. Structural
   - Composite
   - Decorator
   - ...

2. Behavioral
   - Template method
   - Strategy
   - Visitor
   - ...

3. Creational
   - Factory (method)
   - Builder
   - ...
Course overview: the big picture

● **Software architecture and design**
  ○ Software modelling and UML crash course.
  ○ Best practices and OO design principles.
  ○ Architecture and Design patterns.

● **Empirical Software Engineering**
  ○ Reasoning about experimental designs and studies.
  ○ Understanding and reasoning about threats to validity.

● **Software testing and debugging**
  ○ Learning about cutting-edge research.
  ○ Hands-on experience, using testing and debugging techniques.

● **Class project**
  ○ Empirical study, development of a research prototype, etc.