Breaking the Patterns: Extending without Class Inheritance, Reflection, AntiPatterns

Alexandru Olteanu

Universitatea Politehnica Bucuresti
Facultatea de Automatică si Calculatoare, Departamentul Calculatoare
alexandru.olteanu@upb.ro

OOP, 2020

Universitatea
Politehnica
Bucureşti

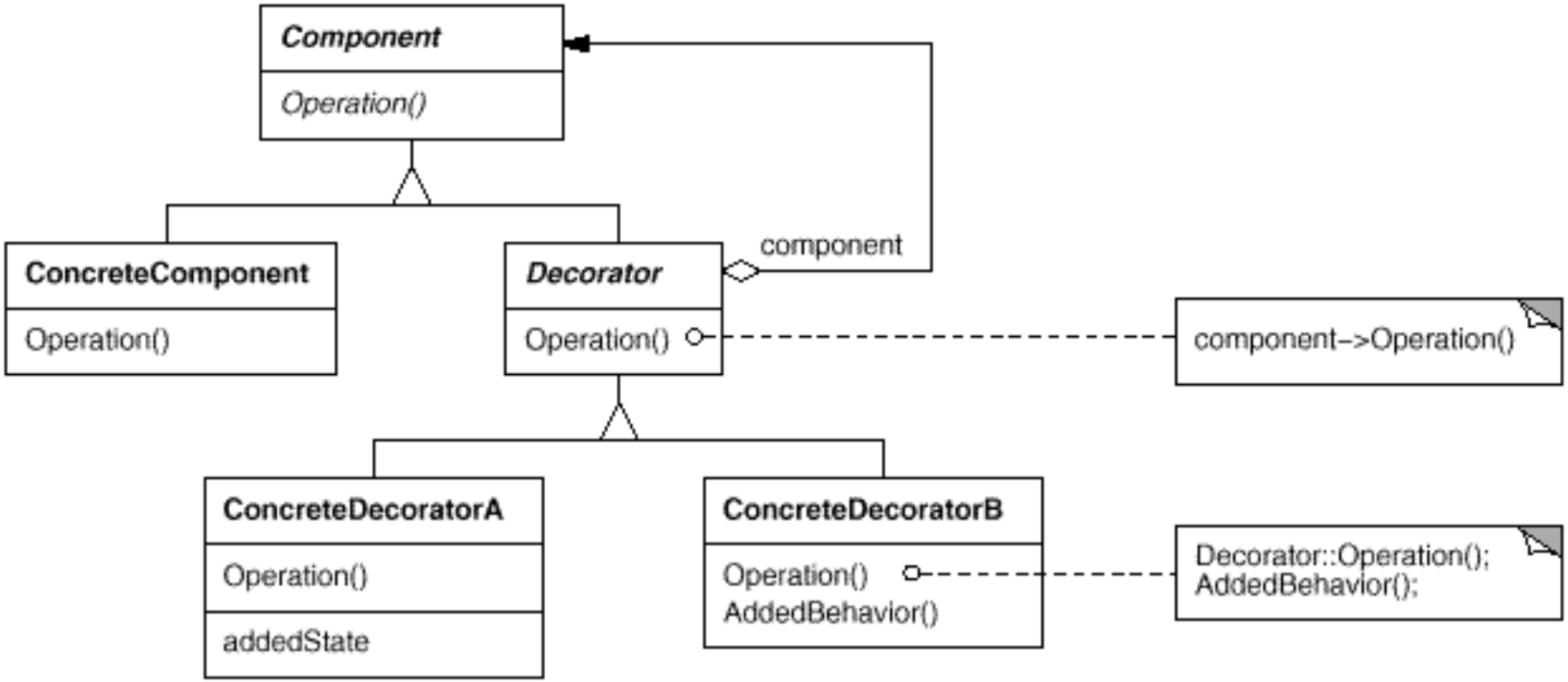
Subiecte de discutie

- Extending without Class Inheritance
- Reflection
- AntiPatterns

Extending without Class Inheritance: Decorator, Prototypal Inheritance

Decorator

Decorators provide a flexible alternative to subclassing for extending functionality (attaching additional responsibilities to an object dinamically).



Decorator

"Changing the skin of an object [...Decorator...] versus changing its guts [...Strategy...]"
- Gang of Four, Design Patterns

Prototypal Inheritance

Prototypal Inheritance objects inherit directly from other objects. Various ways to implement:

- Concatenative Inheritance
- Prototype Delegation
- Functional Inheritance

"Favor object composition over class inheritance", see the Fragile Class Problem.

▸ Master the JavaScript Interview: What's the Difference Between Class & Prototypal Inheritance?

Reflection, Introspection

Introspection

In computing, Type Introspection is the ability of a program to examine the type or properties of an object at runtime. Some programming languages possess this capability.

In Spring, Introspection is a process of analyzing a Bean to determine its capabilities

```
final BeanWrapper wrap = new BeanWrapperImpl(AType.class);
final PropertyDescriptor[] pDescriptors =
  wrap.getPropertyDescriptors();
for (final PropertyDescriptor pDescriptor : pDescriptors) {
   logger.info(pDescriptor.getName() + ":" +
      pDescriptor.getPropertyType());
}
```

Reflection vs Introspection

Introspection should not be confused with Reflection, which goes a step further and is the ability for a program to manipulate the values, meta-data, properties and/or functions of an object at runtime. Some programming languages possess that capability.

▸ detalii

In general, aceasta capabilitate este oferita de o suita de clase, care impreuna formeaza Reflection API, si are efecte asupra diferitelor tool-uri de compilare, rulare si debug.
In Java:

```
java.lang.Class, java.lang.reflect.*
```

Reflection: Class

Primul pas este obtinerea unui obiect de tip Class pentru tipul de date pentru care dorim sa facem reflection:

- daca avem la dispozitie o instanta:

```
Class c = "foo".getClass();
```

- daca nu avem o instanta, dar stim tipul:

```
Class c = java.io.PrintStream.class;
```

- ▸ Toate modalitatile de a obtine obiecte Class

Reflection: Exemple

- doresc sa apelez metoda 'met' a unui obiect oarecare 'obj', daca aceasta exista

```
Method method = obj.getClass().getMethod("met", null);
method.invoke(obj, null);
```

- crearea de obiecte noi si schimbarea valorilor pe care le au campurile ▸ Creating New Class Instances si ▸ Setting Field Values
- JUnit foloseste Reflection pentru a inspecta codul testelor, a observa ce metode au anotarea @Test si a le apela automat pe acestea.
- compatibilitatea unei aplicatii Android cu mai multe versiuni de Android OS ▸ Backward compatibility for Android applications

Reflection: Avantaje si Dezavantaje

Avantaje

- Extensibility Features
- Backwards Compatibility for Android OS ▸ detalii
- Class Browsers and Visual Development Environments
- Debuggers and Test Tools

Dezavantaje

- Performance Overhead
- Security Restrictions
- Exposure of Internals

▸ Tutorial complet

Reflection vs Encapsulation

Exemples of uses that do not break encapsulation:

- finding out members, annotations, super-types of a class
- invoke accessible methods, modify accessible fields

Exemples of uses where it might be acceptable to break encapsulation:

- simplest way to implement a unit test (on someone else's code)
- work around a bug (in someone else's code)

▸ Doesn't Reflection API break the very purpose of Data encapsulation?

Reflection vs ?

Reflection vs Generics

You cannot create an instance of a type parameter.

```
public static <E> void append(List<E> list) {
    E elem = new E();  // compile-time error
    list.add(elem);
}
```

As a workaround, you can create an object of a type parameter through reflection:

```
public static <E> void append(List<E> list, Class<E> cls)
    throws Exception {
    E elem = cls.newInstance();   // OK
    list.add(elem);
}
```

▸ Restrictions on Generics

Reflection vs Singleton

Using Reflection to overcome a typical implementation of Singleton.

```
Constructor[] constructors =
        Singleton.class.getDeclaredConstructors();
for (Constructor constructor : constructors)
{
    // Below code will destroy the singleton pattern
    constructor.setAccessible(true);
    instance2 = (Singleton) constructor.newInstance();
    break;
}
```

As a workaround, you can use Enums, which Java ensures are instantiated only once, but do not allow lazy initialization.

AntiPatterns

Ce sunt AntiPatterns?

AntiPatterns are

- patterns: describe general solutions to problems that occur over and over again
- bad ones: usually ineffective and risk being highly counterproductive

God Object

A God Object is an object that knows too much or does too much:

▸ God Object

Spaghetti Code

Spaghetti Code is object-oriented code written in procedural style, such as by creating classes whose methods are overly long and messy:

- un main de 2500 de linii de code, anyone?

▸ Spaghetti Code

Boilerplate Code

Boilerplate Code refers to sections of code that have to be included in many places with little or no alteration - e.g. classes are often provided with methods for getting and setting instance variables

"programs should be written for people to read, and only incidentally for machines to execute."
- Harold Abelson

Boilerplate Code

Boilerplate Code refers to sections of code that have to be included in many places with little or no alteration - e.g. classes are often provided with methods for getting and setting instance variables

Solutions:

- metaprogramming (which has the computer automatically write the needed boilerplate code or insert it at compile time)

```
public string Name { get; set; }
  //auto-implemented properties in C#
```

- convention over configuration (which provides good default values, reducing the need to specify program details in every project)

```
class ProductsController {
  public ActionResult Index() {...}
}
// when serving /products, framework uses Reflection to
    create instance and call method
```

Singleton

Me: So! Have you ever heard of a book called Design Patterns?
Them: Oh, yeah, um, we had to, uh, study that back in my software engineering class. I use them all the time.
Me: Can you name any of the patterns they covered?
Them: I loved the Singleton pattern!
Me: OK. Were there any others?
Them: Uh, I think there was one called the Visitater.
Me: Oooh, that's right! The one that visits potatoes. I use it all the time. Next!!!
I actually use this as a weeder question now. If they claim expertise at Design Patterns, and they can ONLY name the Singleton pattern, then they will ONLY work at some other company.

▸ Singleton Considered Stupid

Singleton

- *When used as global variables:* using global variables is an enemy of encapsulation because it becomes difficult to define preconditions for the client's object interface.
  - Imagine a couple of objects that use the same global variable; one triggers a side effect which changes the value of the global state, you no longer know the starting state when you execute the code in the other object, which will have unpredictable results. This is the reason why in Dependency Injection, objects are passed as arguments to constructors or by setters.

- *Singletons vs polymorphism:* Singletons tightly couple the code to the exact object type and remove the scope of polymorphism.
  - As a workaround, use Factory.

▸ When Singleton Becomes an Anti-Pattern

Factory Factory Factory

I'm currently in the planning stages of building a hosted Java web application (yes, it has to be Java, for a variety of reasons that I don't feel like going into right now). In the process, I'm evaluating a bunch of J2EE portlet-enabled JSR-compliant MVC role-based CMS web service application container frameworks.

And after spending dozens of hours reading through feature lists and documentation, I'm ready to gouge out my eyes.

.....

▸ Why I hate frameworks, Benji Smith