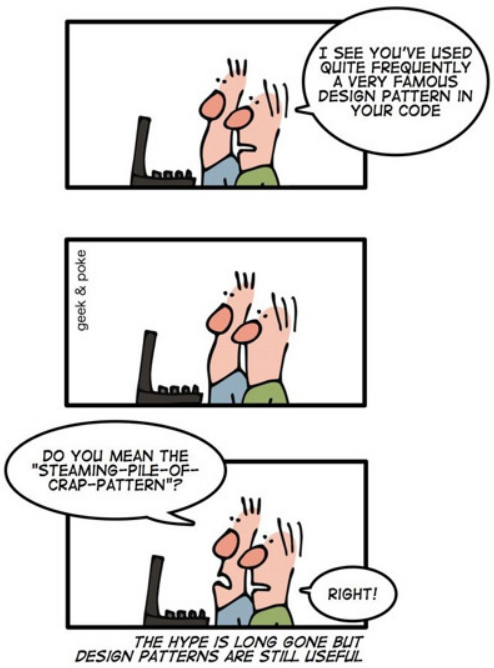


OBJECT-ORIENTED PROGRAMMING

Structural Design Patterns

Lecture #11

TECHNICAL UNIVERSITY OF KOŠICE

Stractural design patterns

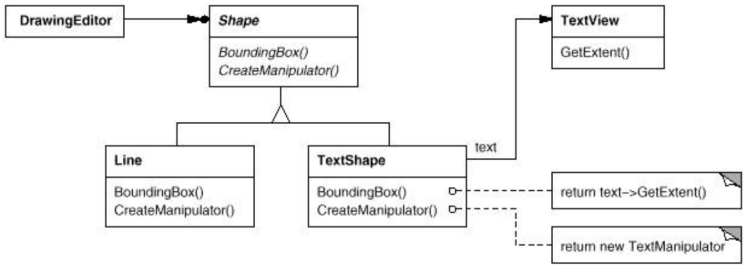
- Deal with decoupling interface and implementation of classes and objects
- Plan for today
  - Adapter
  - Composite
  - Decorator
  - Facade
  - Proxy

Adapter

- Purpose
  - Conversion of interface of the object to another interface used by the client
- Motivation
  - Sometimes we cannot use library classes because they have incompatible interface
  - We cannot change the interface because there is no source code
  - Often we cannot change interface because of other compatibility

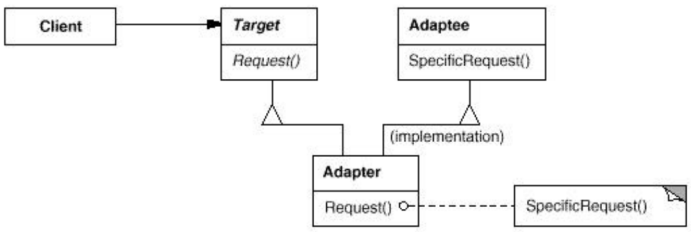
Adapter

- Motivation



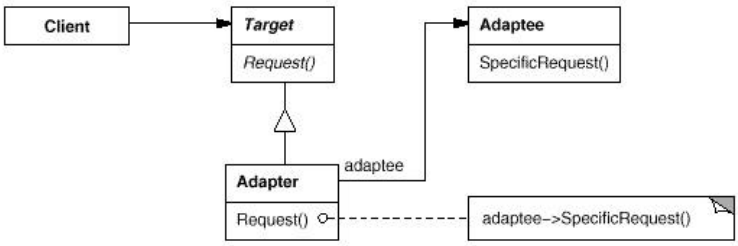
Adapter

- Structure (1st version)
  - `Adapter` class uses multiple inheritance (or multiple interfaces) to join both interfaces



Adapter

- Structure (2nd version)
  - `Adapter` object is using composition of objects



Adapter

- Implementation issues
  - How much adaptation is needed?
    - Conversion of simple interfaces where we only need to rename operations
    - Implementation of completely different set of operations
  - Does adapter support two way transparency?
    - Adapter with two way transparency implements both interfaces (`Target` and `Adaptee`)
    - Adapter can play both `Target` and `Adaptee` roles

Adapter – Example

- Let's have a class implementing a linked list of objects
- We want to use this class for implementation of stack of objects
- Operations in linked list are not compatible with operations in stack
  - We will use Adpater design pattern

Adapter – Example

- Interface of stack we want to use (it represents `Target` within the pattern)

```
public interface Stack<T> {
  public void push(T o);
  public T pop();
}
```

Adapter – Example

- Class implementing linked list (it represents `Adaptee` within the pattern)

```
public class DList<T> {
  public void insert(DNode pos, T o) { ... }
  public void remove(DNode pos) { ... }

  public void insertHead(T o) { ... }
  public void insertTail(T o) { ... }

  public T removeHead() { ... }
  public T removeTail() { ... }

  public T getHead() { ... }
  public T getTail() { ... }
}
```

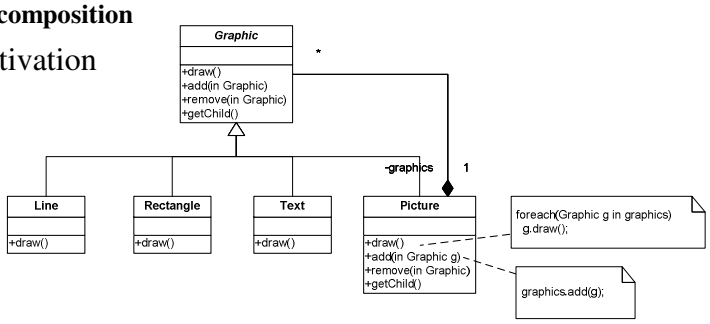

Adapter – Example

- Adapter, which utilizes (converts) operations of the linked list to operations of the stack

```
public class DListImpStack<T> extends DList<T> implements Stack<T> {

  public void push(T o) {
    insertTail(o);
  }

  public T pop() {
    return removeTail();
  }

}
```

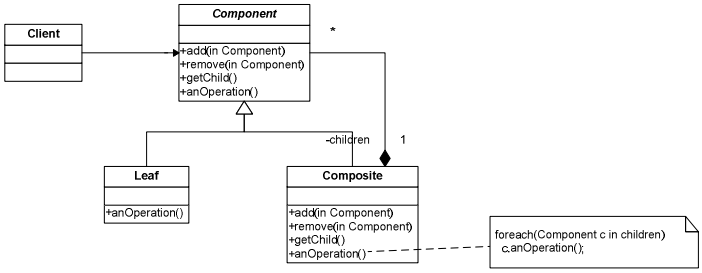
Composite

- Purpose
  - To compose the objects into a tree structures to represent hierarchies of objects
  - Allows clients to manage composed object uniformly – **recursive composition**
- Motivation



Composite

- Usage
  - When we want to represent hierarchies of objects
  - When clients want to treat composed object the same way as individual objects inside the composition
- Structure

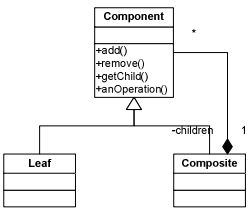


Composite

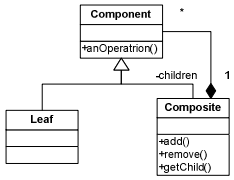
- Advantages
  - It is easy to add new types of components
  - We can implement simple clients that do not need to distinct between composed objects and components
- Implementation issues
  - Sometimes it is good to implement a reference to the parent object – it allows to apply Chain of Responsibility design pattern
  - Two approaches to implement `add()`, `remove()`, `getChild()` methods
    - **Transparent** – inside class `Component`, which allows composed object and component use the same interface
    - **Safe** – inside class `Comosite`, which does not allow clients to use components the same way as composed objects

Composite

- Transparent implementation



- Safe implementation

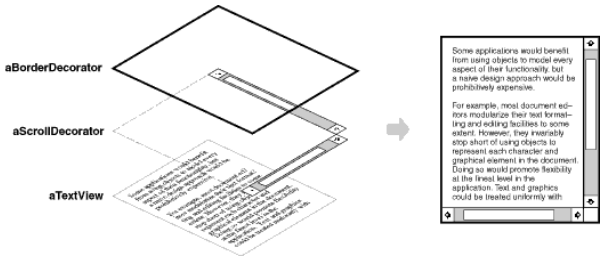


Composite

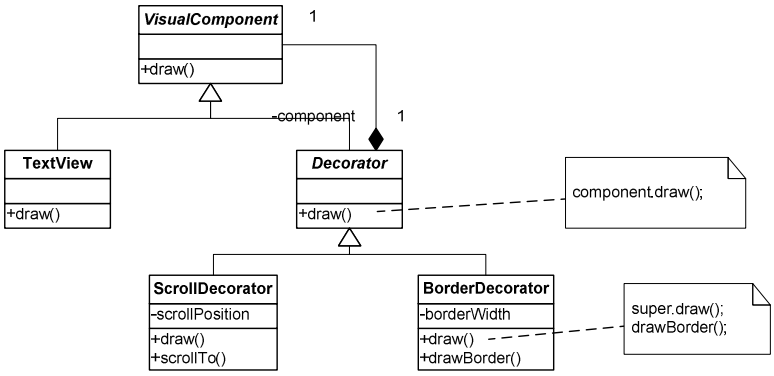
- Implementation issues
  - List of components is implemented in class `Composite` and not in class `Component` (leaf objects do not need to implement the lists)
  - Sorting of components is given by the actual application
  - When the OO language does not support **garbage collection**, we have to delete unused component objects from the memory
  - Implementation of the composition (list) is given by the actual application (array, linked list, etc.)

Decorator

- Purpose
  - To dynamically add new functionality to an object. This is flexible alternative to class inheritance
- Motivation
  - When working with document object, we want to add more (GUI) functionality e.g. frame, scrollers. We cannot use inheritance, we want it dynamically during run-time



Decorator

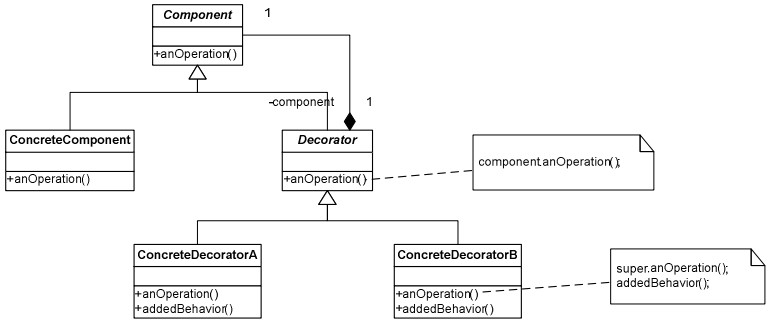


Decorator

- Usage
  - When we want to add new functionality to the object dynamically without its modification
  - Implementation using inheritance is not practical, because we often have to add many various extensions which leads to many various subclasses. Definitions of such classes are usually hidden and we cannot derive a new subclass

Decorator

- Structure



Decorator – Example

- IO classes in Java use Decorator design pattern
- Core IO classes are `InputStream`, `OutputStream`, `Reader` and `Writer` which implement only basic IO functionality
- We want to add more functionality to simple IO streams
  - Buffered stream – additional buffer functionality to the IO stream
  - Data stream – additional operations which work with Java basic types within the IO stream
  - Pushback stream – additional functionality that allows revert IO operations in the IO stream
- We do not want to modify core IO classes, instead of that we use Decorator design pattern to add a new functionality
  - Java calls them filters
  - For example: `BufferedInputStream`, `DataInputStream`, `PushbackInputStream`, etc.
    - Their constructors need object of `InputStream` which is then decorated by new functionality

Facade

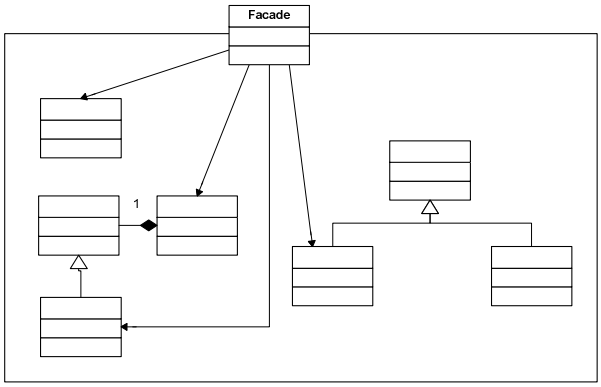
- Purpose
  - Defines a single (simple) interface for a set of interfaces from a subsystem
- Motivation
  - Structuring a system to subsystems reduces the complexity
  - Subsystems are usually groups of classes or groups of classes and other subsystems
  - Interface combining all interfaces of the subsystem can be very complex (almost unusable)

Facade

- Usage
  - When we want to present **simple interface** of a complex subsystem. This new interface will be sufficient for most clients, other (sophisticated) clients can still go deeper "behind the facade"
  - When we need to **hide the interfaces** of some subsystem against clients or other subsystems. This improves independency and portability of the subsystem

Facade

- Structure

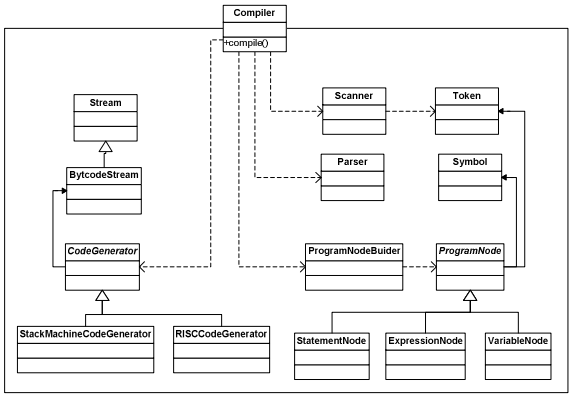


Facade

- Advantages
  - Hides implementation of the subsystem against clients, which makes it simplier to use the subsystem
  - Weakens binding among subsystems, which allows flexibile modification (or change) of subsystems without impacts to the clients
  - Reduces compilation effort in large software systems
  - Simplifies the portability of subsystems
  - Sophisticated clients can still access the whole subsystem
  - This pattern does not support any functionality it just reduces existing interface

Facade – Example

- Compiler



Proxy

- Purpose
  - Presents proxy object which manages access to (usage of) another object
- Motivation
  - There are situations when clients cannot use (refer to) an object directly
  - Proxy object can act as a broker between the client and the original object

Proxy

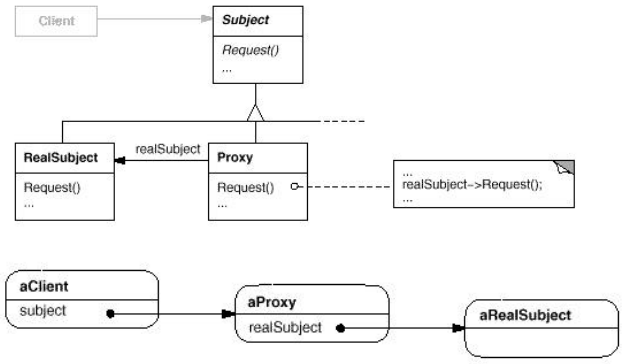
- Usage
  - Proxy object has the same interface as an original object
  - Proxy object keeps the reference (any type of referencing) to an original object and forewards the requests from the client to the original object – delegated execution
  - Proxy object is allowed to act on behalf of the client with the original object
  - Proxy object is usefull whenever there is a need for more complex connection (e.g. remote access) to the original object than a simple object reference

Proxy

- Proxy object types
  - **Remote proxy** – reference to an object in different address space or different computer
  - **Virtual proxy** – original object is created only when it is needed
  - **Copy-on-write proxy** – postpone the copy of original object until the action is performed (variation of virtual proxy)
  - **Protection (access) proxy** – provides the security levels of the clients to access the original object
  - **Cache proxy** – temporary object keeps results of time-consuming operations of original object for the clients
  - **Firewall proxy** – secures access to the object against malicious clients
  - **Synchronization proxy** – manages multiple (concurrent) access to the object
  - **Smart reference proxy** – performs additional operations when refering to original object

Proxy

- Structure



Virtual proxy – Example

- Class `ProxyImage` is used to postpone the image loading to the memory until the image is needed by the client

```
public interface Image {
  public void displayImage();
}

public class RealImage implements Image {
  private String filename;
  public RealImage(String filename) {
    this.filename = filename;
    loadImageFromDisk();
  }
  private void loadImageFromDisk() {
    System.out.println("Loading " + this.filename);
  }
  public void displayImage() {
    System.out.println("Displaying " + this.filename);
  }
}
```

Time consuming operation

Virtual proxy – Example

- Let's create proxy of the real image, class `ProxyImage`

```
public class ProxyImage implements Image {
  private String filename;
  private RealImage image;
  public ProxyImage(String filename) {
    this.filename = filename;
    this.image = null;
  }
  public void displayImage() {
    if (this.image == null)
      this.image = new RealImage(this.filename);
    this.image.displayImage();
  }
}
```


Virtual proxy – Example

- Let's test our application

```
public class ProxyExample {
  public static void main(String[] args) {
    System.out.println("Create images");
    ProxyImage image1 = new ProxyImage("Photo1"));
    ProxyImage image2 = new ProxyImage("Photo2"));
    System.out.println("Use images");
    image1.displayImage();
    image1.displayImage();
    image2.displayImage();
    image2.displayImage();
    image1.displayImage();
  }
}
```

Test application output

```
Create images
Use images
Loading Photo1
Displaying Photo1
Displaying Photo1
Loading Photo2
Displaying Photo2
Displaying Photo2
Displaying Photo1
```

Readings

- GAMMA, E. – HELEM, R. – JOHNSON, R. – VLISSIDES, J. M.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994
  - 4 Structural Patterns