Elements of Design Patterns

Shahram Rahatlou

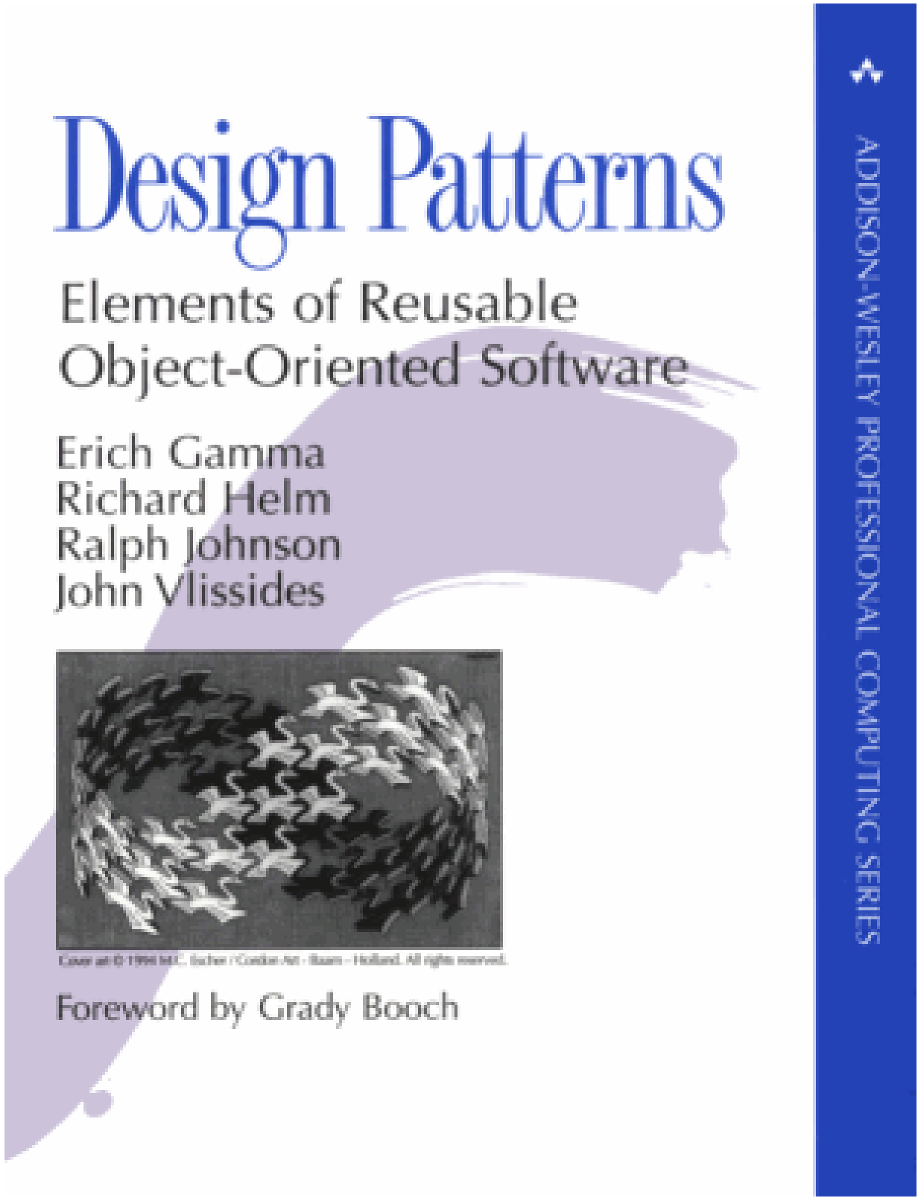
SAPIENZA
UNIVERSITÀ DI ROMA

http://www.roma1.infn.it/people/rahatlou/programmazione++/

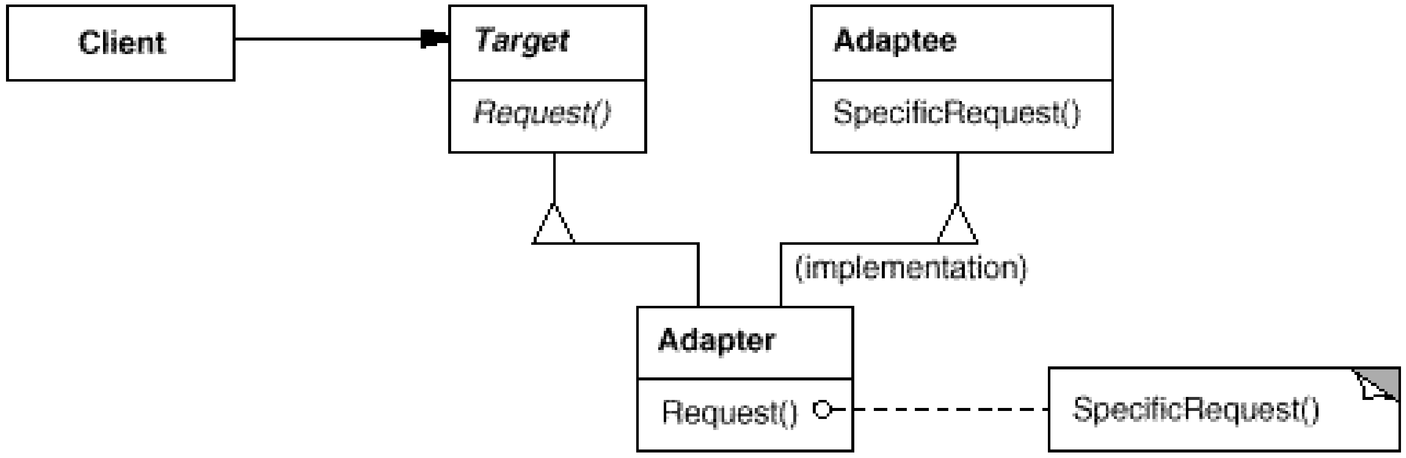
Corso di Programmazione++

Roma, 9 June 2008

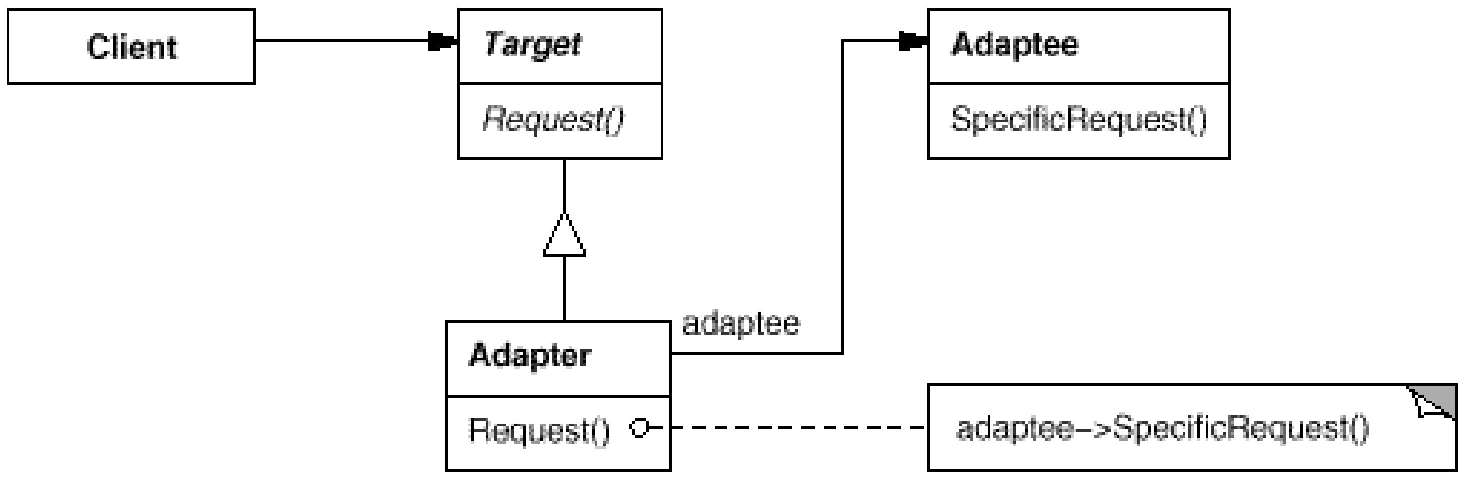
Reference Book


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

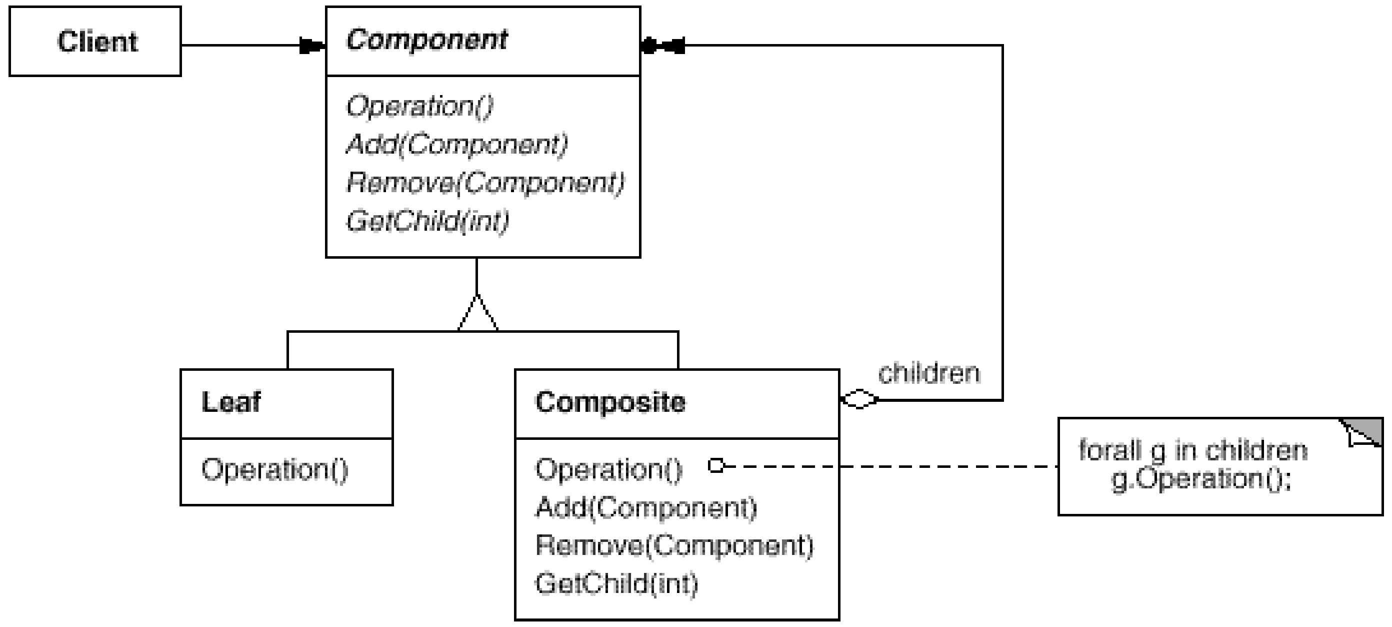
Classe Adapter

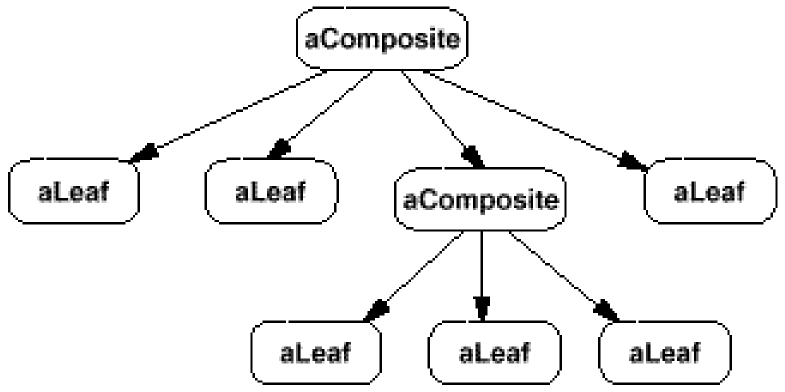


Object Adapter



Composite


Client
Component
Operation()
Add(Component)
Remove(Component)
GetChild(int)
Leaf
Operation()
Composite
Operation()
Add(Component)
Remove(Component)
GetChild(int)
children
forall g in children
g.Operation();


aComposite
aLeaf
aLeaf
aComposite
aLeaf
aLeaf
aLeaf
aLeaf

Singleton

- Pattern to provide a class with
  - only one instance at ANY time
  - Only one global access point to this instance

- Why would such a class be good for?

- Why not using global variables?
  - A part from globals being evil… ☺

Some Answers

- Examples of object with only one instance
  - Quite often in hardware control systems
  - Only one system to be used and controlled by different devices
- Advantage of only one instance
  - No race condition in use of resources by different clients
  - The instance keeps knowledge about who is using what and when
- Why not global variables?
  - Can't ensure only one instance
  - How to provide easy access?
    - What is the name of the object
    - Where was it created?
    - What is its lifetime

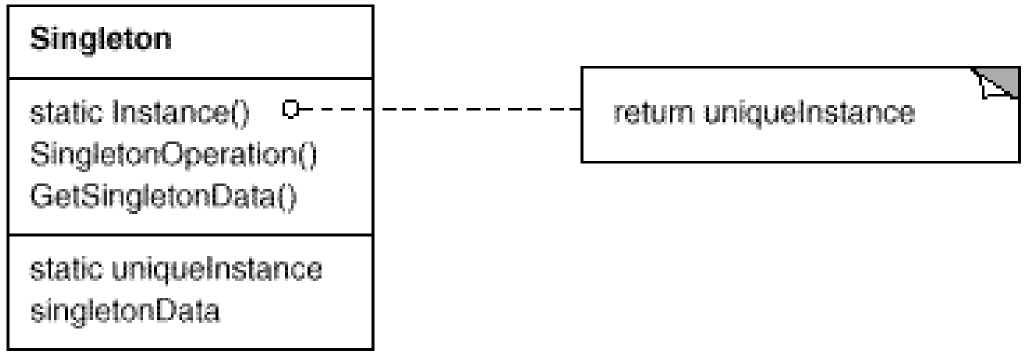
Single Instance of Objects

- How can you force and ensure only one object is created?

- Constructors can be called by anyone
  - Can you limit who can call them?
  - Can you put a condition on when an object is created?

- What if the constructor is not public?
  - Is it possible?
  - How could we create ANY instance of such class?

Structure of Singleton



Example of Singleton

```
#ifndef Singleton_h
#define Singleton_h
class Singleton {
  public:
    static Singleton* Instance();

  protected:
    Singleton();

  private:
    static Singleton* _instance;
};
#endif
```

Lazy Initialization

- The instance not created in memory until first use
- If nobody asks for instance no object created
- With global variables all objects MUST be created at the beginning

Singleton Destructor

- Major problem of Singleton is cleanup
- Destructor should be called by clients
- Possibility of conflict with other clients
- Cleanup is left to OS when program ends executaion

Possible Alternative: Monostate Pattern

- Object with only one possible state
  - Class with ONLY static data members

- Remember: Singleton is required to have a unique static instance_
  - There can be other non static data and methods

- Monostate MUST have all static data
  - All objects will always have exactly same state

Monostate Pattern

```
#ifndef Monostate_h
#define Monostate_h
class PDGTable {
  public:
    PDGTable();
    static Particle* electron();
    ~PDGTable();

  private:
    static table* _table;
};
#endif


PDGTable tab1;
Particle* e = tab1->electron();

PDGTable  tab2;
Particle* proton = tab2->proton();
```

Summary of Singleton

- Singleton Advantages
  - Base class of a singleton can be not a singleton
  - Lazy construction: create object only at first occurrence

- Singleton is bad because
  - Undefined destruction
  - Extra indirection because of access via pointer
  - Subclasses of singletons ARE not automatically singleton
    - Must implement singleton behavior explicitly

Summary of Monostate

- Advantages
  - Derivatives of monostates can be monostates
  - Monostates can have virtual methods
  - Well defined destruction policy
  - No need for access through pointer
  - Simple and clear use of new/delete and creation on stack
    - Always one and only one object is used

- Bad about Monostate
  - Can't make a class Monostate by inheriting from another Monostate
  - Monostate is always allocated: no lazy creation
  - No significant constructor
    - Can only initialize static data members
  - If clients unaware of Monostate class might be using same same object w/o knowing it!

SubClasses of Singletons

- How to handle sub classes?
  - Several implementations of same singleton interface
  - Prefer to hide sub classes from clients
  - Use only the interface instead

- Two possible approaches
  - Make instance() aware of sub classes
  - Use registry to keep track of different sub classes
    - Register by name