LOG6306 : Études empiriques sur les patrons logiciels

Foutse Khomh

The Abstract Factory DP

Context

- From Encapsulation, Inheritance, Types, Overloading, Overriding, Polymorphism, and Abstraction

Everything that `InsertionSort` offers is available to `Client`

- Take a sort algorithm, *e.g.*, an insertion sort

```java
package ca.polymtl.gigl.log6306;
import java.util.List;
public class InsertionSort<E> {
    public List<E> sort(final List<E> aList) {
        final List<E> temporaryList = null;

        // Some implementation...

        return temporaryList;
    }
}
```

```java
public class Client {
    public static void main(final String[] args) {
        final InsertionSort<String> s = new InsertionSort<String>();
        final List<String> l = null;
        s.sort(l);
    }
}
```

Context

- Problems with `InsertionSort`
  - Exposes the implementation of the sort
    - What if we want to replace this implementation by another?
  - Exposes the public interface of the class
    - What if we want to share some public method with other library class **BUT NOT** with our clients?

Context

Only public methods of `AbstractSort` are available to `Client`

- We can "abstract" *all* sort algorithms

```
package ca.polymtl.gigl.log6306;
import java.util.List;
public abstract class AbstractSort<E> {
    public abstract List<E> sort(final List<E> aList);
    public void someHelperMethod() {
        // Some implementation...
    }
    protected void someOtherHelperMethod() {
        // Some implementation...
    }
}
```

```
public class Client {
    public static void main(final String[] args) {
        final AbstractSort<String> s = new InsertionSort<String>();
        final List<String> l = null;
        s.sort(l);
    }
}
```

Context

- Advantage with the abstract class
  - Code reuse
- Problems with `AbstractSort`
  - Exposes the public interface of the class
    - What if we want to share some public methods with other library classes **BUT NOT** with our clients?
  - Exposes the "partial" type of the sort
    - What if we want to use an implementation of a sort algorithm that is **NOT** a subclass of `AbstractSort`?

Conte

Only public methods in `ISort` are available to `Client`

- We can abstract *all* sort algorithms

```java
package ca.polymtl.gigl.log6306;
import java.util.List;
public interface ISort<E> {
    public List<E> sort(final List<E> aList);
}
```

```java
public class Client {
    public static void main(final String[] args) {
        final ISort<String> s = new InsertionSort<String>();
        final List<String> l = null;
        s.sort(l);
    }
}
```

Conte

Still the Client has a dependency on the concrete implementation

- We can abstract *all* sort algorithms

```
package ca.polymtl.gigl.log6306;
import java.util.List;
public interface ISort<E> {
    public List<E> sort(final List<E> aList);
}
```

```
public class Client {
    public static void main(final String[] args) {
        final ISort<String> s = new InsertionSort<String>();
        final List<String> l = null;
        s.sort(l);
    }
}
```

Context

Problem: How to remove the dependency on the concrete implementation?

Solution: Abstract Factory design pattern

```
public class Client {
    public static void main(final String[] args) {
        final ISort<String> s = new InsertionSort<String>();
        final List<String> l = null;
        s.sort(l);
    }
}
```

Abstract Factory (1/13)

- Let us put the previous code all together
- Let us go step by step to remove the dependency on the concrete implementation

- But first, we must package the code…

Abstract Factory (2/13)

```
public interface ISort<E> {
    public List<E> sort(final List<E> aList);
}
public abstract class AbstractSort<E> {
    public abstract List<E> sort(final List<E> aList);
    public void someHelperMethod() {
        // Some implementation...
    }
    protected void someOtherHelperMethod() {
        // Some implementation...
    }
}
public class InsertionSort<E> extends AbstractSort implements ISort {
    public List<E> sort(final List<E> aList) {
        final List<E> temporaryList = null;
        // Some implementation...
        return temporaryList;
    }
}
public class Client {
    public static void main(final String[] args) {
        final ISort<String> s = new InsertionSort<String>();
        final List<String> l = ...;
        s.sort(l);
    }
}
```

Single responsibility principle

Abstract Factory (3/13)

```
public interface ISort<E> {
    public List<E> sort(final List<E> aList);
}
public abstract class AbstractSort<E> implements ISort {
    public abstract List<E> sort(final List<E> aList);
    public void someHelperMethod() {
        // Some implementation...
    }
    protected void someOtherHelperMethod() {
        // Some implementation...
    }
}
public class InsertionSort<E> extends AbstractSort implements ISort {
    public List<E> sort(final List<E> aList) {
        final List<E> temporaryList = null;
        // Some implementation...
        return temporaryList;
    }
}
public class Client {
    public static void main(final String[] args) {
        final ISort<String> s = new InsertionSort<String>();
        final List<String> l = ...;
        s.sort(l);
    }
}
```

Abstract Factory (4/13)

```
package ca.polymtl.gigl.log6306.sort;
public interface ISort<E> {
    public List<E> sort(final List<E> aList);
}
public abstract class AbstractSort<E> implements ISort {
    public abstract List<E> sort(final List<E> aList);
    public void someHelperMethod() {
        // Some implementation...
    }
    protected void someOtherHelperMethod() {
        // Some implementation...
    }
}
public class InsertionSort<E> extends AbstractSort implements ISort {
    public List<E> sort(final List<E> aList) {
        final List<E> temporaryList = null;
        // Some implementation...
        return temporaryList;
    }
}
```

```
package ca.polymtl.gigl.log6306.client;
public class Client {
    public static void main(final String[] args) {
        final ISort<String> s = new InsertionSort<String>();
        final List<String> l = ...;
        s.sort(l);
    }
}
```

Abstract Factory (5/13)

```
package ca.polymtl.gigl.log6306.sort;
public interface ISort<E> {
    public List<E> sort(final List<E> aList);
}
public abstract class AbstractSort<E> implements ISort {
    public abstract List<E> sort(final List<E> aList);
    public void someHelperMethod() {
        // Some implementation...
    }
    protected void someOther
        // Some implementati
    }
}
public class InsertionSort<E
    public List<E> sort(fina
        final List<E> tempor
        // Some implementation...
        return temporaryList;
    }
}
```

```
package ca.polymtl.gigl.log6306.client;
public class Client {
    public static void main(final String[] args) {
        final ISort<String> s = new InsertionSort<String>();
        final List<String> l = ...;
        s.sort(l);
    }
}
```

No exposure of the client to our implementation

Abstract Factory (6/13)

```
package ca.polymtl.gigl.log6306.sort;
public interface ISort<E> {
    public List<E> sort(final List<E> aList);
}
public abstract class AbstractSort<E> implements ISort {
    public abstract List<E> sort(final List<E> aList);
    public void someHelperMethod() {
        // Some implementation...
    }
    protected void someOtherHelperMethod() {
        // Some implementation...
    }
}
public class InsertionSort<E> extends AbstractSort implements ISort {
    public List<E> sort(final List<E> aList) {
        final List<E> temporaryList = null;
        // Some implementation...
        return temporaryList;
    }
}

package ca.polymtl.gigl.log6306.client;
public class Client {
    public static void main(final String[] args) {
        final ISort<String> s = new InsertionSort<String>();
        final List<String> l = ...;
        s.sort(l);
    }
}
```

Abstract Factory (7/13)

```
package ca.polymtl.gigl.log6306.sort;
public interface ISort<E> {
    public List<E> sort(final List<E> aList);
}
public abstract class AbstractSort<E> implements ISort {
    public abstract List<E> sort(final List<E> aList);
    public void someHelperMethod() {
        // Some implementation...
    }
    protected void someOtherHelperMe
        // Some implementation...
    }
}
public class InsertionSort<E> extends AbstractSort implements ISort {
    public List<E> sort(final List<E> aList) {
        final List<E> temporaryList = null;
        // Some implementation...
        return temporaryList;
    }
}

package ca.polymtl.gigl.log6306.client;
public class Client {
    public static void main(final String[] args) {
        final ISort<String> s = new InsertionSort<String>();
        final List<String> l = ...;
        s.sort(l);
    }
}
```

Not visible anymore!

Abstract Factory (8/13)

- Intent
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes

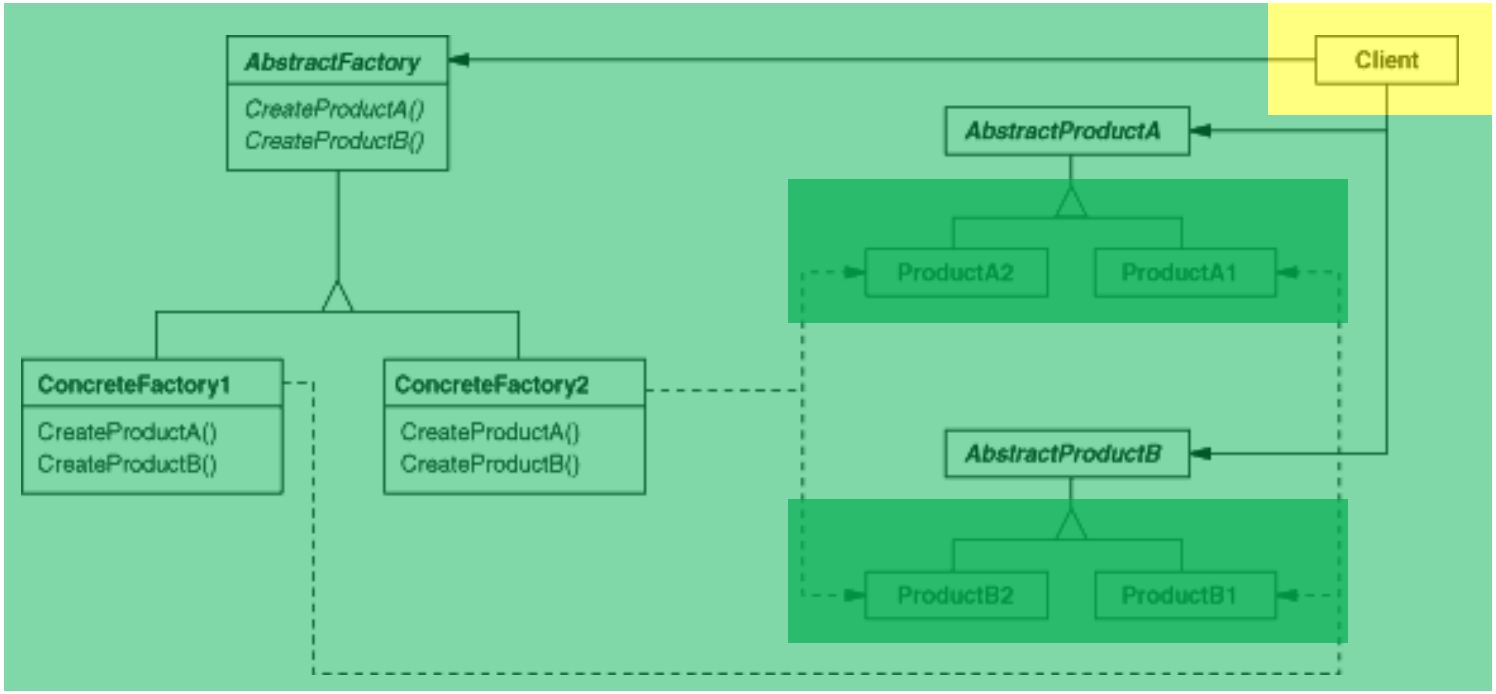
    (i.e., obtain new objects without knowing their concrete classes)

Abstract Factory (9/13)

- Applicability
  - A system should be independent of how its products are created, composed, and represented
  - A system should be configured with one of multiple families of products
  - A family of related product objects is designed to be used together and you must enforce this constraint
  - You want to provide a library of products, and you want to reveal just their interfaces, not their implementations

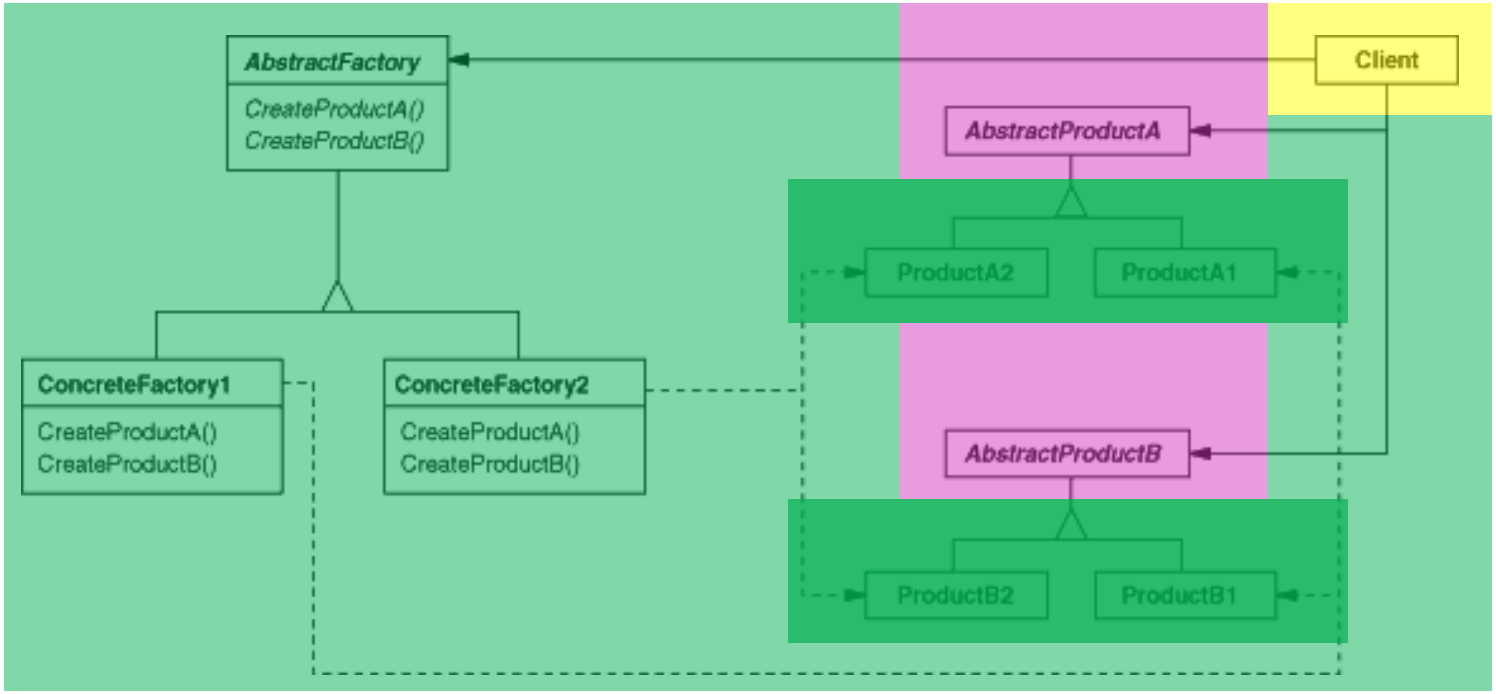
Abstract Factory (10/13)

- Design



Abstract Factory (11/13)

- Design



Abstract Factory (12/13)

```
package ca.polymtl.gigl.log6306.sort;
public interface ISort<E> {
    public List<E> sort(final List<E> aList);
}

package ca.polymtl.gigl.log6306.sort.impl;
public class Factory {
    public <E> ISort<E> getSortAlgorithm() {
        return new InsertionSort<E>();
    }
}
class InsertionSort<E> extends AbstractSort implements ISort {
    public List<E> sort(final List<E> aList) {
        final List<E> temporaryList = null;
        // Some implementation...
        return temporaryList;
    }
}

package ca.polymtl.gigl.log6306.client;
public class Client {
    public static void main(final String[] args) {
        final ISort<String> s = new Factory ().getSortAlgorithm();
        final List<String> l = ...;
        s.sort(l);
    }
}
```

Abstract Factory (13/13)

```
package ca.polymtl.gigl.log6306.sort;
public interface ISort<E> {
    public List<E> sort(final List<E> aList);
}

package ca.polymtl.gigl.log6306.sort.impl;
public class Factory {
    public <E> ISort<E> getSortAlgorithm
        return new InsertionSort<E>();
    }
}
class InsertionSort<E> extends AbstractS
    public List<E> sort(final List<E> aL
        final List<E> temporaryList = nu
        // Some implementation...
        return temporaryList;
    }
}

package ca.polymtl.gigl.log6306.client;
public class Client {
    public static void main(final String[] args) {
        final ISort<String> s = new Factory().getSortAlgorithm();
        final List<String> l = ...;
        s.sort(l);
    }
}
```

No more dependent on implementation

Choosing the Factory

- Still, the code has a dependency on the Factory to obtain the concrete implementation

Problem: How to remove the dependency on the Factory class?

Solution: Use class-loading and introspection

Choosing the Factory

- As with the Visitor design pattern
  - Bypass the compiler / virtual machine
  - Choose at runtime a concrete type

Choosing the Factory

- As with the Visitor design pattern
  - Bypass the compiler / virtual machine
  - Choose at runtime a concrete type
- Solution
  - Use introspection and class-loading
  - Use a configuration file

Conclusion

- By
  - Packaging the code
  - Hiding information

  And through
  - Using the Abstract Factory design pattern

- Decouple the client code from any particular implementation of an abstraction

LOG6306 : Études empiriques sur les patrons logiciels

Foutse Khomh

The Singleton DP

Context

- From The Abstract Factory DP

Context

```java
package ca.polymtl.gigl.log6306.sort;
public interface ISort<E> {
    public List<E> sort(final List<E> aList);
}

package ca.polymtl.gigl.log6306.sort.impl;
public class Factory {
    public <E> ISort<E> getSortAlgorithm() {
        return new InsertionSort<E>();
    }
}
class InsertionSort<E> extends AbstractSort implements ISort {
    public List<E> sort(final List<E> aList) {
        final List<E> temporaryList = null;
        // Some implementation...
        return temporaryList;
    }
}

package ca.polymtl.gigl.log6306.client;
public class Client {
    public static void main(final String[] args) {
        final ISort<String> s = new Factory ().getSortAlgorithm();
        final List<String> l = ...;
        s.sort(l);
    }
}
```

Context

```
package ca.polymtl.gigl.log6306.sort;
public interface ISort<E> {
    public List<E> sort(final List<E> aList);
}

package ca.polymtl.gigl.log6306.sort.impl;
public class Factory {
    public <E> ISort<E> getSortAlgorithm() {
        return new InsertionSort<E>()
    }
}
class InsertionSort<E> extends Abstra
    public List<E> sort(final List<E>
        final List<E> temporaryList =
        // Some implementation...
        return temporaryList;
    }
}

package ca.polymtl.gigl.log6306.client;
public class Client {
    public static void main(final String[] args) {
        final ISort<String> s = new Factory().getSortAlgorithm();
        final List<String> l = ...;
        s.sort(l);
    }
}
```

Several unnecessary instances of Factory

Context

Problem: How maintain one, and only one instance, of a class in a system?

Solution: Singleton design pattern

```
public class Client {
    public static void main(final String[] args) {
        final ISort<String> s = new Factory().getSortAlgorithm();
        final List<String> l = null;
        s.sort(l);
    }
}
```

Singleton (1/7)

- Let us modify `Factory` so that `Client`s can obtain and use the one and only one instance of this class

Singleton (2/7)

- Intent
  - Ensure a class only has one instance, and provide a global point of access to it

Singleton (3/7)

- Applicability
  - There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point
  - When the unique instance should be extensible by subclassing and clients should be able to use its subclass without modifying their code

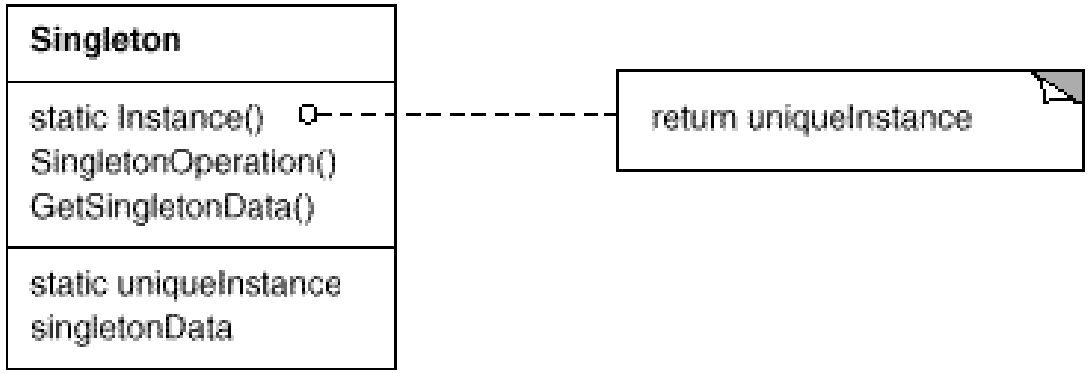
Singleton

Remember the Fragile Base Class Problem!

- Applicability
  - There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point
  - When the unique instance should be extensible by subclassing and clients should be able to use its subclass without modifying their code

Singleton (5/7)

- Design



Singleton (6/7)

- Implementation

```
package ca.polymtl.gigl.log6306.sort.impl;
import ca.polymtl.gigl.log6306.ISort;
public class Factory {
    private static Factory TheUniqueFactory;

    public static Factory getInstance() {
        if (Factory.TheUniqueFactory == null) {
            Factory.TheUniqueFactory = new Factory();
        }
        return Factory.TheUniqueFactory;
    }

    private Factory() {
        // Some implementation if needed...
        // This constructor could take in parameters
        // then getInstance() should have parameters too.
    }

    public <E> ISort<E> getSortAlgorithm() {
        return new InsertionSort<E>();
    }
}
```

Singleton

Information hiding:
- Packaging
- Accessing

- Implementation

```
package ca.polymtl.gigl.log6306.sort.impl;
import ca.polymtl.gigl.log6306.ISort;
public class Factory {
    private static Factory TheUniqueFactory;

    public static Factory getInstance() {
        if (Factory.TheUniqueFactory == null) {
            Factory.TheUniqueFactory = new Factory();
        }
        return Factory.TheUniqueFactory;
    }

    private Factory() {
        // Some implementation if needed...
        // This constructor could take in parameters
        // then getInstance() should have parameters too.
    }

    public <E> ISort<E> getSortAlgorithm() {
        return new InsertionSort<E>();
    }
}
```

Singleton

Broken in multi-threaded programs!

- Implementation

```
package ca.polymtl.gigl.log6306.sort.impl;
import ca.polymtl.gigl.log6306.ISort;
public class Factory {
    private static Factory TheUniqueFactory;

    public static Factory getInstance() {
        if (Factory.TheUniqueFactory == null) {
            Factory.TheUniqueFactory = new Factory();
        }
        return Factory.TheUniqueFactory;
    }

    private Factory() {
        // Some implementation if needed...
        // This constructor could take in parameters
        // then getInstance() should have parameters too.
    }

    public <E> ISort<E> getSortAlgorithm() {
        return new InsertionSort<E>();
    }
}
```

Singleton (1/7)

(Scoped Locking)

- Context
  - A multi-threaded program in which a unique `Factory` must be shared among multiple threads of execution

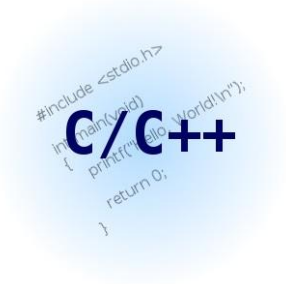
See also http://www.cs.wustl.edu/~schmidt/PDF/ScopedLocking.pdf

Singleton (2/7)

(Scoped Locking)

- Problems
  - Some sort of token must be locked and released correctly before and after instantiating the unique instance of the `Factory`
  - It is difficult to identify all exits in the control flow because of possible `return`, `break`, and `continue` statements and uncaught exceptions
- Different programming languages allow for different solutions

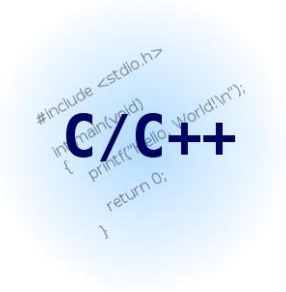
Singleton
(Scoped Locking)



- Solution for C++
  - Use a guard class
    - Its constructor locks a token
    - Its destructor releases a token
  - Implementation

```
Thread_Mutex lock_;
…
Thread_Mutex_Guard guard (lock_);
```

Singleton
(Scoped Locking)



- Solution for C++
  - Advantages
    - Will always release the token
    - Can be used in many contexts
    - Could be subclassed
  - Limitations
    - Possible dead-locks in case of recursive calls
    - (Compiler complains about unused `_guard` variable)
  - Know uses
    - `Threads.h++`

Singleton
(Scoped Locking)

- Solution for Java
  - Use the keyword `synchronised`

```
package ca.polymtl.gigl.log6306.sort.impl;
import ca.polymtl.gigl.log6306.ISort;
public class Factory {
    private static Factory TheUniqueFactory;

    public static Factory getInstance() {
        if (Factory.TheUniqueFactory == null) {
            Factory.TheUniqueFactory = new Factory();
        }
        return Factory.TheUniqueFactory;
    }

    private Factory() {
        // Some implementation if needed...
    }

    public <E> ISort<E> getSortAlgorithm() {
        return new InsertionSort<E>();
    }
}
```

Singleton
(Scoped Locking)

Implementation of C++ scoped locking

- Solution for Java
  - Use the keyword `synchronised`

```
package ca.polymtl.gigl.log6306.sort.impl;
import ca.polymtl.gigl.log6306.ISort;
public class Factory {
    private static Factory TheUniqueFactory;

    public static synchronized Factory getInstance() {
        if (Factory.TheUniqueFactory == null) {
            Factory.TheUniqueFactory = new Factory();
        }
        return Factory.TheUniqueFactory;
    }

    private Factory() {
        // Some implementation if needed...
    }

    public <E> ISort<E> getSortAlgorithm() {
        return new InsertionSort<E>();
    }
}
```

Singleton
(Scoped Locking)

- Solution for Java
  - Advantages
    - Very simple to use
  - Limitations
    - Performance costs 100×
  - Know uses
    - `java.util.Hashtable<K,V>`

Singleton (1/7)

(Double-check Locking Optimisation)

- Context
  - A multi-threaded program in which a unique `Factory` must be shared among multiple threads of execution
  - A multi-threaded program in which the “synchronised” method is used heavily

See also http://en.wikipedia.org/wiki/Double-checked_locking

Singleton

(Double-check Locking Optimisation)

- First solution for Java
  - Combination of the keywords `synchronised` and `volatile`
    - Make some blocks of code `synchronized` to prevent threads to access the same block at the “same” time
    - Make some instances variables `volatile` to prevent threads to cache their values and make their reading/writing atomic

Singleton

(Double-check Locking Optimisation)

- First solution for Java
  - Use `synchronised` and `volatile`

```
package ca.polymtl.gigl.log6306.sort.impl;
import ca.polymtl.gigl.log6306.ISort;
public class Factory {
    private static volatile Factory TheUniqueFactory;

    public static Factory getInstance() {
        if (Factory.TheUniqueFactory == null) {
            synchronized (Factory.class) {
                if (Factory.TheUniqueFactory == null) {
                    Factory.TheUniqueFactory = new Factory();
                }
            }
        }
        return Factory.TheUniqueFactory;
    }
    // ...
}
```

Singleton

(Double-check Locking Optimisation)

Cost only the first time

- First solution for Java
  - Use `synchronised` and `volatile`

```
package ca.polymtl.gigl.log6306.sort.impl;
import ca.polymtl.gigl.log6306.ISort;
public class Factory {
    private static volatile Factory TheUniqueFactory;

    public static Factory getInstance() {
        if (Factory.TheUniqueFactory == null) {
            synchronized (Factory.class) {
                if (Factory.TheUniqueFactory == null) {
                    Factory.TheUniqueFactory = new Factory();
                }
            }
        }
        return Factory.TheUniqueFactory;
    }
    // ...
}
```

Singleton

(Double-check Locking Optimisation)

- First solution for Java
  - Advantages
    - Simple to use
    - Cost only the first time
  - Know uses
    - The ADAPTIVE Communication Environment (ACE)

Singleton

(Double-check Locking Optimisation)

- Second solution for Java
  - Without `synchronised` and `volatile`

```
package ca.polymtl.gigl.log6306.sort.impl;
import ca.polymtl.gigl.log6306.ISort;
public class Factory {
    private static class FactoryUniqueInstanceHolder {
        private static final Factory THE_UNIQUE_FACTORY = new Factory();
    }

    public static Factory getInstance() {
        return FactoryUniqueInstanceHolder.THE_UNIQUE_FACTORY;
    }
    // ...
}
```

Singleton

(Double-check Locking Opti

Uses guarantees made by JVM class loader

- Second solution for Java
  - Without `synchronised` and `volatile`

```
package ca.polymtl.gigl.log6306.sort.impl;
import ca.polymtl.gigl.log6306.ISort;
public class Factory {
    private static class FactoryUniqueInstanceHolder {
        private static Factory THE_UNIQUE_FACTORY = new Factory();
    }

    public static Factory getInstance() {
        return FactoryUniqueInstanceHolder.THE_UNIQUE_FACTORY;
    }
    // ...
}
```

Singleton

- No matter the choice of the implementation of the Singleton, `Factory`, and sort algorithm; the client remains the same

```
public class Client {
    public static void main(final String[] args) {
        final ISort<String> s = Factory.getInstance().getSortAlgorithm();
        final List<String> l = null;
        s.sort(l);
    }
}
```

Conclusion

- By
  - Packaging the code
  - Hiding information

  And through
  - Using the Abstract Factory design pattern
  - Using the Singleton design pattern

- Ensure that all clients use the same and only instance of a factory class

Conclusion

- When instantiating classes whose instances must be shared across threads, beware of race conditions
- Use appropriate (possibly language-dependent) constructs and idioms to prevent race conditions

LOG6306 : Études empiriques sur les patrons logiciels

Foutse Khomh

The Composite DP

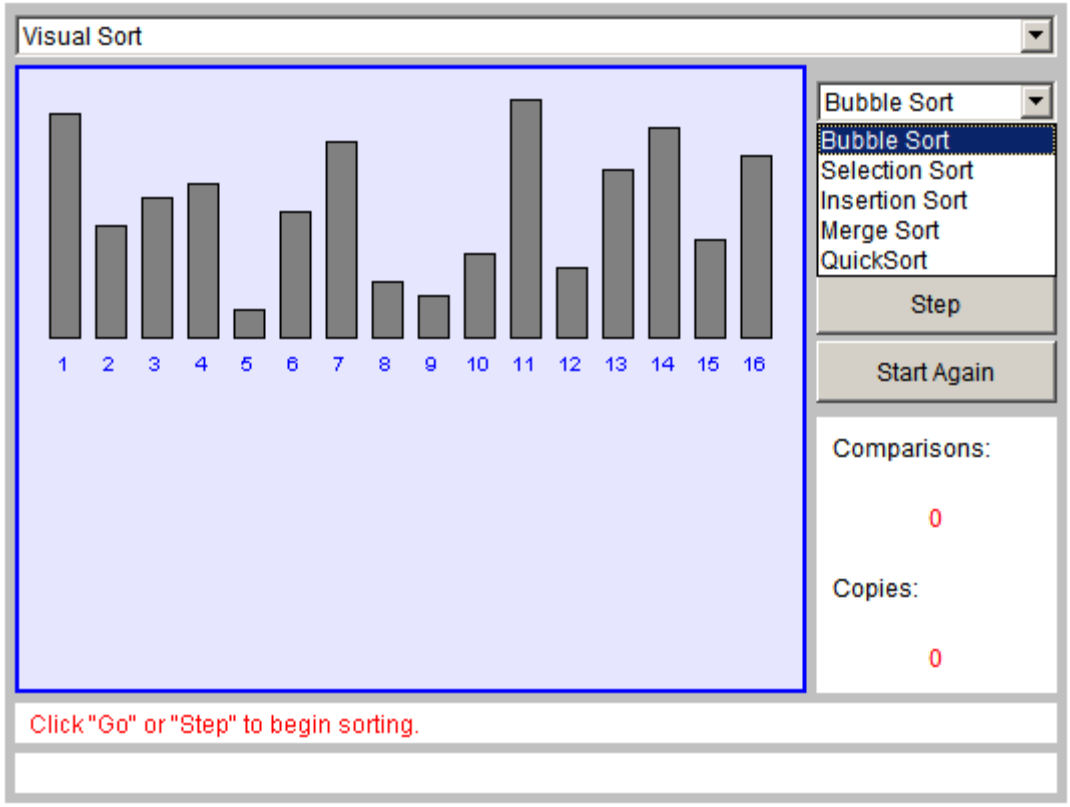
Context

- From The Singleton DP

Context

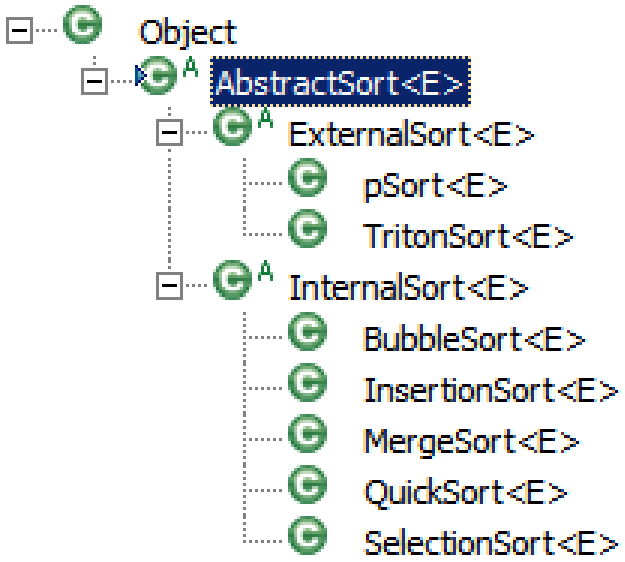
```
package ca.polymtl.gigl.log6306.sort;
public interface ISort<E> {
    List<E> sort(final List<E> aList);
}
package ca.polymtl.gigl.log6306.sort.impl;
public class Factory {
    private static class FactoryUniqueInstanceHolder {
        private static final Factory THE_UNIQUE_FACTORY = new Factory();
    }
    public static Factory getInstance() {
        return FactoryUniqueInstanceHolder.THE_UNIQUE_FACTORY;
    }
    private Factory() {
        // Some implementation if needed...
    }
    public <E> ISort<E> getSortAlgorithm() {
        return new InsertionSort<E>();
    }
}
package ca.polymtl.gigl.log6306;
public class Client {
    public static void main(final String[] args) {
        final List<String> l = ...;
        final ISort<String> s = Factory.getInstance().getSortAlgorithm();
        s.sort(l);
    }
}
```

Context


Visual Sort
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Bubble Sort
Bubble Sort
Selection Sort
Insertion Sort
Merge Sort
QuickSort
Step
Start Again
Comparisons:
0
Copies:
0
Click "Go" or "Step" to begin sorting.

Context

- Types
  - Internal
  - External
- Other types
  - Complexity in time
  - Complexity in space
- …



Context

- Many sort algorithms, two categories
- Client wants to call all the sorts, by categories **or** one chosen sort

Problem: How to let client see similarly one sort algorithm or a set of algorithms?

Solution: Composite design pattern

Composite (1/7)

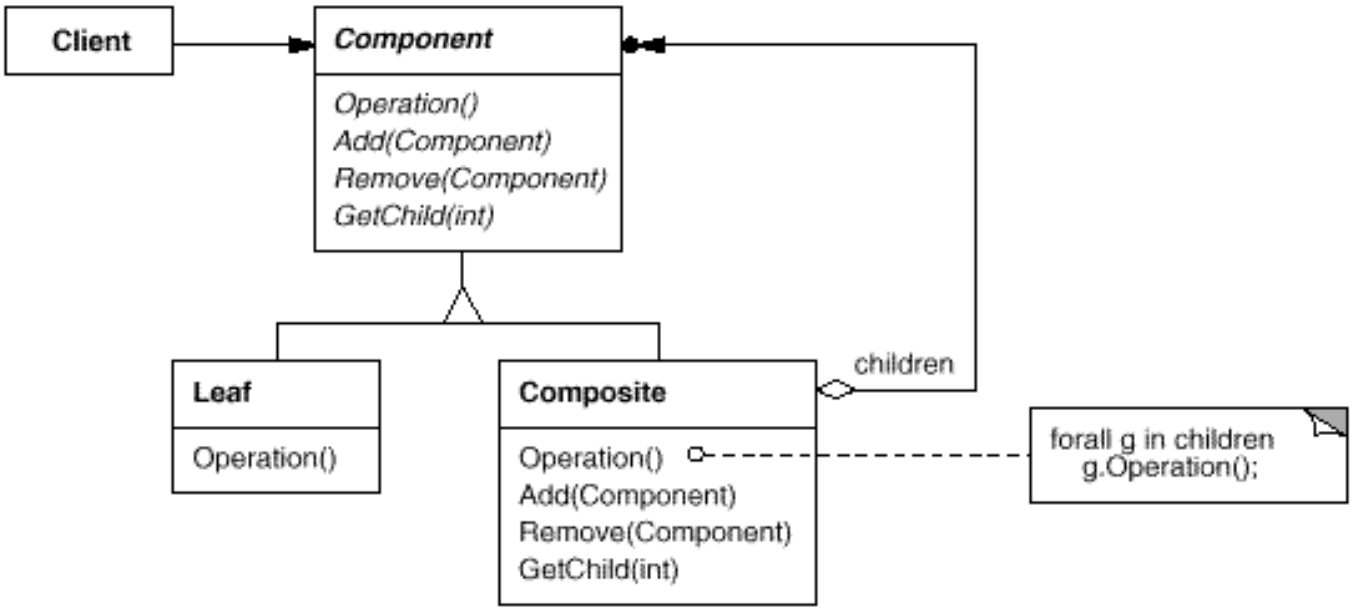
- Intent
  - Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly

Composite (2/7)

- Applicability
  - You want to represent part-whole hierarchies of objects
  - You want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly

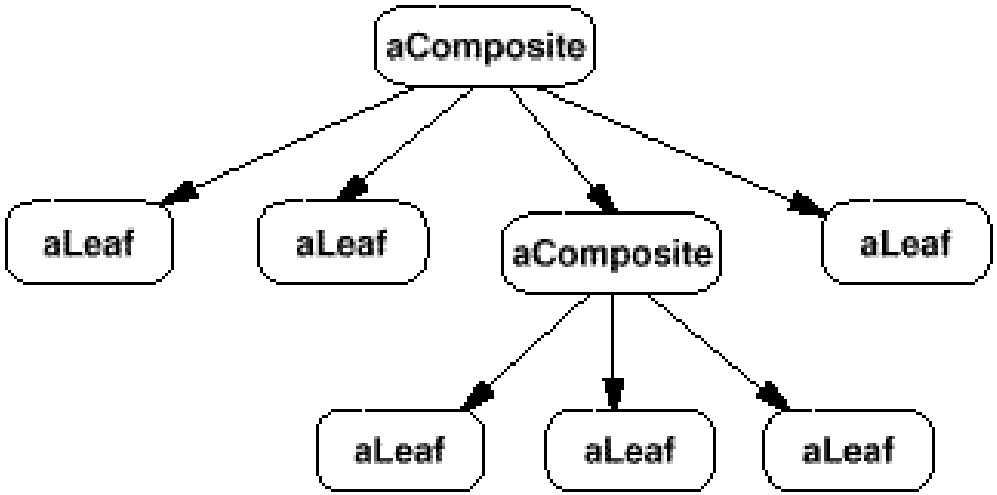
Composite (3/7)

- Design



Composite (4/7)

- Design



Composite

A “sort” that contains other sort algorithms

- Implementation

```
package ca.polymtl.gigl.log6306.sort;

public interface ITypeOfSort<E extends Comparable<E>> extends ISort<E> {
    List<ISort<E>> getSortAlgorithms();
    String getTypeName();
    List<E> sort(final List<E> aList);
}
```

Composite (6/7)

- Implementation

```
package ca.polymtl.gigl.log6306.sort.impl;
class TypeOfSort<E extends Comparable<E>> extends AbstractSort<E> implements ITypeOfSort<E> {
    private final List<ISort<E>> listOfSortAlgorithms;
    private final String typeName;

    public TypeOfSort(final String aTypeName) {
        this.listOfSortAlgorithms = new ArrayList<ISort<E>>();
        this.typeName = aTypeName;
    }
    public void addSortAlgorithm(final ISort<E> aSortAlgorithm) {
        this.listOfSortAlgorithms.add(aSortAlgorithm);
    }
    public String getTypeName() {
        return this.typeName;
    }

    // ... Continued on the next slide ...

}
```

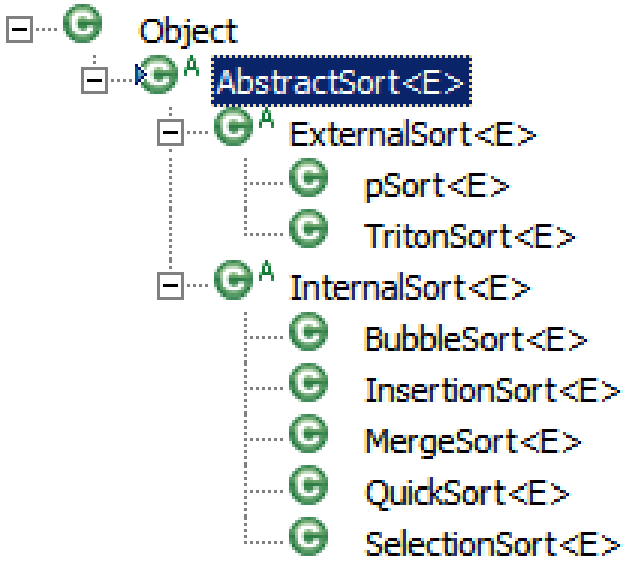
Composite (7/7)

- Implementation

```
package ca.polymtl.gigl.log6306.sort.impl;
class TypeOfSort<E extends Comparable<E>> extends AbstractSort<E> implements ITypeOfSort<E> {

    // ... Continued from the previous slide ...

    public List<E> sort(final List<E> aList) {
        // Call each sort algorithm of this type one after the other...
        final Iterator<ISort<E>> iterator = this.listOfSortAlgorithms.iterator();
        List<E> sortedList = null;
        while (iterator.hasNext()) {
            final ISort<E> sortAlgorithm = (ISort<E>) iterator.next();
            sortedList = sortAlgorithm.sort(aList);
        }
        return sortedList;
    }
    public List<ISort<E>> getSortAlgorithms() {
        return this.listOfSortAlgorithms;
    }
}
```

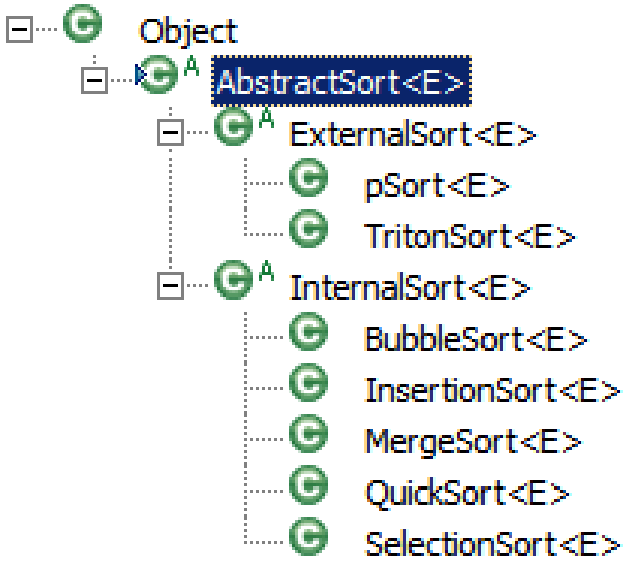
Usage

- We must build the composite objects
- Clients use these composite objects



Usage

- **We** must build the composite objects
- **Clients** use these composite objects



Usage

No difference in use

- Clients use these composite objects

```
package ca.polymtl.gigl.log6306;

public class Client {
    public static void main(final String[] args) {
        final List<String> l = Arrays.asList(new String[] { "Venus", "Earth", "Mars" });

        final ISort<String> s = Factory.getInstance().getBubbleSortAlgorithm();
        System.out.println(s.sort(l));

        final ISort<String> t = Factory.getInstance().getInternalSortAlgorithms();
        System.out.println(t.sort(l));
    }
}
```

Usage

We are responsible for creating the composites

- We must build the composite objects

```
package ca.polymtl.gigl.log6306.sort.impl;
public class Factory {
    // ... As on slide 3 ...

    public <E extends Comparable<E>> ISort<E> getInternalSortAlgorithms() {
        final TypeOfSort<E> internalSorts = new TypeOfSort<E>("Internal Sorts");

        final ISort<E> bubbleSortAlgorithm = this.getBubbleSortAlgorithm();
        internalSorts.addSortAlgorithm(bubbleSortAlgorithm);
        final ISort<E> insertionSortAlgorithm = this.getInsertionSortAlgorithm();
        internalSorts.addSortAlgorithm(insertionSortAlgorithm);
        final ISort<E> mergeSortAlgorithm = this.getMergeSortAlgorithm();
        internalSorts.addSortAlgorithm(mergeSortAlgorithm);
        final ISort<E> quickSortAlgorithm = this.getQuickSortAlgorithm();
        internalSorts.addSortAlgorithm(quickSortAlgorithm);
        final ISort<E> selectionSortAlgorithm = this.getSelectionSortAlgorithm();
        internalSorts.addSortAlgorithm(selectionSortAlgorithm);

        return internalSorts;
    }
}
```

Conclusion

- Using the composite design pattern let clients treat objects and composition of objects uniformly
  - The client does not need to know

```
package ca.polymtl.gigl.log6306;

public class Client {
    public static void main(final String[] args) {
        final List<String> l = Arrays.asList(new String[] { "Venus", "Earth", "Mars" });

        final ISort<String> s = Factory.getInstance().getBubbleSortAlgorithm();
        System.out.println(s.sort(l));

        final ISort<String> t = Factory.getInstance().getInternalSortAlgorithms();
        System.out.println(t.sort(l));
    }
}
```

Conclusion

- The Composite design pattern is applied to any “recursive ”data structure
  - Graphical objects
    - AWT
    - Swing
    - ..
  - Tree structures
    - AST
    - …

Conclusion

- The common interface between objects and composite objects is defined by the interface playing the “Component” role

```
package ca.polymtl.gigl.log6306.sort;

import java.util.List;

public interface ISort<E extends Comparable<E>> {
    List<E> sort(final List<E> aList);
}
```

Beware of not “bloating” this interface!

LOG6306 : Études empiriques sur les patrons logiciels

Foutse Khomh

The Observer DP

ÉCOLE POLYTECHNIQUE MONTRÉAL

연세대학교 YONSEI UNIVERSITY

Context

- From The Composite DP

Context

What's the good of this Composite?

```
package ca.polymtl.gigl.log6306;

public class Client {
    public static void main(final String[] args) {
        final List<String> l =
            Arrays.asList(new String[] { "Venus", "Earth", "Mars" });

        final ISort<String> s = Factory.getInstance().getBubbleSortAlgorithm();
        System.out.println(s.sort(l));

        final ISort<String> t = Factory.getInstance().getInternalSortAlgorithms();
        final ITypeOfSort<String> c = (ITypeOfSort<String>) t;

        // Use one specific sort algorithm...
        final ISortIterator<String> i = c.getSortAlgorithms();
        System.out.println(i.getNext().sort(l));

        // Use all sort algorithms...
        System.out.println(t.sort(l));
    }
}
```

Context

- Having a Composite object is interesting but we do not know what is happening inside it, and inside each of the sort algorithm

Problem: How to notify clients of events occurring in an object?

Solution: Observer design pattern

Observer (1/11)

“A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects. You don't want to achieve consistency by making the classes tightly coupled, because that reduces their reusability.”

[Gamma et al.]

Observer (2/11)

- Name: Observer

- Intent: “Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.”

Observer (3/11)

- Motivation: “Many graphical user interface toolkits separate the presentational aspects of the user interface from the underlying application data. Classes defining application data and presentations can be reused independently. They can work together, too.”

Observer (4/11)

- Motivation (cont'd): "[A] spreadsheet and [a] bar chart don't know about each other, thereby letting you reuse only the one you need. When the user changes the information in the spreadsheet, the bar chart **reflects the changes immediately**, and vice versa."

Observer (5/11)

- Motivation (cont'd): "This behavior implies that the spreadsheet and bar chart are dependent on the data object and therefore should be notified of any change in its state. […] The Observer pattern describes how to establish these relationships. The key objects in this pattern are **subject** and **observers**."

Observer (6/11)

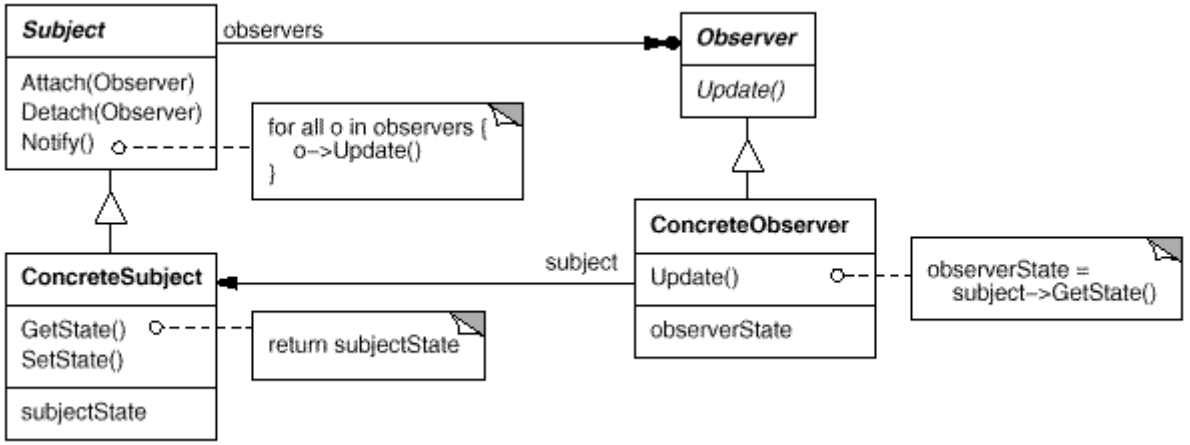
- Motivation (cont'd): "This kind of interaction is also known as **publish–subscribe**. The subject is the publisher of notifications. […] Any number of observers can subscribe to receive notifications."

Observer (7/11)

- Applicability
  - When an abstraction has two aspects, one dependent on the other
  - When a change to one object requires changing others, and you do not know how many objects must change
  - When an object should notify other objects without making assumptions about who these objects are

Observer (8/11)

- Structure

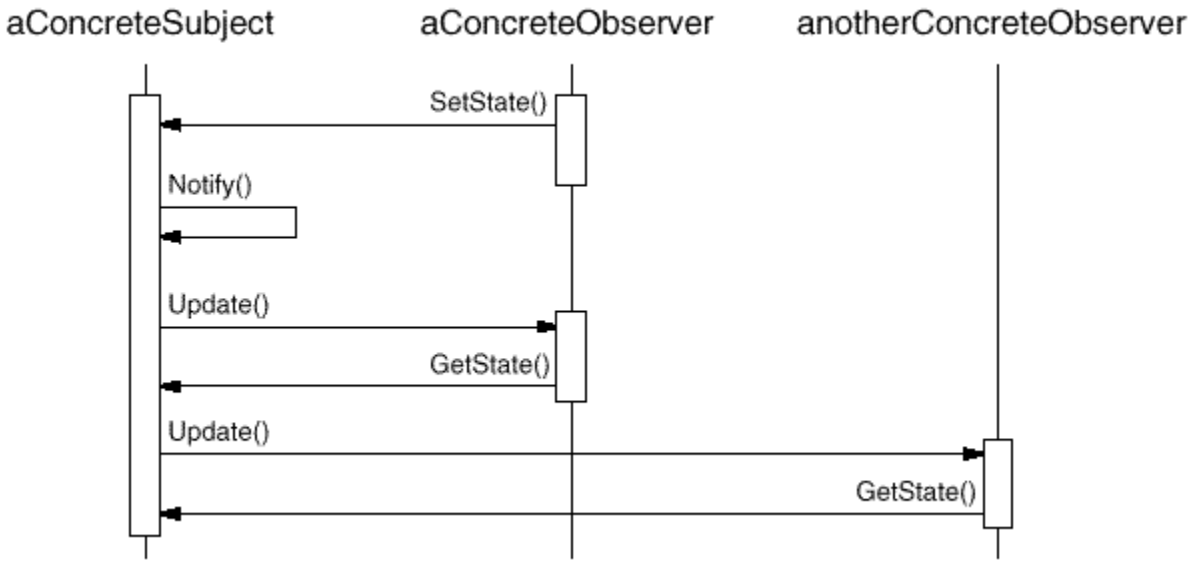


Observer (9/11)

- Participants
  - Subject
    - Knows its observers. Any number of Observer objects may observe a subject
    - Provides an interface for attaching and detaching Observer objects
  - Observer
    - Defines an updating interface for objects that should be notified of changes in a subject
  - ConcreteSubject
    - Stores state of interest to ConcreteObserver objects
    - Sends a notification to its observers when its state changes
  - ConcreteObserver
    - Maintains a reference to a ConcreteSubject object
    - Stores state that should stay consistent with the subject's
    - Implements the Observer updating interface to keep its state consistent with the subject's

Observer (10/11)

- Collaborations



Observer (11/11)

- Consequences
  - Abstract coupling between Subject and Observer
  - Support for broadcast communication
  - Unexpected updates

Implementation

- Finally, the Bubble sort algorithm!

```
package ca.polymtl.gigl.log6306.sort.impl;

class BubbleSort<E extends Comparable<E>> extends InternalSort<E> implements ISort<E> {
    public List<E> sort(final List<E> aList) {
        final int listSize = aList.size();
        final E[] array = this.convertListToArray(aList);

        E temp;
        for (int i = 0; i < listSize - 1; i++) {
            for (int j = 0; j < listSize - 1; j++) {
                if (array[j].compareTo(array[j + 1]) > 0) {
                    temp = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = temp;
                }
            }
        }
        return Arrays.asList(array);
    }
    public String getName() {
        return "BubbleSort";
    }
}
```

Implementation

- Two interesting operations
  - Comparison
  - Swap

Implementation

- Bubble sort algorithm

```
package ca.polymtl.gigl.log6306.sort.impl;

class BubbleSort<E extends Comparable<E>> extends InternalSort<E> implements ISort<E> {
    public List<E> sort(final List<E> aList) {
        final int listSize = aList.size();
        final E[] array = this.convertListToArray(aList);

        E temp;
        for (int i = 0; i < listSize - 1; i++) {
            for (int j = 0; j < listSize - 1; j++) {
                if (array[j].compareTo(array[j + 1]) > 0) {
                    temp = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = temp;
                }
            }
        }
        return Arrays.asList(array);
    }
    public String getName() {
        return "BubbleSort";
    }
}
```

Implementation

- Bubble sort algorithm

```
package ca.polymtl.gigl.log6306.sort.impl;

class BubbleSort<E extends Comparable<E>> extends InternalSort<E> implements ISort<E> {
    public List<E> sort(final List<E> aList) {
        final int listSize = aList.size();
        final E[] array = this.convertListToArray(aList);

        E temp;
        for (int i = 0; i < listSize - 1; i++) {
            for (int j = 0; j < listSize - 1; j++) {
                if (array[j].compareTo(array[j + 1]) > 0) {
                    temp = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = temp;
                }
            }
        }
        return Arrays.asList(array);
    }

    // ...
```

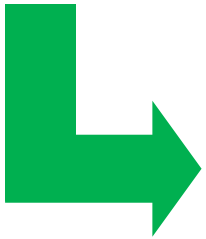
Implementation

- Two interesting operations
  - Comparison
  - Swap

Must refactor the code

Implementation

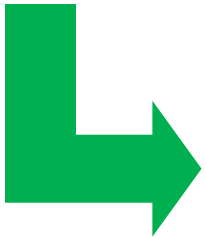
```
public List<E> sort(final List<E> aList) {
    final int listSize = aList.size();
    final E[] array = this.convertListToArray(aList);

    E temp;
    for (int i = 0; i < listSize - 1; i++) {
        for (int j = 0; j < listSize - 1; j++) {
            if (array[j].compareTo(array[j + 1]) > 0) {
                temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;
            }
        }
    }
    return Arrays.asList(array);
}
```



```
public List<E> sort(final List<E> aList) {
    final int listSize = aList.size();
    final E[] array = this.convertListToArray(aList);

    for (int i = 0; i < listSize - 1; i++) {
        for (int j = 0; j < listSize - 1; j++) {
            if (this.compareValues(array, j, j + 1) > 0) {
                this.swapValues(array, j, j + 1);
            }
        }
    }
    return Arrays.asList(array);
}
```

Implementation

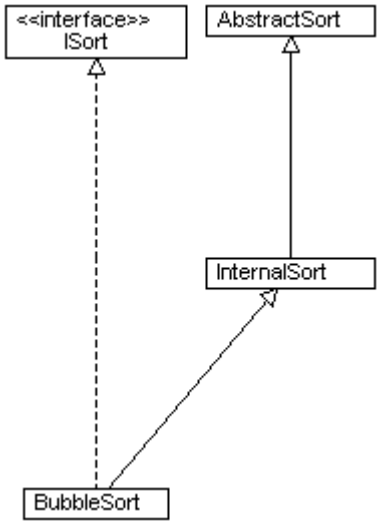
```
public List<E> sort(final List<E> aList) {
    final int listSize = aList.size();
    final E[] array = this.convertListToArray(aList);

    E temp;
    for (int i = 0; i < listSize - 1; i++) {
        for (int j = 0; j < listSize - 1; j++) {
            if (array[j].compareTo(array[j + 1]) > 0) {
                temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;
            }
        }
    }
    return Arrays.asList(array);
}
```



```
public List<E> sort(final List<E> aList) {
    final int listSize = aList.size();
    final E[] array = this.convertListToArray(aList);

    for (int i = 0; i < listSize - 1; i++) {
        for (int j = 0; j < listSize - 1; j++) {
            if (this.compareValues(array, j, j + 1) > 0) {
                this.swapValues(array, j, j + 1);
            }
        }
    }
    return Arrays.asList(array);
}
```

Implementation

```
package ca.polymtl.gigl.log6306.sort.impl;
abstract class AbstractSort<E extends Comparable<E>> {

    // ...

    protected int compareValues(final E[] values, final int position1, int position2) {
        return values[position1].compareTo(values[position2]);
    }
    protected void swapValues(final E[] array, final int i, final int j) {
        final E temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }

    // ...
```



Implementation

```
package ca.polymtl.gigl.log6306.sort.impl;
abstract class AbstractSort<E extends Comparable<E>> {
    private final List<ISortObserver<E>> listOfObservers;
    public AbstractSort() {
        this.listOfObservers = new ArrayList<ISortObserver<E>>();
    }
    protected int compareValues(final E[] values, final int position1, int position2) {
        this.notifyObserversOfAComparison(values,
            position1, values[position1],
            position2, values[position2]);
        return values[position1].compareTo(values[position2]);
    }
    private void notifyObserversOfAComparison(
        final E[] values, final int position1, final E value1,
        final int position2, final E value2) {
        final ComparisonEvent<E> event =
            new ComparisonEvent<E>(values, position1, value1, position2, value2);

        final Iterator<ISortObserver<E>> iterator = this.listOfObservers.iterator();
        while (iterator.hasNext()) {
            final ISortObserver<E> sortObserver = (ISortObserver<E>) iterator.next();
            sortObserver.valuesCompared(event);
        }
    }

    // ...
```

Implementation

```
package ca.polymtl.gigl.log6306.sort.impl;
abstract class AbstractSort<E extends Comparable<E>> {
    private final List<ISortObserver<E>> listOfObservers;
    public AbstractSort() {
        this.listOfObservers = new ArrayList<ISortObserver<E>>();
    }
    protected int compareValues(final E[] values, final int position1, int position2) {
          this.notifyObserversOfAComparison(values,
              position1, values[position1],
              position2, values[position2]);
        return values[position1].compareTo(values[position2]);
    }
    private void notifyObserversOfAComparison(
        final E[] values, final int position1, final E value1,
        final int position2, final E value2) {
        final ComparisonEvent<E> event =
            new ComparisonEvent<E>(values, position1, value1, position2, value2);

        final Iterator<ISortObserver<E>> iterator = this.listOfObservers.iterator();
        while (iterator.hasNext()) {
            final ISortObserver<E> sortObserver = (ISortObserver<E>) iterator.next();
            sortObserver.valuesCompared(event);
        }
    }

    // ...
```

Implementation

- We must define the observers
- We must allow the observers to attach to the sort algorithms

```
package ca.polymtl.gigl.log6306.sort.observer;

public interface ISortObserver<E extends Comparable<E>> {
    void valuesCompared(final ComparisonEvent<E> comparisonEvent);
    void valuesSwapped(final SwapEvent<E> swapEvent);
}
```

```
package ca.polymtl.gigl.log6306.sort;

public interface ISort<E extends Comparable<E>> {
    List<E> sort(final List<E> aList);
    void addObserver(final ISortObserver<E> anObserver);
}
```

Implementation

- We must define the observers
- We must allow the observers to attach to the sort algorithms

```
package ca.polymtl.gigl.log6306.sort.observer;

public interface ISortObserver<E extends Comparable<E>> {
    void valuesCompared(final ComparisonEvent<E> comparisonEvent);
    void valuesSwapped(final SwapEvent<E> swapEvent);
}
```

```
package ca.polymtl.gigl.log6306.sort;

public interface ISort<E extends Comparable<E>> {
    List<E> sort(final List<E> aList);
    void addObserver(final ISortObserver<E> anObserver);
}
```

Implementation

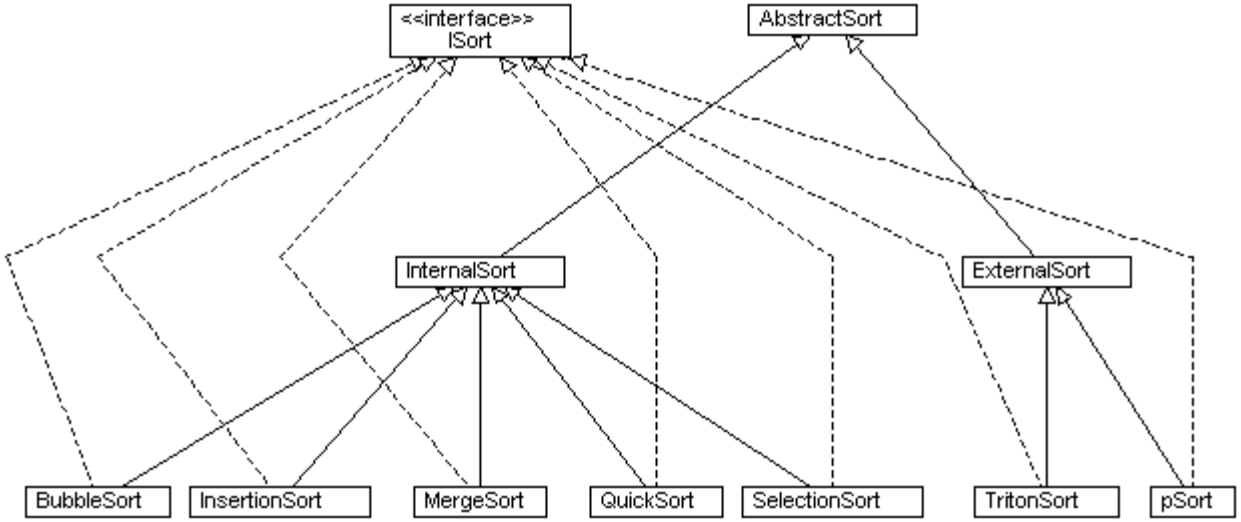
```
package ca.polymtl.gigl.log6306.sort.impl;
abstract class AbstractSort<E extends Comparable<E>> {
    private final List<ISortObserver<E>> listOfObservers;
    public AbstractSort() {
        this.listOfObservers = new ArrayList<ISortObserver<E>>();
    }
    public void addObserver(final ISortObserver<E> anObserver) {
        this.listOfObservers.add(anObserver);
    }
    protected int compareValues(final E[] values, final int position1, int position2) {
        this.notifyObserversOfAComparison(values,
            position1, values[position1],
            position2, values[position2]);
        return values[position1].compareTo(values[position2]);
    }
    private void notifyObserversOfAComparison(
        final E[] values, final int position1, final E value1,
        final int position2, final E value2) {
        final ComparisonEvent<E> event =
            new ComparisonEvent<E>(values, position1, value1, position2, value2);

        final Iterator<ISortObserver<E>> iterator = this.listOfObservers.iterator();
        while (iterator.hasNext()) {
            final ISortObserver<E> sortObserver = (ISortObserver<E>) iterator.next();
            sortObserver.valuesCompared(event);
        }
    }

    // ...
```

Implementatio

All sort algorithms benefit from this method

```
package ca.polymtl.gigl.log6306.sort.impl;
abstract class AbstractSort<E extends Comparable<E>> {
    private final List<ISortObserver<E>> listOfObservers;
    public AbstractSort() {
        this.listOfObservers = new ArrayList<ISortObserver<E>>();
    }
    public void addObserver(final ISortObserver<E> anObserver) {
         this.listOfObservers.add(anObserver);
    }
    protected int compareValues(final E[] values, final int position1, int position2) {
        this.notifyObserversOfAComparison(values,
            position1, values[position1],
            position2, values[position2]);
        return values[position1].compareTo(values[position2]);
    }
    private void notifyObserversOfAComparison(
        final E[] values, final int position1, final E value1,
        final int position2, final E value2) {
        final ComparisonEvent<E> event =
            new ComparisonEvent<E>(values, position1, value1, position2, value2);

        final Iterator<ISortObserver<E>> iterator = this.listOfObservers.iterator();
        while (iterator.hasNext()) {
            final ISortObserver<E> sortObserver = (ISortObserver<E>) iterator.next();
            sortObserver.valuesCompared(event);
        }
    }

    // ...
```

Implementation



Implementation

- The Composite overrides `addObserver()`

```
package ca.polymtl.gigl.log6306.sort.impl;

class TypeOfSort<E extends Comparable<E>> extends AbstractSort<E> implements ITypeOfSort<E> {
    private final List<ISort<E>> listOfSortAlgorithms;
    private final String typeName;

    public TypeOfSort(final String aTypeName) {
        this.listOfSortAlgorithms = new ArrayList<ISort<E>>();
        this.typeName = aTypeName;
    }
    @Override
    public void addObserver(final ISortObserver<E> anObserver) {
        final Iterator<ISort<E>> iterator = this.listOfSortAlgorithms.iterator();
        while (iterator.hasNext()) {
            final ISort<E> sortAlgorithm = (ISort<E>) iterator.next();
            sortAlgorithm.addObserver(anObserver);
        }
    }

    // ...
```

Usage

- The Client can observer the algorithms

```
package ca.polymtl.gigl.log6306;
public class Client {
    public static void main(final String[] args) {
        final List<String> l =
            Arrays.asList(new String[] { "Venus", "Earth", "Mars" });
        final SimpleObserver<String> observer = new SimpleObserver<String>();

        final ISort<String> s = Factory.getInstance().getBubbleSortAlgorithm();
        s.addObserver(observer);
        System.out.println(s.sort(l));

        final ISort<String> t = Factory.getInstance().getInternalSortAlgorithms();
        t.addObserver(observer);
        final ITypeOfSort<String> c = (ITypeOfSort<String>) t;

        // Use one specific sort algorithm...
        final ISortIterator<String> i = c.getSortAlgorithms();
        System.out.println(i.getNext().sort(l));
        // Use all sort algorithms...
        System.out.println(t.sort(l));
    }
}
```

Usage

- The Client can observer the algorithms

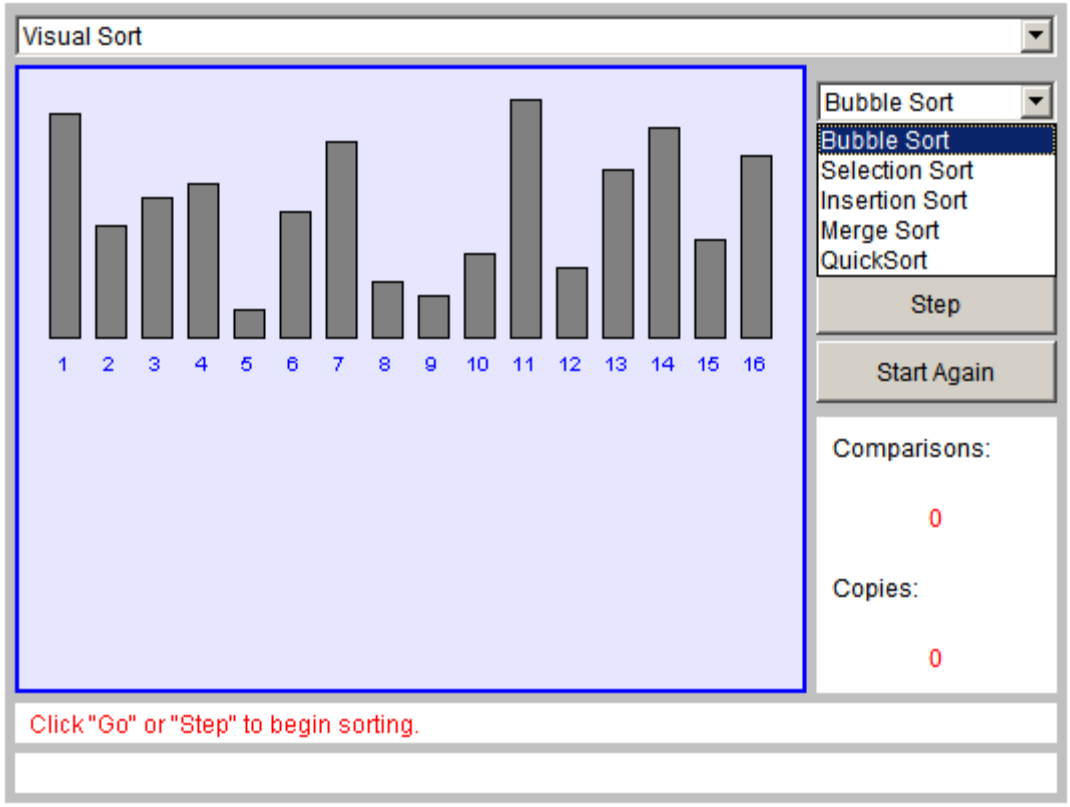
```
package ca.polymtl.gigl.log6306;
public class Client {
    public static void main(final String[] args) {
        final List<String> l =
            Arrays.asList(new String[] { "Venus", "Earth", "Mars" });
        final SimpleObserver<String> observer = new SimpleObserver<String>();

        final ISort<String> s = Factory.getInstance().getBubbleSortAlgorithm();
        s.addObserver(observer);
        System.out.println(s.sort(l));

        final ISort<String> t = Factory.getInstance().getInternalSortAlgorithms();
        t.addObserver(observer);
        final ITypeOfSort<String> c = (ITypeOfSort<String>) t;

        // Use one specific sort algorithm...
        final ISortIterator<String> i = c.getSortAlgorithms();
        System.out.println(i.getNext().sort(l));
        // Use all sort algorithms...
        System.out.println(t.sort(l));
    }
}
```

Usage

- The Client can observer the algorithms

```
...
Comparison of Venus with Earth
Swap of Venus with Earth
Comparison of Venus with Mars
Swap of Venus with Mars
Comparison of Earth with Mars
Comparison of Mars with Venus
[Earth, Mars, Venus]
...
```

Usage


Visual Sort
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Bubble Sort
Bubble Sort
Selection Sort
Insertion Sort
Merge Sort
QuickSort
Step
Start Again
Comparisons:
0
Copies:
0
Click "Go" or "Step" to begin sorting.

Conclusion

- The Observer design pattern allows objects to be notified of events **without strong dependencies** between the observers and the subjects
  - Subjects do not know who observe them
  - Observers only have access to information provided by the subjects
    - Could include “private” information

Conclusion

- The Observer design pattern is very much used in libraries and frameworks
  - `*Listener`s in Java
  - Several C++ frameworks
  - ...

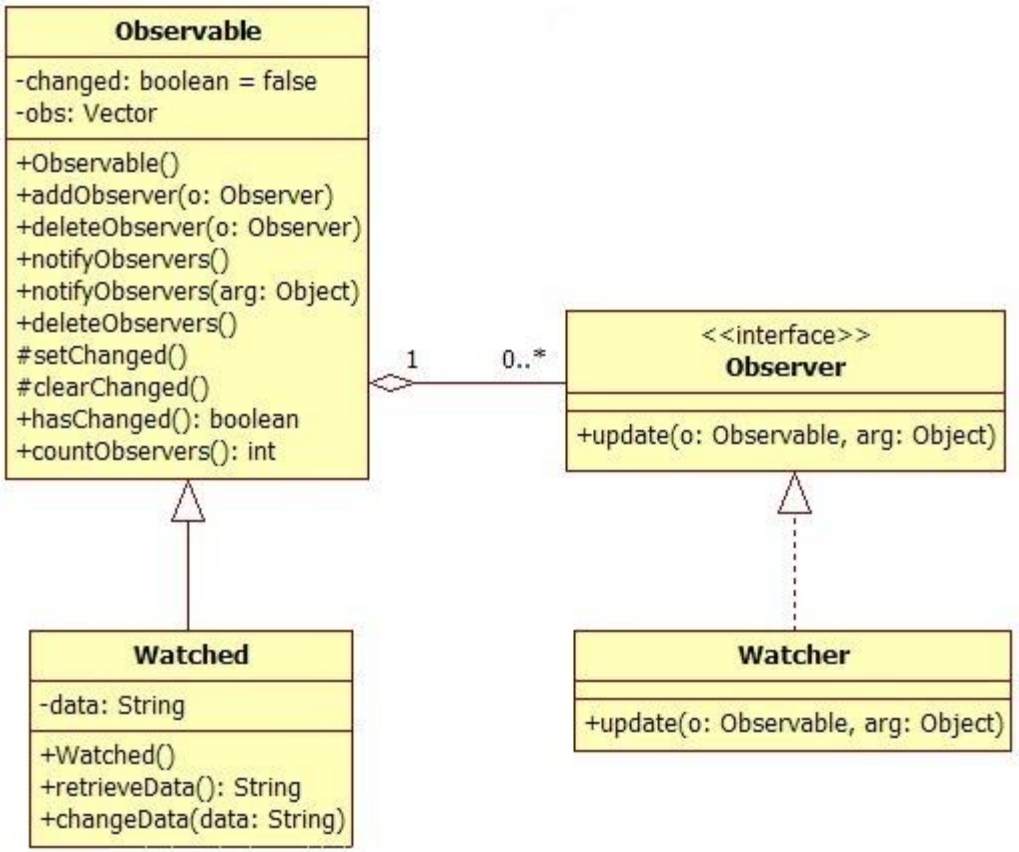
Conclusion

- User interfaces and their models sometimes use double observers
  - UI components observe the models
  - The models observe the UI components

  … beware of loops!

Other Implementation

- Implementation directly available in Java



Other Implementation

- Implementation directly available in Java
  - `Observable` is a concrete class
    - How to use it?
    - Why should it be final?
  - When notifying observers without arguments, the argument is set to `null`
    - Why is that a problem?
    - What alternative?

LOG6306 : Études empiriques sur les patrons logiciels

Foutse khomh

The Decorator DP

ÉCOLE POLYTECHNIQUE MONTRÉAL

연세대학교 YONSEI UNIVERSITY

Context

- From The Observer DP

Context

What if we want to sort lower-case?

```
package ca.polymtl.gigl.log6306;

public class Client {
    public static void main(final String[] args) {
        final List<String> l =
            Arrays.asList(new String[] { "Venus", "Earth", "Mars" });

        final ISort<String> s = Factory.getInstance().getBubbleSortAlgorithm();
        System.out.println(s.sort(l));

        final ISort<String> t = Factory.getInstance().getInternalSortAlgorithms();
        final ITypeOfSort<String> c = (ITypeOfSort<String>) t;

        // Use one specific sort algorithm...
        final ISortIterator<String> i = c.getSortAlgorithms();
        System.out.println(i.getNext().sort(l));

        // Use all sort algorithms...
        System.out.println(t.sort(l));
    }
}
```

Context

- Having a sort algorithm is interesting but we could also provide “typical” transformations pre- and post-sort?

Problem: Add/modify the behaviour of some methods of some objects at runtime

Solution: Decorator design pattern

Decorator (1/11)

“The important aspect of this pattern is that it lets decorators appear anywhere […]. That way clients generally can't tell the difference between a decorated component and an undecorated one, and so they don't depend at all on the decoration.”

[Gamma et al.]

Decorator (2/11)

- Name: Decorator

- Intent: “Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.”

Decorator (3/11)

- Motivation: “Sometimes we want to add responsibilities to **individual objects**, not to an entire class. […] One way to add responsibilities is with inheritance. […] This is inflexible, however, because the choice […] is made **statically**.”

Decorator (4/11)

- Motivation (cont'd): "A more flexible approach is to enclose the component in another object […]. The enclosing object is called a **decorator**. The decorator conforms to the interface of the component it decorates so that its presence is transparent to the component's clients."

Decorator (5/11)

- Motivation (cont'd): "The decorator **forwards requests** to the component and may perform additional actions […] before or after forwarding. Transparency lets you nest decorators **recursively**, thereby allowing an unlimited number of added responsibilities."

Decorator (6/11)

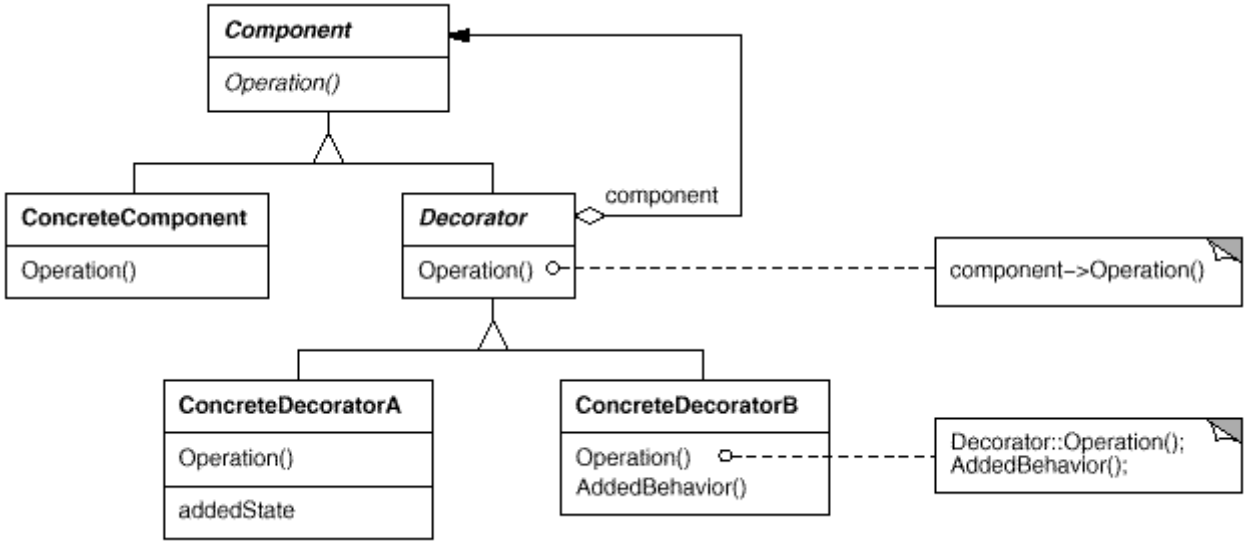
- Motivation (cont'd): "Decorator subclasses are **free to add operations for specific functionality**. For example, [the] [`ScrollDecorator.ScrollTo()`] operation lets other objects scroll the interface **if** they know there happens to be a ScrollDecorator object in the interface."

Decorator (7/11)

- Applicability
  - To add responsibilities to individual objects dynamically and transparently
  - To add responsibilities that can be withdrawn
  - When extension by subclassing is impractical

Decorator (8/11)

- Structure



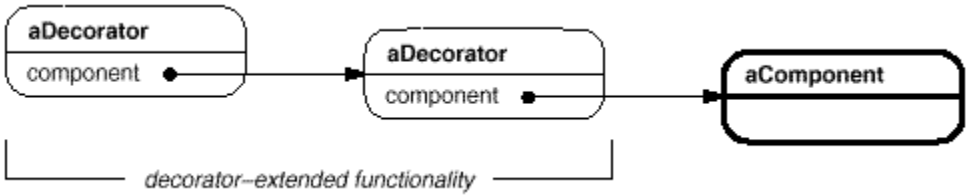
Decorator (9/11)

- Participants
  - Component
    - Defines the interface for objects that can have responsibilities added to them dynamically
  - ConcreteComponent
    - Defines an object to which additional responsibilities can be attached
  - Decorator
    - Maintains a reference to a Component object and defines an interface that conforms to Component's interface
  - ConcreteDecorator
    - Adds responsibilities to the component

Decorator (10/11)

- Collaborations

  “Decorator forwards requests to its Component object. It may optionally perform additional operations before and after forwarding the request.”



Decorator (11/11)

- Consequences
  - Provides more flexibility than static inheritance
  - Avoids feature-laden classes high up in the hierarchy
  - Means that the decorator and its component are not identical
  - Implies lots of little objects

Implementation

```
package ca.polymtl.gigl.log6306;

public class Client {
    public static void main(final String[] args) {
        final List<String> l = Arrays.asList(new String[] { "Venus", "Earth", "Mars" });
        final SimpleObserver<String> observer = new SimpleObserver<String>();

        final ISort<String> s = Factory.getInstance().getBubbleSortAlgorithm();
        s.addObserver(observer);
        System.out.println(s.sort(l));

        final ISort<String> d1 = new ToLowerCaseDecorator(s);
        d1.addObserver(observer);
        System.out.println(d1.sort(l));

        final ISort<String> d2 = new EncryptAfterSortingDecorator(s);
        d2.addObserver(observer);
        System.out.println(d2.sort(l));
    }
}
```

Implementation

```
Comparison of Venus with Earth
Swap of Venus with Earth
Comparison of Venus with Mars
Swap of Venus with Mars
Comparison of Earth with Mars
Comparison of Mars with Venus
[Earth, Mars, Venus]

Comparison of venus with earth
Swap of venus with earth
Comparison of venus with mars
Swap of venus with mars
Comparison of earth with mars
Comparison of mars with venus
[earth, mars, venus]

Comparison of venus with earth
Swap of venus with earth
Comparison of venus with mars
Swap of venus with mars
Comparison of earth with mars
Comparison of mars with venus
[96278602, 3344085, 112093821]
```

Implementation

- Two “decorable” methods
  - `addObserver(…)`
  - `sort(…)`

```
package ca.polymtl.gigl.log6306.sort;

import java.util.List;
import ca.polymtl.gigl.log6306.sort.observer.ISortObserver;

public interface ISort<E extends Comparable<E>> {
    List<E> sort(final List<E> aList);
    void addObserver(final ISortObserver<E> anObserver);
}
```

Implementation

- Two “decorable” methods
  - `addObserver(…)`
  - **`sort(…)`**

```
package ca.polymtl.gigl.log6306.sort;

import java.util.List;
import ca.polymtl.gigl.log6306.sort.observer.ISortObserver;

public interface ISort<E extends Comparable<E>> {
    List<E> sort(final List<E> aList);
    void addObserver(final ISortObserver<E> anObserver);
}
```

Implementation

The Decorator handles observers

Decorators perform the sorting

```
package ca.polymtl.gigl.log6306.sort.impl;

import java.util.List;
import ca.polymtl.gigl.log6306.sort.ISort;
import ca.polymtl.gigl.log6306.sort.observer.ISortObserver;

public abstract class SortDecorator<E extends Comparable<E>> implements ISort<E> {
    private final ISort<E> decoratedSortAlgorithm;

    public SortDecorator(final ISort<E> aSortAlgorithm) {
        this.decoratedSortAlgorithm = aSortAlgorithm;
    }
    @Override
    public final void addObserver(final ISortObserver<E> anObserver) {
        this.decoratedSortAlgorithm.addObserver(anObserver);
    }
    protected final ISort<E> getDecoratedSortAlgorithm() {
        return this.decoratedSortAlgorithm;
    }
    @Override
    public abstract List<E> sort(final List<E> aList);
}
```

Implementation

Modifies the input

- Decorators extends `SortDecorator` and implement `sort()`

```
package ca.polymtl.gigl.log6306.sort.decorators;

public class ToLowerCaseDecorator extends SortDecorator<String> {
    public ToLowerCaseDecorator(final ISort<String> aSortAlgorithm) {
        super(aSortAlgorithm);
    }

    @Override
    public List<String> sort(final List<String> aList) {
        final List<String> newList = new ArrayList<String>();
        final Iterator<String> iterator = aList.iterator();
        while (iterator.hasNext()) {
            final String s = iterator.next();
            newList.add(s.toLowerCase());
        }
        return this.getDecoratedSortAlgorithm().sort(newList);
    }
}
```

Implementation

Modifies the output

- Decorators extends `SortDecorator` and implement `sort()`

```
package ca.polymtl.gigl.log6306.sort.decorators;

public class EncryptAfterSortingDecorator extends SortDecorator<String> {
    public EncryptAfterSortingDecorator(final ISort<String> aSortAlgorithm) {
        super(aSortAlgorithm);
    }

    @Override
    public List<String> sort(final List<String> aList) {
        final List<String> sortedList = this.getDecoratedSortAlgorithm().sort(aList);
        final List<String> newList = new ArrayList<String>();
        final Iterator<String> iterator = sortedList.iterator();
        while (iterator.hasNext()) {
            final String s = iterator.next();
            newList.add(String.valueOf(s.hashCode()));
        }
        return newList;
    }
}
```

Usage

- The Client can declare and combine the decorators at will

```
package ca.polymtl.gigl.log6306;

public class Client {
    public static void main(final String[] args) {
        final List<String> l = Arrays.asList(new String[] { "Venus", "Earth", "Mars" });
        final SimpleObserver<String> observer = new SimpleObserver<String>();

        final ISort<String> s = Factory.getInstance().getBubbleSortAlgorithm();
        s.addObserver(observer);
        System.out.println(s.sort(l));

        final ISort<String> d1 = new ToLowerCaseDecorator(s);
        d1.addObserver(observer);
        System.out.println(d1.sort(l));

        final ISort<String> d2 = new EncryptAfterSortingDecorator(d1);
        d2.addObserver(observer);
        System.out.println(d2.sort(l));
    }
}
```

Usage

- The Client can declare and combine the decorators at will

```
...
Comparison of venus with earth
Swap of venus with earth
Comparison of venus with mars
Swap of venus with mars
Comparison of earth with mars
Comparison of mars with venus
[96278602, 3344085, 112093821]
```

Usage

- The Client can declare and combine the decorators at will, including Composites

```
package ca.polymtl.gigl.log6306;

public class Client {
    public static void main(final String[] args) {
        final List<String> l = Arrays.asList(new String[] { "Venus", "Earth", "Mars" });
        final SimpleObserver<String> observer = new SimpleObserver<String>();

        // ...

        final ISort<String> t = Factory.getInstance().getInternalSortAlgorithms();
        t.addObserver(observer);
        final ISort<String> d3 = new ToLowerCaseDecorator(t);
        d3.addObserver(observer);
        System.out.println(d3.sort(l));
    }
}
```

Usage

- The Client can declare and combine the decorators at will, including Composites

```
...
Comparison of venus with earth
Swap of venus with earth
Comparison of venus with mars
Swap of venus with mars
Comparison of earth with mars
Comparison of mars with venus
[earth, mars, venus]
```

Conclusion

- The Decorator design pattern allows modifying the behaviour of methods of objects at runtime
  - Without subclassing
  - Pre- and post-treatments
  - Allows to intercept and to proxy methods, see reflection and aspect-oriented programming

Conclusion

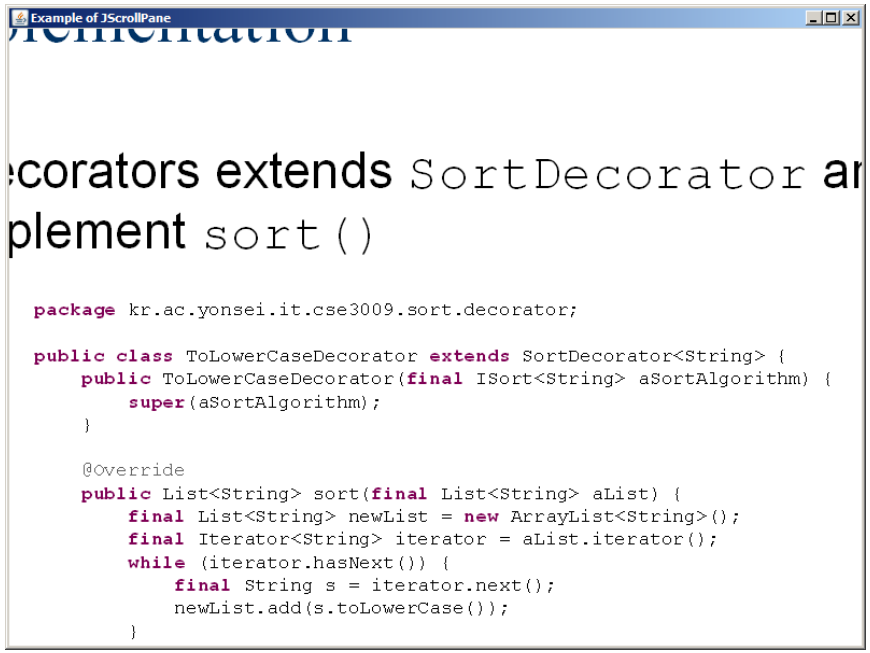
- The Decorator design pattern allows modifying the behaviour of methods of objects at runtime
  - Without subclassing
  - Pre- and post-treatments
  - Allows to intercept and to proxy methods, see **reflection** and **aspect-oriented programming**

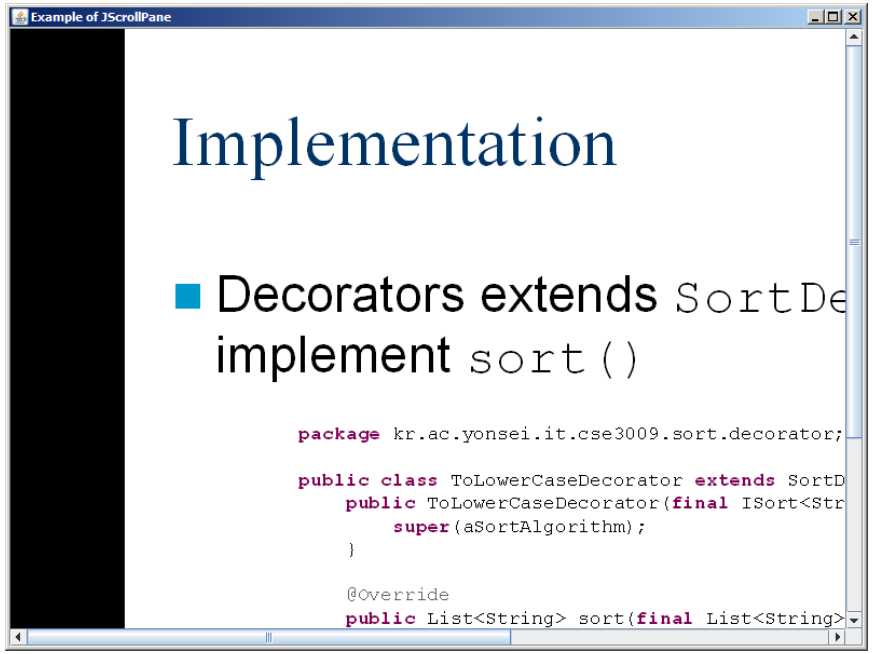
Conclusion

- Yet again, we added one level of indirection to provide more flexibility to the design and implementation!

Conclusion

- The Decorator design pattern is very much used in libraries and frameworks
  - `JScrollPane` in Java





Decorates label with scrollbars

Conclusion

- The Decorator design pattern is very much used in libraries and frameworks
  - `JScrollPane` in Java

```
public class Example {
    public static void main(final String[] args) {
        EventQueue.invokeLater(new Runnable() {
            public void run() {
                final JFrame frame = new JFrame("Example of JScrollPane");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setLocation(50, 50);
                frame.setSize(800, 600);
                frame.setVisible(true);
                try {
                    final Image image = ImageIO.read(new File("rsc/Slide.png"));
                    final JLabel imageLabel = new JLabel(new ImageIcon(image));
                    final JScrollPane scrollPane = new JScrollPane(imageLabel);
                    frame.getContentPane().add(scrollPane);
                // ...
```

Conclusion

- Caveat: what is happening?

```
package ca.polymtl.gigl.log6306.sort.decorators;

public class YetAnotherDecorator extends SortDecorator<String> {
    public YetAnotherDecorator(final ISort<String> aSortAlgorithm) {
        super(aSortAlgorithm);
    }

    @Override
    public List<String> sort(final List<String> aList) {
        final List<String> sortedList = this.getDecoratedSortAlgorithm().sort(aList);
        final List<String> newList = new ArrayList<String>();
        final Iterator<String> iterator = aList.iterator();
        while (iterator.hasNext()) {
            final String s = iterator.next();
            newList.add(String.valueOf(s.hashCode()));
        }
        return newList;
    }
}
```

Conclusion

We are not sorting the list!
(Not really anyways…)

- Caveat: what is happening?

```
package ca.polymtl.gigl.log6306.sort.decorators;

public class YetAnotherDecorator extends SortDecorator<String> {
    public YetAnotherDecorator(final ISort<String> aSortAlgorithm) {
        super(aSortAlgorithm);
    }

    @Override
    public List<String> sort(final List<String> aList) {
        final List<String> sortedList = this.getDecoratedSortAlgorithm().sort(aList);
        final List<String> newList = new ArrayList<String>();
        final Iterator<String> iterator = aList.iterator();
        while (iterator.hasNext()) {
            final String s = iterator.next();
            newList.add(String.valueOf(s.hashCode()));
        }
        return newList;
    }
}
```

LOG6306 : Études empiriques sur les patrons logiciels

Foutse Khomh

The Iterator DP

Context

- From The Composite DP

Context

```
package ca.polymtl.gigl.log6306.sort.impl;
class TypeOfSort<E extends Comparable<E>> extends AbstractSort<E> implements ITypeOfSort<E> {
    private final List<ISort<E>> listOfSortAlgorithms;
    private final String typeName;

    public TypeOfSort(final String aTypeName) {
        this.listOfSortAlgorithms = new ArrayList<ISort<E>>();
        this.typeName = aTypeName;
    }
    public void addSortAlgorithm(final ISort<E> aSortAlgorithm) {
        this.listOfSortAlgorithms.add(aSortAlgorithm);
    }
    public String getTypeName() {
        return this.typeName;
    }

    // ... Continued on the next slide ...

}
```

Context

```
package ca.polymtl.gigl.log6306.sort.impl;
class TypeOfSort<E extends Comparable<E>> extends AbstractSort<E> implements ITypeOfSort<E> {

    // ... Continued from the previous slide ...

    public List<E> sort(final List<E> aList) {
        // Call each sort algorithm of this type one after the other...
        final Iterator<ISort<E>> iterator = this.listOfSortAlgorithms.iterator();
        List<E> sortedList = null;
        while (iterator.hasNext()) {
            final ISort<E> sortAlgorithm = (ISort<E>) iterator.next();
            sortedList = sortAlgorithm.sort(aList);
        }
        return sortedList;
    }
    public List<ISort<E>> getSortAlgorithms() {
        return this.listOfSortAlgorithms;
    }
}
```

Context

Access to objects in the tree structure

```
package ca.polymtl.gigl.log6306;

public class Client {
    public static void main(final String[] args) {
        final List<String> l =
            Arrays.asList(new String[] { "Venus", "Earth", "Mars" });

        final ISort<String> s = Factory.getInstance().getBubbleSortAlgorithm();
        System.out.println(s.sort(l));

        final ISort<String> t = Factory.getInstance().getInternalSortAlgorithms();
        System.out.println(t.sort(l));

        final ITypeOfSort<String> c = (ITypeOfSort<String>) t;
        final List<ISort<String>> i = c.getSortAlgorithms();
        System.out.println(i.get(0));
    }
}
```

Context

```
package ca.polymtl.gigl.log6306;

public class Client {
    public static void main(final String[] args) {
        final List<String> l =
            Arrays.asList(new String[] { "Venus", "Earth", "Mars" });

        final ISort<String> s = Factory.getInstance().getBubbleSortAlgorithm();
        System.out.println(s.sort(l));

        final ISort<String> t = Factory.getInstance().getInternalSortAlgorithms();
        System.out.println(t.sort(l));

        final ITypeOfSort<String> c = (ITypeOfSort<String>) t;
        final List<ISort<String>> i = c.getSortAlgorithms();
        System.out.println(i.get(0));
        System.out.println(i.remove(0));
    }
}
```

Context

"Leak" of mutable list, clients could change it!

```
package ca.polymtl.gigl.log6306.sort.impl;
class TypeOfSort<E extends Comparable<E>> extends AbstractSort<E> implements ITypeOfSort<E> {

    // ... Continued from the previous slide ...

    public List<E> sort(final List<E> aList) {
        // Call each sort algorithm of this type one after the other...
        final Iterator<ISort<E>> iterator = this.listOfSortAlgorithms.iterator();
        List<E> sortedList = null;
        while (iterator.hasNext()) {
            final ISort<E> sortAlgorithm = (ISort<E>) iterator.next();
            sortedList = sortAlgorithm.sort(aList);
        }
        return sortedList;
    }
    public List<ISort<E>> getSortAlgorithms() {
        return this.listOfSortAlgorithms;
    }
}
```

Context

- Returning a mutable collection (or copy thereof) is dangerous (or waste resources)

Problem: How to let clients access algorithms, hiding the underlying collection?
Solution: Iterator design pattern

Iterator (1/9)

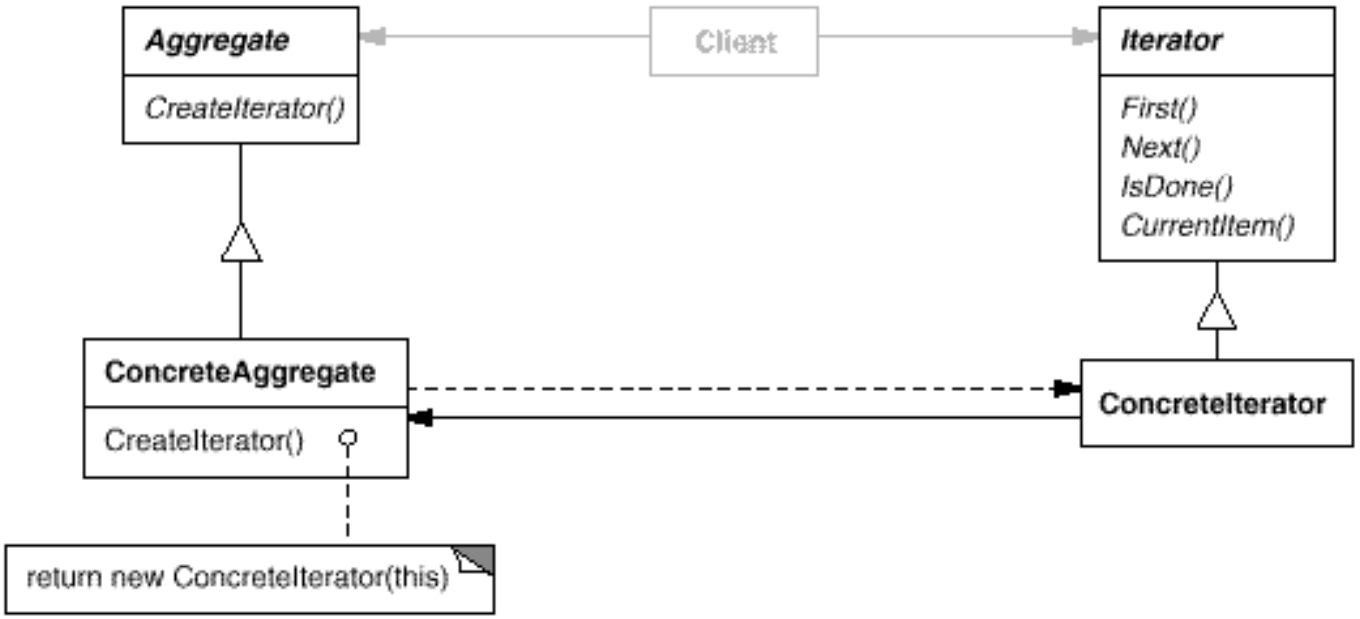
- Intent
  - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation

Iterator (2/9)

- Applicability
  - To access an aggregate object's contents without exposing its internal representation
  - To support multiple traversals of aggregate objects
  - To provide a uniform interface for traversing different aggregate structures

Iterator (3/9)

- Design



Iterator (4/9)

- Implementation

```
package ca.polymtl.gigl.log6306;
public interface ISortIterator<E> {
    boolean hasNext();
    ISort<E> getNext();
}

package ca.polymtl.gigl.log6306.sorts.impl;
public class ConcreteSortIterator<E> implements ISortIterator<E> {
    private final List<ISort<E>> privateCopyOfList;
    private int cursor;
    public ConcreteSortIterator(final List<ISort<E>> aListOfItems) {
        this.privateCopyOfList = aListOfItems;
    }
    public boolean hasNext() {
        return this.cursor < this.privateCopyOfList.size() - 1;
    }
    public ISort<E> getNext() {
        final ISort<E> currentSortAlgorithm =
            this.privateCopyOfList.get(this.cursor);
        this.cursor++;
        return currentSortAlgorithm;
    }
}
```

Iterator (5/9)

- Implementation

```
package ca.polymtl.gigl.log6306;

public class Client {
    public static void main(final String[] args) {
        final List<String> l =
            Arrays.asList(new String[] { "Venus", "Earth", "Mars" });

        final ISort<String> s = Factory.getInstance().getBubbleSortAlgorithm();
        System.out.println(s.sort(l));

        final ISort<String> t = Factory.getInstance().getInternalSortAlgorithms();
        System.out.println(t.sort(l));

        final ITypeOfSort<String> c = (ITypeOfSort<String>) t;
        final ISortIterator<String> i = c.getSortAlgorithms();
        System.out.println(i.hasNext());
    }
}
```

Iterator (6/9)

- Implementation

```
public class TypeOfSort<E> extends AbstractSort<E> implements ISort<E> {
    private final List<ISort<E>> listOfSortAlgorithms;
    private final String typeName;
    public TypeOfSort(final String aTypeName) {
        this.listOfSortAlgorithms = new ArrayList<ISort<E>>();
        this.typeName = aTypeName;
    }

    // ...

    public ISortIterator<E> getSortAlgorithms() {
        return new ConcreteSortIterator<E>(this.listOfSortAlgorithms);
    }
}
```

Iterator

Information hiding:
- Packaging
- Accessing

- Implementation

```
public class TypeOfSort<E> extends AbstractSort<E> implements ISort<E> {
    private final List<ISort<E>> listOfSortAlgorithms;
    private final String typeName;
    public TypeOfSort(final String aTypeName) {
        this.listOfSortAlgorithms = new ArrayList<ISort<E>>();
        this.typeName = aTypeName;
    }

    // ...

   public ISortIterator<E> getSortAlgorithms() {
      return new ConcreteSortIterator<E>(this.listOfSortAlgorithms);
   }
}
```

Iterator

Beware of co-modifications

- Implementation

```
package ca.polymtl.gigl.log6306.sorts.impl;

import java.util.List;
import ca.polymtl.gigl.log6306.Iterator;

public class ConcreteSortIterator<E> implements ISortIterator<E> {
    private final List<ISort<E>> privateCopyOfList;
    private int cursor;
    public ConcreteSortIterator(final List<ISort<E>> aListOfItems) {
        this.privateCopyOfList = aListOfItems;
    }
    public boolean hasNext() {
        return this.cursor < this.privateCopyOfList.size() - 1;
    }
    public ISort<E> getNext() {
        final ISort<E> currentSortAlgorithm =
            this.privateCopyOfList.get(this.cursor);
        this.cursor++;
        return currentSortAlgorithm;
    }
}
```

Iterator

Lazy initialisation, on-demand instantiation

- Implementation

```
package ca.polymtl.gigl.log6306.sorts.impl;

import java.util.List;
import ca.polymtl.gigl.log6306.Iterator;

public class ConcreteSortIterator<E> implements ISortIterator<E> {
    private final List<ISort<E>> privateCopyOfList;
    private int cursor;
    public ConcreteSortIterator(final List<ISort<E>> aListOfItems) {
        this.privateCopyOfList = aListOfItems;
    }
    public boolean hasNext() {
        return this.cursor < this.privateCopyOfList.size() - 1;
    }
    public ISort<E> getNext() {
        final ISort<E> currentSortAlgorithm =
            this.privateCopyOfList.get(this.cursor);
        this.cursor++;
        return currentSortAlgorithm;
    }
}
```

Conclusion

- The Iterator design pattern
  - Again, adds one level of indirection!
  - Hides underlying collection
  - Provides access to elements
  - Allows
    - Lazy initialisation / instantiation
    - Caching and performance tuning

Other Implementation

- Using Java `Iterator` interface

```
public Iterator<ISort<E>> getSortAlgorithms() {
    return this.listOfSortAlgorithms.iterator();
}
```

LOG6306 : Études empiriques sur les patrons logiciels

Foutse Khomh

The Template Method DP

Context

- From The Decorator DP

Context

We are not sorting the list!
(Not really anyways…)

```
package ca.polymtl.gigl.log6306.sort.decorators;

public class YetAnotherDecorator extends SortDecorator<String> {
    public YetAnotherDecorator(final ISort<String> aSortAlgorithm) {
        super(aSortAlgorithm);
    }

    @Override
    public List<String> sort(final List<String> aList) {
        final List<String> sortedList = this.getDecoratedSortAlgorithm().sort(aList);
        final List<String> newList = new ArrayList<String>();
        final Iterator<String> iterator = aList.iterator();
        while (iterator.hasNext()) {
            final String s = iterator.next();
            newList.add(String.valueOf(s.hashCode()));
        }
        return newList;
    }
}
```

Context

- The Client may misuse, overuse, or not use at all the methods provided by a framework

Problem: Let the framework decided when/how to call some user-defined methods

Solution: Template Method design pattern

Template Method (1/11)

"[In] a framework […] the methods defined by the user […] will often be called from within the framework itself, rather than from the user's […] code. The framework […] plays the role of the main program [and coordinates method calls]. This inversion of control gives frameworks the power to serve as extensible skeletons. The methods supplied by the user tailor the generic algorithms defined in the framework […]."

—Ralph Johnson and Brian Foote

Template Method (1/11)

"[In] a **framework** […] the methods defined by the user […] will often be called **from within** the framework itself, rather than from the user's […] code. The framework […] plays the role of the main program [**and coordinates method calls**]. This **inversion of control** gives frameworks the power to serve as **extensible skeletons**. The methods supplied by the user **tailor** the generic algorithms defined in the framework […]."

—Ralph Johnson and Brian Foote

Template Method (2/11)

- Name: Template Method

- Intent: “Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.”

Template Method (3/11)

- Motivation: “A template method defines an algorithm in terms of **abstract operations** that subclasses override to provide concrete behaviour.”

Template Method (4/11)

- Motivation (cont'd): "By defining some of the steps of an algorithm using abstract operations, the template method fixes their ordering, but it lets […] subclasses vary those steps to suit their needs."

Template Method (5/11)

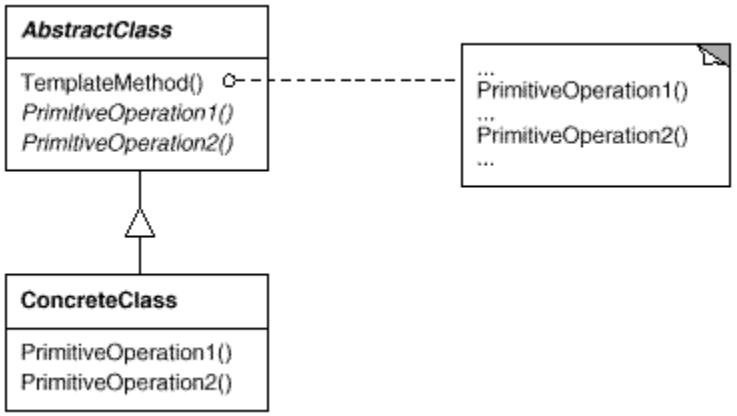
- Motivation (cont'd): "Template methods are a fundamental technique for code reuse. They are particularly important in class libraries [and frameworks], because they are the means for factoring out common behaviour."

Template Method (6/11)

- Applicability
  - To implement the invariant parts of an algorithm and let subclasses implement varying behaviour
  - To refactor to generalize as described by Opdyke and Johnson
  - To control subclasses extensions

Template Method (7/11)

- Structure

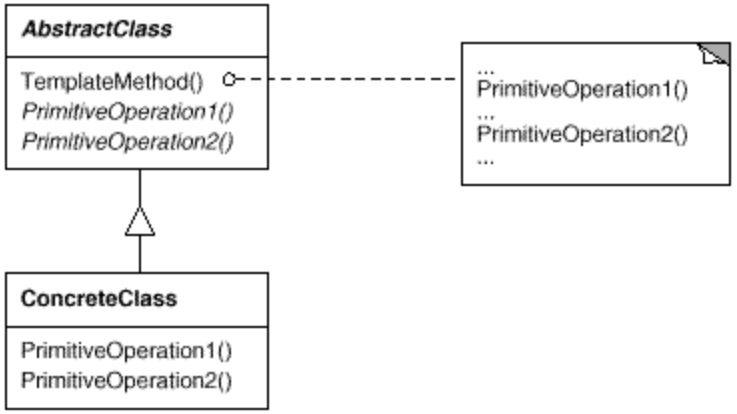


Template Method (8/11)

- Participants
  - AbstractClass
    - Defines abstract primitive operations that concrete subclasses define to implement steps of an algorithm
    - Implements a template method defining the **skeleton** of an algorithm
  - ConcreteClass
    - Implements the primitive operations to carry out subclass-specific steps of the algorithm

Template Method (9/11)

- Collaborations



"ConcreteClass relies on AbstractClass to implement the invariant steps of the algorithm."

Template Method (10/11)

- Consequences

  “Template methods lead to an **inverted control** structure that's sometimes referred to as “the Hollywood principle”, that is, “Don't call us, we'll call you”. This refers to how a parent class calls the operations of a subclass and not the other way around.”

Template Method (11/11)

- Consequences

  “It is important for template methods to specify which operations are **hooks** (*may* be overridden) and which are **abstract operations** (*must* be overridden). To reuse an abstract class effectively, subclass writers must understand which operations are designed for overriding.”

Frameworks vs. Libraries

- Framework are extensible skeletons
  - **Tailored by** users' methods
- Libraries are collection of classes
  - **Used in** users' methods

Frameworks vs. Libraries

- Remember
  - Reflection in OO Programming Languages
  - Interconnections
  - Subclassing

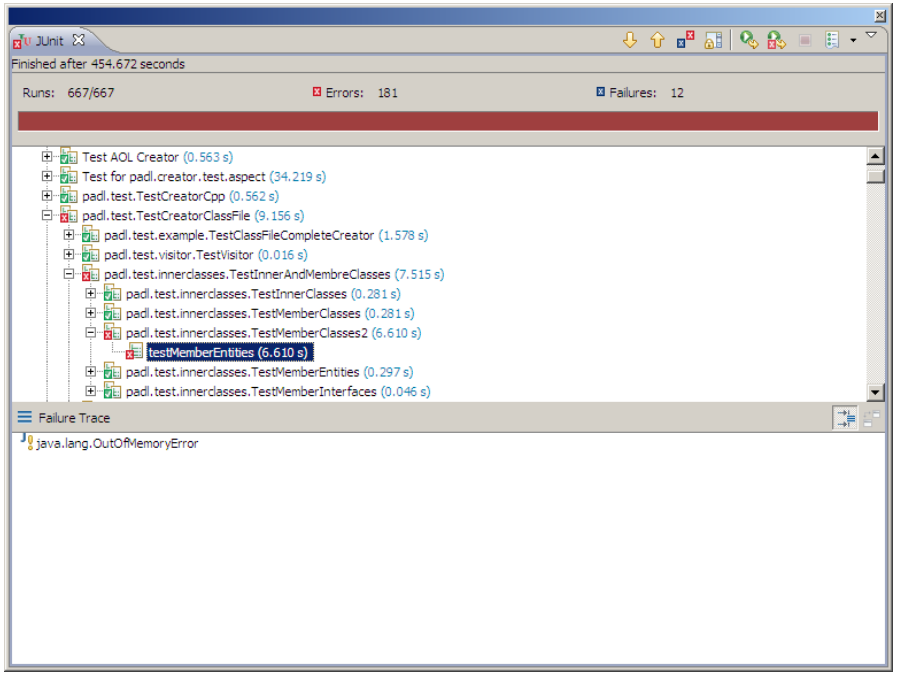
Frameworks vs. Libra

Also called extension points

- Subclassing
  - Hooks and templates
    - Hot spots = hooks
    - Frozen spots = templates
  - Hooks are typically
    - Default implementations
    - **Or** abstract methods
  - Templates use hooks

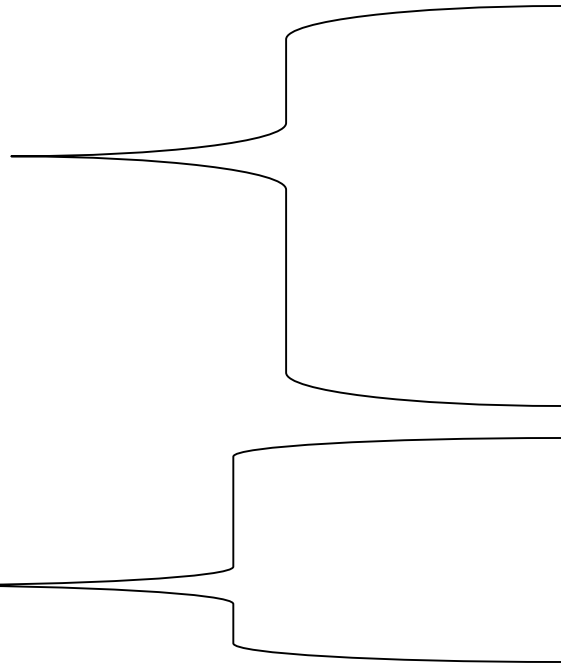
Frameworks vs. Libraries

- Subclassing
  - Hooks and templates
    - JUnit



Frameworks vs. Libraries

- Template
- Hooks



```
public abstract class TestCase
     extends Assert implements Test {

     public void runBare() throws Throwable {
          setUp();
          try {
               runTest();
          }
          finally {
               tearDown();
          }
     }

     protected void setUp() throws Exception {
     }

     protected void tearDown() throws Exception {
     }
     // ...
}
```

Frameworks vs. Libraries

- JUnit is a typical framework
  - Uses Template Method to control the testing process but let client implement their own tests
- Frameworks typically uses template methods to provide **inversion of control**

Frameworks vs. Libraries

- Inversion of control
  - Dependency injection
    - Constructor injection: the dependency is provided through a class constructor
    - Setter injection: the dependency is provided through setter method (called by the injector)
    - (Interface injection: the injectee implements a specific interface to perform setter injection)
  - Subclassing
    - Template Method design pattern

http://stackoverflow.com/questions/6550700/inversion-of-control-vs-dependency-injection

Frameworks vs. Libraries

- Inversion of control
  - Dependency injection
    - Visitor design pattern
  - Subclassing
    - Template Method design pattern

Frameworks vs. Libraries

- Inversion of control
  - Dependency injection
    - **Visitor design pattern**
  - Subclassing
    - **Template Method design pattern**

Implement

Identical to previous Decorator implementations

```
package ca.polymtl.gigl.log6306;

public class Client {
    public static void main(final String[] args) {
        final List<String> l = Arrays.asList(new String[] { "Venus", "Earth", "Mars" });
        final SimpleObserver<String> observer = new SimpleObserver<String>();

        final ISort<String> s = Factory.getInstance().getBubbleSortAlgorithm();
        s.addObserver(observer);
        System.out.println(s.sort(l));

        final ISort<String> d1 = new ToLowerCaseDecorator(s);
        d1.addObserver(observer);
        System.out.println(d1.sort(l));

        final ISort<String> d2 = new EncryptAfterSortingDecorator(d1);
        d2.addObserver(observer);
        System.out.println(d2.sort(l));

        final ISort<String> t = Factory.getInstance().getInternalSortAlgorithms();
        t.addObserver(observer);
        final ISort<String> d3 = new ToLowerCaseDecorator(t);
        d3.addObserver(observer);
        System.out.println(d3.sort(l));
    }
}
```

Implement

Identical to previous Decorator implementations

```
Comparison of Venus with Earth
Swap of Venus with Earth
Comparison of Venus with Mars
Swap of Venus with Mars
Comparison of Earth with Mars
Comparison of Mars with Venus
[Earth, Mars, Venus]

Comparison of venus with earth
Swap of venus with earth
Comparison of venus with mars
Swap of venus with mars
Comparison of earth with mars
Comparison of mars with venus
[earth, mars, venus]

Comparison of venus with earth
Swap of venus with earth
Comparison of venus with mars
Swap of venus with mars
Comparison of earth with mars
Comparison of mars with venus
[96278602, 3344085, 112093821]
```

Implementation

- Two “decorable” methods
  - `addObserver(…)`
  - `sort(…)`

```
package ca.polymtl.gigl.log6306.sort;

import java.util.List;
import ca.polymtl.gigl.log6306.sort.observer.ISortObserver;

public interface ISort<E extends Comparable<E>> {
    List<E> sort(final List<E> aList);
    void addObserver(final ISortObserver<E> anObserver);
}
```

Implementation

- Two “decorable” methods
  - `addObserver(…)`
  - **`sort(…)`**

```
package ca.polymtl.gigl.log6306.sort;

import java.util.List;
import ca.polymtl.gigl.log6306.sort.observer.ISortObserver;

public interface ISort<E extends Comparable<E>> {
    List<E> sort(final List<E> aList);
    void addObserver(final ISortObserver<E> anObserver);
}
```

Implementation

The Decorator handles observers

The Decorator performs the sorting

```
public abstract class SortDecorator<E extends Comparable<E>> implements ISort<E> {
    private final ISort<E> decoratedSortAlgorithm;

    public SortDecorator(final ISort<E> aSortAlgorithm) {
        this.decoratedSortAlgorithm = aSortAlgorithm;
    }
    @Override
    public final void addObserver(final ISortObserver<E> anObserver) {
        this.decoratedSortAlgorithm.addObserver(anObserver);
    }
    protected List<E> postProcessOutput(final List<E> sortedList) {
        return sortedList;
    }
    protected List<E> preProcessInput(final List<E> aList) {
        return aList;
    }
    @Override
    public final List<E> sort(final List<E> aList) {
        final List<E> preList    = this.preProcessInput(aList);
        final List<E> sortedList = this.decoratedSortAlgorithm.sort(preList);
        final List<E> postList   = this.postProcessOutput(sortedList);
        return postList;
    }
}
```

Implementation

The Decorator defines two hooks

```
public abstract class SortDecorator<E extends Comparable<E>> implements ISort<E> {
    private final ISort<E> decoratedSortAlgorithm;

    public SortDecorator(final ISort<E> aSortAlgorithm) {
        this.decoratedSortAlgorithm = aSortAlgorithm;
    }
    @Override
    public final void addObserver(final ISortObserver<E> anObserver) {
        this.decoratedSortAlgorithm.addObserver(anObserver);
    }
    protected List<E> postProcessOutput(final List<E> sortedList) {
        return sortedList;
    }
    protected List<E> preProcessInput(final List<E> aList) {
        return aList;
    }
    @Override
    public final List<E> sort(final List<E> aList) {
        final List<E> preList    = this.preProcessInput(aList);
        final List<E> sortedList = this.decoratedSortAlgorithm.sort(preList);
        final List<E> postList   = this.postProcessOutput(sortedList);
        return postList;
    }
}
```

Implementation

Modifies the input

- Decorators extends `SortDecorator` and implement `sort()`, and override the default implementations of the hooks at will

```
package ca.polymtl.gigl.log6306.sort.decorators;
public class ToLowerCaseDecorator extends SortDecorator<String> {
    public ToLowerCaseDecorator(final ISort<String> aSortAlgorithm) {
        super(aSortAlgorithm);
    }

    @Override
    protected List<String> preProcessInput(final List<String> aList) {
        final List<String> newList = new ArrayList<String>();
        final Iterator<String> iterator = aList.iterator();
        while (iterator.hasNext()) {
            final String s = iterator.next();
            newList.add(s.toLowerCase());
        }
        return newList;
    }
}
```

Implementation

Modifies the output

- Decorators extends `SortDecorator` and implement `sort()`, and override the default implementations of the hooks at will

```
package ca.polymtl.gigl.log6306.sort.decorators;
public class EncryptAfterSortingDecorator extends SortDecorator<String> {
    public EncryptAfterSortingDecorator(final ISort<String> aSortAlgorithm) {
        super(aSortAlgorithm);
    }

    @Override
    protected List<String> postProcessOutput(final List<String> sortedList) {
        final List<String> newList = new ArrayList<String>();
        final Iterator<String> iterator = sortedList.iterator();
        while (iterator.hasNext()) {
            final String s = iterator.next();
            newList.add(String.valueOf(s.hashCode()));
        }
        return newList;
    }
}
```

Usage

- The Client can declare and combine the decorators at will

```
package ca.polymtl.gigl.log6306;

public class Client {
    public static void main(final String[] args) {
        final List<String> l = Arrays.asList(new String[] { "Venus", "Earth", "Mars" });
        final SimpleObserver<String> observer = new SimpleObserver<String>();

        final ISort<String> s = Factory.getInstance().getBubbleSortAlgorithm();
        s.addObserver(observer);
        System.out.println(s.sort(l));

        final ISort<String> d1 = new ToLowerCaseDecorator(s);
        d1.addObserver(observer);
        System.out.println(d1.sort(l));

        final ISort<String> d2 = new EncryptAfterSortingDecorator(d1);
        d2.addObserver(observer);
        System.out.println(d2.sort(l));
    }
}
```

Usage

- The Client can declare and combine the decorators at will

```
...
Comparison of venus with earth
Swap of venus with earth
Comparison of venus with mars
Swap of venus with mars
Comparison of earth with mars
Comparison of mars with venus
[96278602, 3344085, 112093821]
```

Usage

- The Client can declare and combine the decorators at will, including Composites

```
package ca.polymtl.gigl.log6306;

public class Client {
    public static void main(final String[] args) {
        final List<String> l = Arrays.asList(new String[] { "Venus", "Earth", "Mars" });
        final SimpleObserver<String> observer = new SimpleObserver<String>();

        // ...

        final ISort<String> t = Factory.getInstance().getInternalSortAlgorithms();
        t.addObserver(observer);
        final ISort<String> d3 = new ToLowerCaseDecorator(t);
        d3.addObserver(observer);
        System.out.println(d3.sort(l));
    }
}
```

Usage

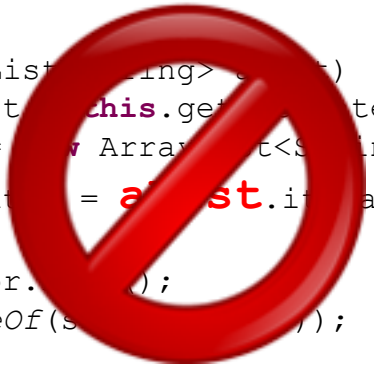
- The Client can declare and combine the decorators at will, including Composites

```
...
Comparison of venus with earth
Swap of venus with earth
Comparison of venus with mars
Swap of venus with mars
Comparison of earth with mars
Comparison of mars with venus
[earth, mars, venus]
```

Usage

- The implementors of Decorator cannot forget to sort the list

```
package ca.polymtl.gigl.log6306.sort.decorators;

public class YetAnotherDecorator extends SortDecorator<String> {
    public YetAnotherDecorator(final ISort<String> aSortAlgorithm) {
        super(aSortAlgorithm);
    }

    @Override
    public List<String> sort(final List<String> aList) {
        final List<String> sortedList = this.getDecoratedSortAlgorithm().sort(aList);
        final List<String> newList = new ArrayList<String>();
        final Iterator<String> iterator = aList.iterator();
        while (iterator.hasNext()) {
            final String s = iterator.next();
            newList.add(String.valueOf(s));
        }
        return newList;
    }
}
```



Usage

- The implementors of Decorator cannot forget to sort the list

```
package ca.polymtl.gigl.log6306.sort.decorators;

public class YetAnotherDecorator extends SortDecorator<String> {
    public YetAnotherDecorator(final ISort<String> aSortAlgorithm) {
        super(aSortAlgorithm);
    }

    @Override
    protected List<String> postProcessOutput(final List<String> sortedList) {
        // Cannot forget to sort the list first!
        final List<String> newList = new ArrayList<String>();
        final Iterator<String> iterator = sortedList.iterator();
        while (iterator.hasNext()) {
            final String s = iterator.next();
            newList.add(String.valueOf(s.hashCode()));
        }
        return newList;
    }
}
```

Conclusion

- The Template Method design pattern defines
  - The invariant steps of an algorithms
  - **Hook methods** that are part of the steps and **may** be overridden by clients
  - **Abstract methods** that are part of the steps and **must** be overridden by clients

Conclusion

- The Template Method design pattern is an implementation of **inversion of control**
  - However, must subclass the template class
- Another possible implementation is **dependency injection**

Conclusion

- Some concrete uses
  - `java.util.AbstractList.addAll(int, Collection<? extends E>)`

```
// ...
public void add(int index, E element) {
    throw new UnsupportedOperationException();
}
// ...
public boolean addAll(int index, Collection<? extends E> c) {
    rangeCheckForAdd(index);
    boolean modified = false;
    for (E e : c) {
        add(index++, e);
        modified = true;
    }
    return modified;
}
// ...
```

http://stackoverflow.com/questions/1553856/where-should-we-use-template-method-pattern

Conclusion

- Some concrete uses
  - `javax.servlet.http.HttpServlet`

```
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {

    String protocol = req.getProtocol();
    String msg = lStrings.getString("http.method_post_not_supported");
    if (protocol.endsWith("1.1")) {
        resp.sendError(HttpServletResponse.SC_METHOD_NOT_ALLOWED, msg);
    }
    else {
        resp.sendError(HttpServletResponse.SC_BAD_REQUEST, msg);
    }
}
```

Called in `protected void service(HttpServletRequest, HttpServletResponse)`, called in `public void service(ServletRequest, ServletResponse)`

http://stackoverflow.com/questions/1553856/where-should-we-use-template-method-pattern

LOG6306 : Études empiriques sur les patrons logiciels

Foutse Khomh

The Extension DP

ÉCOLE POLYTECHNIQUE MONTRÉAL

연세대학교 YONSEI UNIVERSITY

Context

- From The Template Method DP

Context

The decorator is “around” of the sort algorithm

```
package ca.polymtl.gigl.log6306;

public class Client {
    public static void main(final String[] args) {
        final List<String> l = Arrays.asList(new String[] { "Rick Deckard", "Roy
            Batty", "Harry Bryant", "Hannibal Chew", "Gaff", "Holden", "Leon
            Kowalski", "Taffey Lewis", "Pris", "Rachael", "J.F. Sebastian", "Dr.
            Eldon Tyrell", "Zhora", "Hodge", "Mary" });
        final SimpleObserver<String> observer = new SimpleObserver<String>();

        final ISort<String> t = Factory.getInstance().getInternalSortAlgorithms();
        t.addObserver(observer);
        final ISort<String> d3 = new ToLowerCaseDecorator(t);
        d3.addObserver(observer);
        System.out.println(d3.sort(l));
    }
}
```

Context

- The Client wraps the sort algorithm in the decorators of its choice when the algorithm should provide the “decorators”

Problem: Let the “decorated” objects decide the extensions that they offer

Solution: Extension Object design pattern

Extension Object (1/11)

"For some abstractions it is difficult to anticipate their complete interface since different clients can require a different view on the abstraction. Combining all the operations and state that the different clients need into a single interface results in a bloated interface. Such interfaces are difficult to maintain and understand. Moreover, a change to a client specific part of an interface can affect other clients that use the same abstraction."

http://www.mif.vu.lt/~plukas/resources/Extension%20Objects/ExtensionObjectsPattern%20Gamma96.pdf

Extension Object (1/11)

"For some abstractions it is difficult **to anticipate** their complete interface since different clients can require a **different view** on the abstraction. Combining all the operations and state that the different clients need into a single interface results in a **bloated interface**. Such interfaces are difficult to maintain and understand. Moreover, a change to a client specific part of an interface **can affect other** clients that use the same abstraction."

http://www.mif.vu.lt/~plukas/resources/Extension%20Objects/ExtensionObjectsPattern%20Gamma96.pdf

Extension Object (2/11)

- Name: Extension Object

- Intent: “Anticipate that an object’s interface needs to be extended in the future. Additional interfaces are defined by extension objects.”

Extension Object (3/11)

- Motivation: “The idea of the Extension Objects pattern is to anticipate such extensions. It proposes to package the [extension] in a separate object. Clients that want to use this extended interface can query whether a component supports it.”

Extension Object (4/11)

- Motivation (cont'd): "ComponentExtension is the common base class for extensions. It provides only a minimal interface used to manage the extension itself."

Extension Object (5/11)

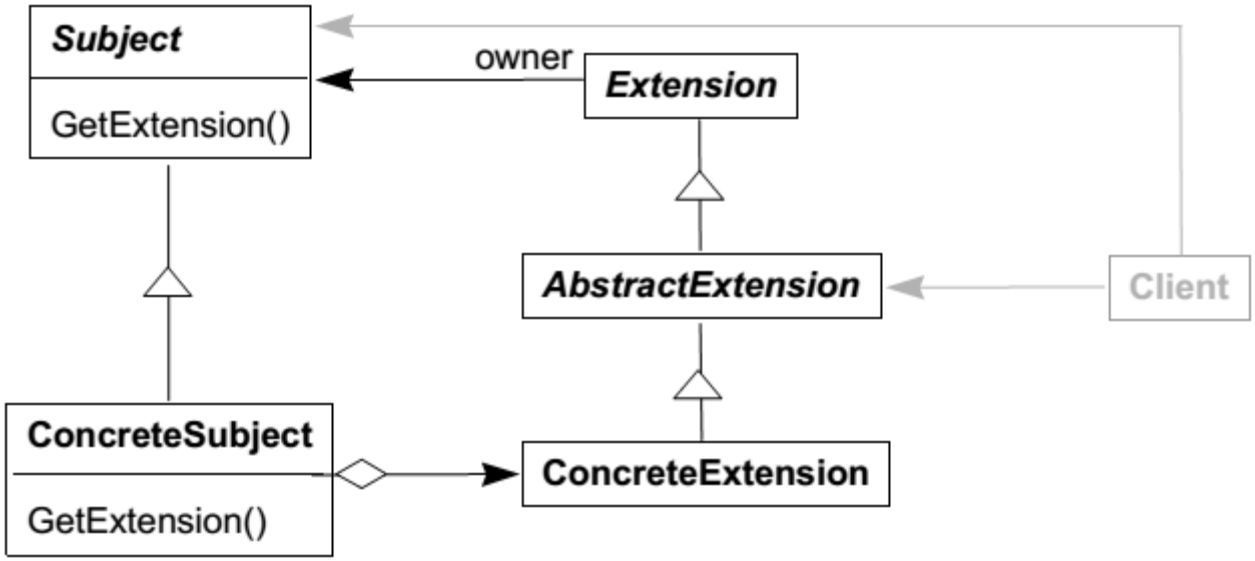
- Motivation (cont'd): "Extensions themselves aren't useful—there needs to be a way to find out whether a component supports a specific extension. […] [W]e [can] name an extension with a simple string."

Extension Object (6/11)

- Applicability
  - To support the addition of new or unforeseen interfaces to existing classes
  - To limit impact to clients that do not need these new interfaces
  - To give different roles to a key abstraction for different clients
  - To extend a class with new behaviour without subclassing from it

Extension Object (7/11)

- Structure



Extension Object (8/11)

- Participants
  - Subject
    - Defines the identity of an abstraction
    - Declares the interface to query whether an object has a particular extension
  - ConcreteSubject
    - Implement the operation to return an extension object when the client asks for it
  - Extension
    - Defines some support for managing extensions themselves
    - Knows its owning subject
  - AbstractExtension
    - Declares the interface for a specific extension
  - ConcreteExtension
    - Implements the extension interface for a particular subject

Extension Object (9/11)

- Collaborations
  - A client asks a Subject for a specific extension
  - If the extension exists, then the Subject returns the corresponding extension object
  - The client uses the extension object to access additional functionalities

Extension Object (10/11)

- Consequences
  - Extension Objects facilitates adding interfaces
  - No bloated class interfaces for key abstractions
  - Modeling of different roles of key abstractions in different subsystems
  - Clients become more complex
  - Abuse of extensions for interfaces that should be explicitly modeled

Extension Object (11/11)

- Consequences
  - Decorator
    - Client control the decorators that they use
    - Decorated objects have different identities
  - Extension objects
    - Objects control the extensions that they provide
      - On-demand instantiation (Singleton)
      - Un-loading possible
    - Extended objects keep their identities

Implementat

Extensions do not depend on the type of the objects being sorted

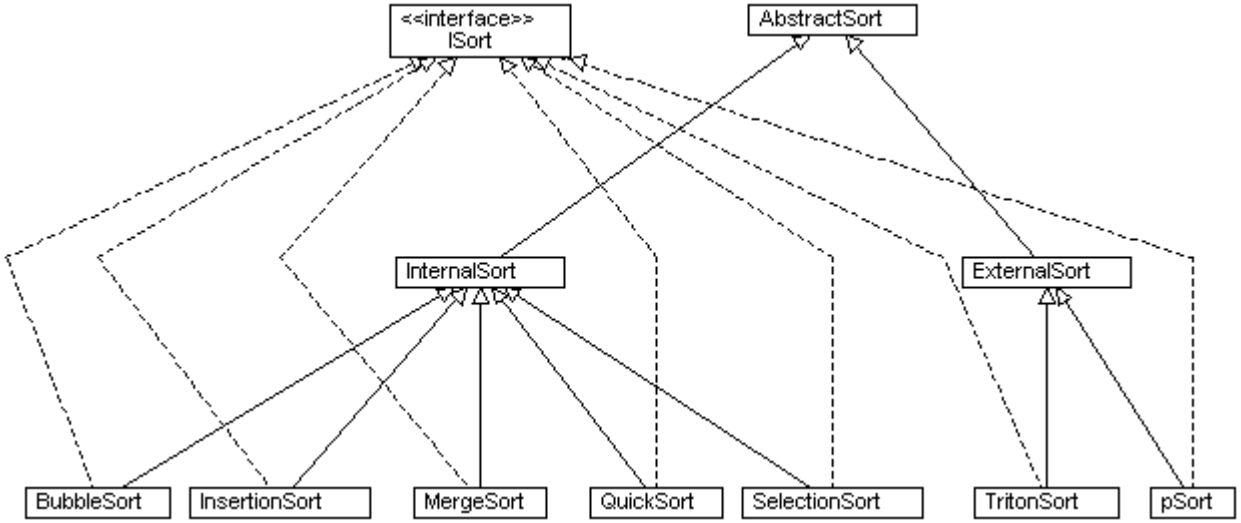
```
package ca.polymtl.gigl.log6306.sort;

public interface ISortExtension {
    <E extends Comparable<E>> void setExtendedSort(final ISort<E> anExtendedSort);
}
```

Dependency injection through setter method

```
package ca.polymtl.gigl.log6306.sort

public interface ISort<E extends Com
    List<E> sort(final List<E> aList
    void addObserver(final ISortObse
    void addExtension(
        final String anExtensionName,
        final Class<? extends ISortExtension> anExtensionClass);
    ISortExtension getExtension(final String anExtensionName);
    void removeExtension(final String anExtensionName);
}
```

Extensions management

Implementation



Implementation

Why no `final`?

```
package ca.polymtl.gigl.log6306.sort.impl;

abstract class AbstractSort<E extends Comparable<E>> {
    private final Map<String, ISortExtension> mapOfExtensionInstances;

    public void addExtension(
        final String anExtensionName,
        final Class<? extends ISortExtension> anExtensionClass) {

        final ISortExtension extension = anExtensionClass.newInstance();
        final Method dependencyInjector =
            anExtensionClass.getMethod("setExtendedSort", ISort.class);
        dependencyInjector.invoke(extension, this);
        this.mapOfExtensionInstances.put(anExtensionName, extension);
    }
    public final ISortExtension getExtension(final String anExtensionName) {
        return this.mapOfExtensionInstances.get(anExtensionName);
    }
    public final void removeExtension(final String anExtensionName) {
        this.mapOfExtensionInstances.remove(anExtensionName);
    }

    // ...
}
```

Dependency injection through setter method

Implementation

Propagate extensions to composed sorts

- Composite design pattern

```
package ca.polymtl.gigl.log6306.sort.impl;

class TypeOfSort<E extends Comparable<E>> extends AbstractSort<E> implements ITypeOfSort<E> {

    @Override
    public final void addExtension(
        final String anExtensionName,
        final Class<? extends ISortExtension> anExtensionClass) {

        final Iterator<ISort<E>> iterator = this.listOfSortAlgorithms.iterator();
        while (iterator.hasNext()) {
            final ISort<E> sortAlgorithm = (ISort<E>) iterator.next();
            sortAlgorithm.addExtension(anExtensionName, anExtensionClass);
        }
    }

    // ...
}
```

Implementation

Propagate extensions to decorated sort

- Composite design pattern

```
package ca.polymtl.gigl.log6306.sort.impl;

public abstract class SortDecorator<E extends Comparable<E>>
    extends AbstractSort<E> implements ISort<E> {

    @Override
    public final void addExtension(
        final String anExtensionName,
        final Class<? extends ISortExtension> anExtensionClass) {

        this.decoratedSortAlgorithm.addExtension(anExtensionName, anExtensionClass);
    }

    // ...
}
```

Usage

- The Client can (add and) use extensions

```
package ca.polymtl.gigl.log6306;

public class Client {
    public static void main(final String[] args) {
        final List<String> l = Arrays.asList(new String[] { "Rick Deckard", "Roy Batty",
            "Harry Bryant", "Hannibal Chew", "Gaff", "Holden", "Leon Kowalski", "Taffey
            Lewis", "Pris", "Rachael", "J.F. Sebastian", "Dr. Eldon Tyrell", "Zhora",
            "Hodge", "Mary" });

        final ITypeOfSort<String> t2 = Factory.getInstance().getInternalSortAlgorithms();
        t2.addExtension("Statistics", CountingExtension.class);
        t2.sort(l);
        final ISortIterator<String> iterator = t2.getSortAlgorithms();
        while (iterator.hasNext()) {
            final ISort<String> sort = iterator.getNext();
            final CountingExtension countingExtension = sort.getExtension("Statistics");
            System.out.println(countingExtension.getCounts());
        }
    }
}
```

Usage

- The Client can (add and) use extensions

```
package ca.polymtl.gigl.log6306;

public class Client {
    public static void main(final String[] args) {
        final List<String> l = Arrays.asList(new String[] { "Rick Deckard", "Roy Batty",
            "Harry Bryant", "Hannibal Chew", "Gaff", "Holden", "Leon Kowalski", "Taffey
            Lewis", "Pris", "Rachael", "J.F. Sebastian", "Dr. Eldon Tyrell", "Zhora",
            "Hodge", "Mary" });

        final ITypeOfSort<String> t2 = Factory.getInstance().getInternalSortAlgorithms();
        t2.addExtension("Statistics", CountingExtension.class);
        t2.sort(l);
        final ISortIterator<String> iterator = t2.getSortAlgorithms();
        while (iterator.hasNext()) {
            final ISort<String> sort = iterator.getNext();
            final CountingExtension countingExtension = sort.getExtension("Statistics");
            System.out.println(countingExtension.getCounts());
        }
    }
}
```

Usage

- The Client can (add and) use extensions

```
TypeOfSort
[Dr. Eldon Tyrell, Gaff, Hannibal Chew, Harry Bryant, Hodge, Holden, J.F. Sebastian,
Leon Kowalski, Mary, Pris, Rachael, Rick Deckard, Roy Batty, Taffey Lewis, Zhora]
BubbleSort    Comparisons: 196
              Swaps      : 51
InsertionSort Comparisons: 61
              Swaps      : 51
MergeSort     Comparisons: 42
              Swaps      : 59
QuickSort     Comparisons: 78
              Swaps      : 13
```

Usage

Not necessarily up to the client

- The Client can (add and) use extensions

```
package ca.polymtl.gigl.log6306;

public class Client {
    public static void main(final String[] args) {
        final List<String> l = Arrays.asList(new String[] { "Rick Deckard", "Roy Batty",
            "Harry Bryant", "Hannibal Chew", "Gaff", "Holden", "Leon Kowalski", "Taffey
            Lewis", "Pris", "Rachael", "J.F. Sebastian", "Dr. Eldon Tyrell", "Zhora",
            "Hodge", "Mary" });

        final ITypeOfSort<String> t2 = Factory.getInstance().getInternalSortAlgorithms();
        t2.addExtension("Statistics", CountingExtension.class);
        t2.sort(l);
        final ISortIterator<String> iterator = t2.getSortAlgorithms();
        while (iterator.hasNext()) {
            final ISort<String> sort = iterator.getNext();
            final CountingExtension countingExtension = sort.getExtension("Statistics");
            System.out.println(countingExtension.getCounts());
        }
    }
}
```

Usage

- The Client can (add and) use extensions

```
package ca.polymtl.gigl.log6306;

public class CountingExtension implements ISortExtension {
    private ISort extendedSort;
    private CountingObserver<?> countingObserver;

    @Override
    public <E extends Comparable<E>> void setExtendedSort(final ISort<E> anExtendedSort) {
        this.extendedSort = anExtendedSort;
        this.countingObserver = new CountingObserver<E>();
        this.extendedSort.addObserver(this.countingObserver);
    }
    public String getCounts() {
        final StringBuilder builder = new StringBuilder();
        builder.append(this.extendedSort.getClass().getSimpleName());
        builder.append("\tComparisons: ");
        builder.append(this.countingObserver.getNumberOfComparisons());
        builder.append("\n\t\tSwaps      : ");
        builder.append(this.countingObserver.getNumberOfSwaps());
        return builder.toString();
    }
}
```

Usage

- The Client can (add and) use extensions

```
package ca.polymtl.gigl.log6306;

public class CountingObserver<E extends Comparable<E>> implements ISortObserver<E> {
    private int numberOfComparisons;
    private int numberOfSwaps;

    @Override
    public void valuesCompared(final ComparisonEvent<E> comparisonEvent) {
        this.numberOfComparisons++;
    }
    @Override
    public void valuesSwapped(final SwapEvent<E> swapEvent) {
        this.numberOfSwaps++;
    }
    public int getNumberOfComparisons() {
        return this.numberOfComparisons;
    }
    public int getNumberOfSwaps() {
        return this.numberOfSwaps;
    }
}
```

Usage

- The Client can (add and)

Attaching extensions through the Composite

Asking each leaf for its counts

```
package ca.polymtl.gigl.log6306;

public class Client {
    public static void main(final String[] args) {
        final List<String> l = Arrays.asList(new String[] { "Rick Deckard", "Roy Batty",
            "Harry Bryant", "Hannibal Chew", "Gaff", "Holden", "Leon Kowalski", "Taffey
            Lewis", "Pris", "Rachael", "J.F. Sebastian", "Dr. Eldon Tyrell", "Zhora",
            "Hodge", "Mary" });

        final ITypeOfSort<String> t2 = Factory.getInstance().getInternalSortAlgorithms();
        t2.addExtension("Statistics", CountingExtension.class);
        t2.sort(l);
        final ISortIterator<String> iterator = t2.getSortAlgorithms();
        while (iterator.hasNext()) {
            final ISort<String> sort = iterator.getNext();
            final CountingExtension countingExtension = sort.getExtension("Statistics");
            System.out.println(countingExtension.getCounts());
        }
    }
}
```

Usage

- The Client can (add and) use extensions

```
TypeOfSort
[Dr. Eldon Tyrell, Gaff, Hannibal Chew, Harry Bryant, Hodge, Holden, J.F. Sebastian,
Leon Kowalski, Mary, Pris, Rachael, Rick Deckard, Roy Batty, Taffey Lewis, Zhora]
BubbleSort    Comparisons: 196
              Swaps      : 51
InsertionSort Comparisons: 61
              Swaps      : 51
MergeSort     Comparisons: 42
              Swaps      : 59
QuickSort     Comparisons: 78
              Swaps      : 13
```

Concerns

- Internal vs. External extensions
- Identifying extensions
- On-demand loading of extensions
- Freeing subjects and extensions

Concerns

- Internal vs. External extensions
  - In our example, the client add the extension
  - Extensions can be internal to the subject
- Identifying extensions
- On-demand loading of extensions
- Freeing subjects and extensions

Concerns

- Internal vs. External extensions
- Identifying extensions
  - In our example, a `String` to identify extensions
  - Risk of name clash; possibly, use reflection
- On-demand loading of extensions
- Freeing subjects and extensions

Concerns

- Internal vs. External extensions
- Identifying extensions
- On-demand loading of extensions
  - In our example, the extension must be instantiated upon addition to the subject
  - Extension could be instantiated on-demand
    - Singleton design pattern
- Freeing subjects and extensions

Concerns

- Internal vs. External extensions
- Identifying extensions
- On-demand loading of extensions
- Freeing subjects and extensions
  - In our example, the subject is injected into the extension object
  - Extension object could be considered internal to the subject to free extension and subjects

Conclusion

- The Extension Object design pattern is related to the Decorator design pattern
  - Decorators are “around” the decorated objects
    - **Decorate composites and leafs similarly**
  - Extensions are “inside” the extended objects
    - **Different extensions for composites and leafs**

Conclusion

- The Extension Object design pattern allows
  - Extended objects to keep their identities
  - Extended objects to control their extensions
    - Instantiations
    - Garbage collections
  - Extended objects keep their (minimal) interface clean and minimal
  - Clients must know about the possible extensions