Basic design patterns

Noël PLOUZEAU

IRISA/ISTIC

Concept of design pattern

- We have already seen a form of design patterns
  - canned wisdom
  - harvested from designs and code made by thousands of developers in the last 30 years

The real design patterns

- Traditionally the term is used for
  - a reusable solution
  - to a frequently occurring problem
  - in a given context

Origin

- Proposed by an architect (Ch. Alexander) around 1979
- Applied to software design starting from 1987
- Gained popularity with this book:
  - Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. ISBN 0-201-63361-2.

A form of template

- Main features to describe a DP
  - name, intent (what is the goal), motivation (problem), applicability (where)
  - participants, collaboration, structure
  - implementation example
  - consequences (side effects, dependencies)

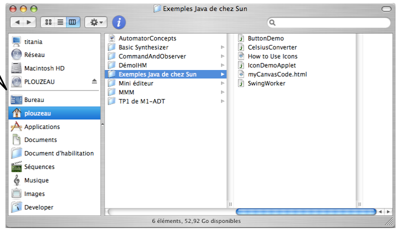
A first example: Observer

- Motivation
  - some objects must keep their state consistent with the state of other objects (the subjects)
  - when the subjects change, the dependent object must update their state

Example

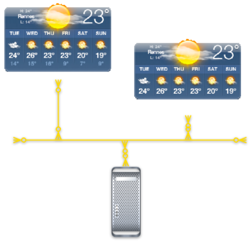
View

File system



Files, directories and views must be kept in sync

Another example



Data must be sent
to the clients
when it changes

General classes of solution

- Data source, data sink
- Who has the initiative?
  - Data source -> “push”
  - Data sink -> “pull”
  - Another object -> “pull/push”
- Observer is based on “push”

The Observer PC

- Intent: to propagate data changes of an object to other objects
- Motivation: some objects must keep their state in sync with others’ states
- Participants: mutator, subject, observer, concrete subject, concrete observer

The CRC template

- To describe a pattern one can use the CRC (class, responsibilities, collaborations) template
  - class: these are the types (interfaces, implementation classes)
  - responsibilities: what each type must ensure
  - collaborations: an implementation to guarantee that the responsibilities are fullfilled

Classes (types)

- mutator, subject and observer define a protocol
  - rules of interaction
  - interfaces between participants
  - one participant often plays the role of “the outside of the PC”

Details of responsibilities

- subject
  - manages observers’ subscription (interface plus storage of subscriptions)
- observer
  - provides an interface to receive update notification from observers

Details of responsibilities (cont’d)

- mutator
  - the outside, the reason why subject states change
- mutator, subject and observer are declarations, not implementations (therefore Java/UML interfaces)

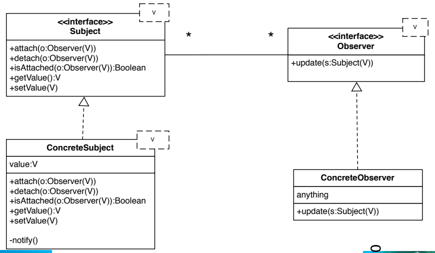
Details of responsibilities (cont’d)

- concrete subject
  - stores a data state, provides R/U operations for it
  - must notify observers when its state changes

Details of responsibilities (cont'd)

- concrete observer
  - has a state that depends on the subject's state
  - provides a method for the operation that the subject can call to notify changes

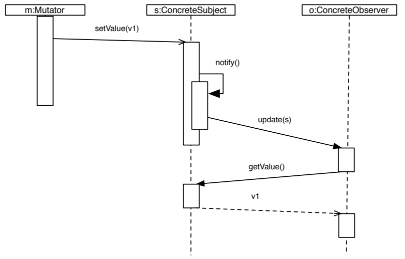
Structure



Collaborations

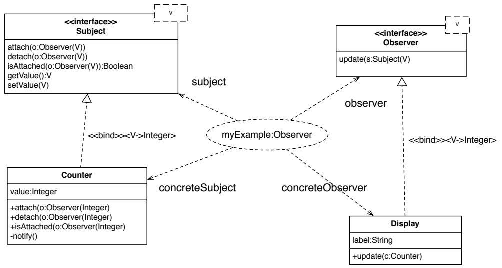
- There are many ways to describe them
  - informal description using text
  - UML sequence diagrams (including HMSC constructs)
  - UML collaboration diagrams
  - statecharts
  - …

Observer collaboration



Note that only one observer is represented he

Example of instantiation



Implementation possibilites

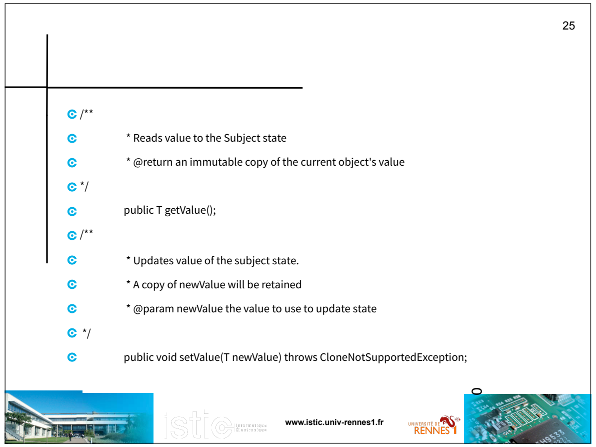
- As Observer is often used one could try to build a reusable implementation
  - Oracle has one but it is awkward
  - Factorize the notification code into a stateless concrete subject
  - Use genericity

An example of implementation

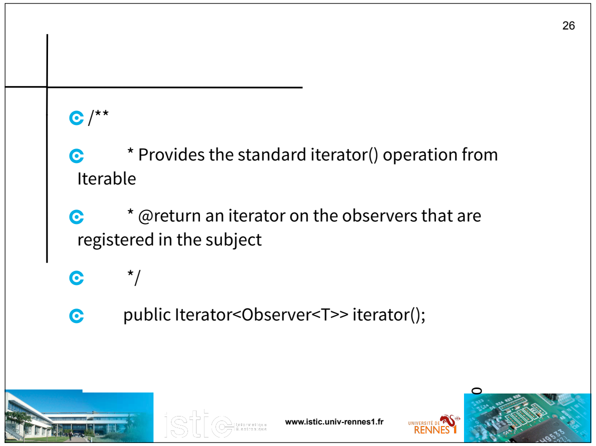
- We use genericity for the value type
  - Java implementation of the UML structure presented earlier
- We also apply the “return copies of state” for the value of the concrete subject

- In file Subject.java
- package fr.istic.nplouzeau.observer;
- import java.util.Iterator;
- /**
- * Author: PLOUZEAU, Noël
- * Date: 2013-07-31
- * Time: 08:52
- */

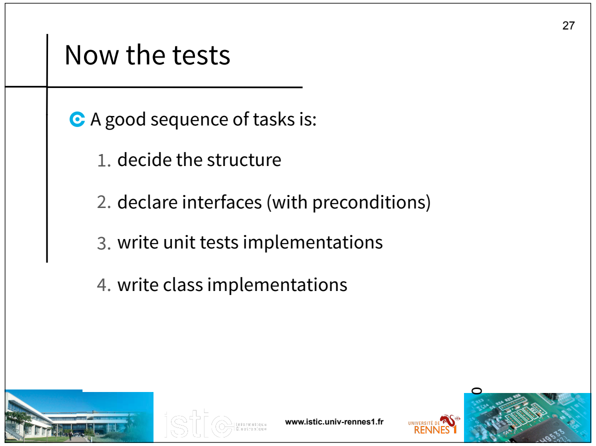
- public interface Subject<T extends ValueType> extends Iterable<Observer<T>> {
-         /**
-          * Adds o to the set of observers that will be notified when
-          * the value of this changes.
-          * pre : o is not already attached
-          * @param o   the observer to attach
-          * @throws IllegalArgumentException when o is already attached
-          */
-         public void attach(Observer<T> o); // **No need for throws here**



```
/**
         * Reads value to the Subject state
         * @return an immutable copy of the current object's value
*/
        public T getValue();
/**
         * Updates value of the subject state.
         * A copy of newValue will be retained
         * @param newValue the value to use to update state
*/
        public void setValue(T newValue) throws CloneNotSupportedException;
```

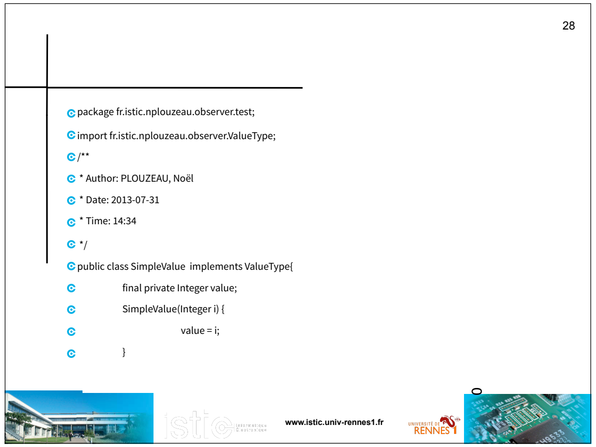


```
/**
         * Provides the standard iterator() operation from Iterable
         * @return an iterator on the observers that are registered in the subject
         */
        public Iterator<Observer<T>> iterator();
```

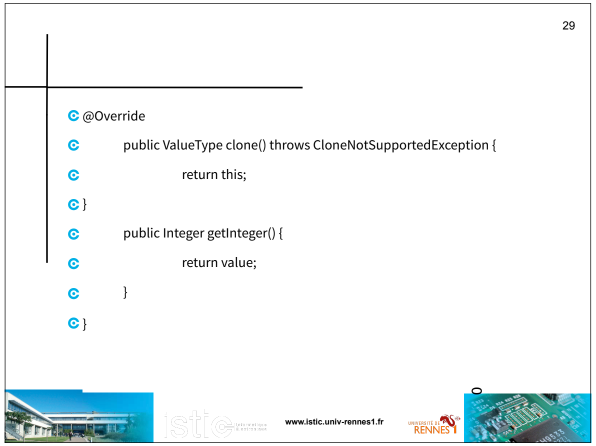


Now the tests

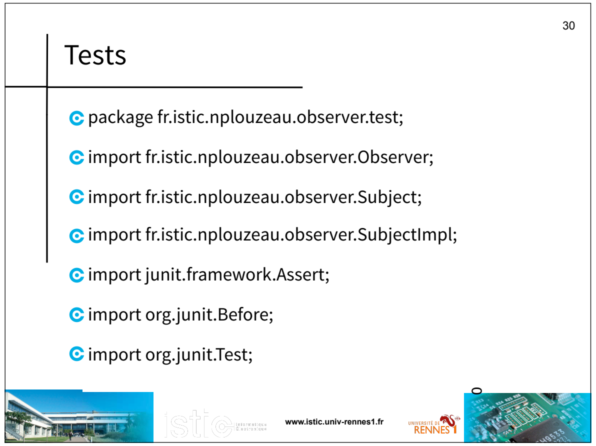
- A good sequence of tasks is:
  1. decide the structure
  2. declare interfaces (with preconditions)
  3. write unit tests implementations
  4. write class implementations



```
package fr.istic.nplouzeau.observer.test;
import fr.istic.nplouzeau.observer.ValueType;
/**
 * Author: PLOUZEAU, Noël
 * Date: 2013-07-31
 * Time: 14:34
 */
public class SimpleValue  implements ValueType{
        final private Integer value;
        SimpleValue(Integer i) {
                value = i;
        }
```

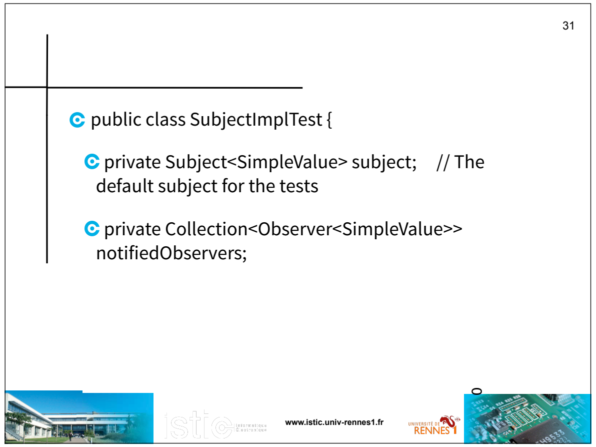


```
@Override
        public ValueType clone() throws CloneNotSupportedException {
                return this;
}
        public Integer getInteger() {
                return value;
        }
}
```

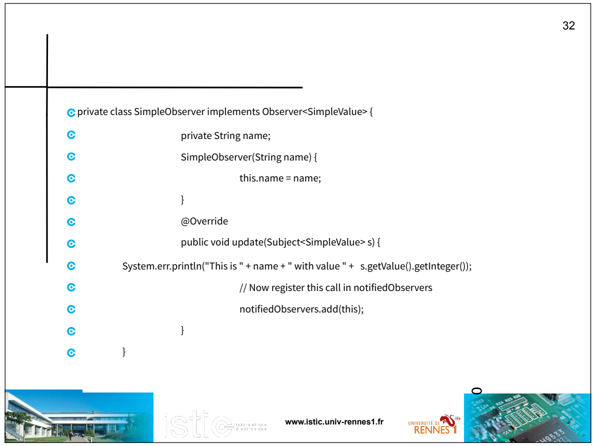


Tests

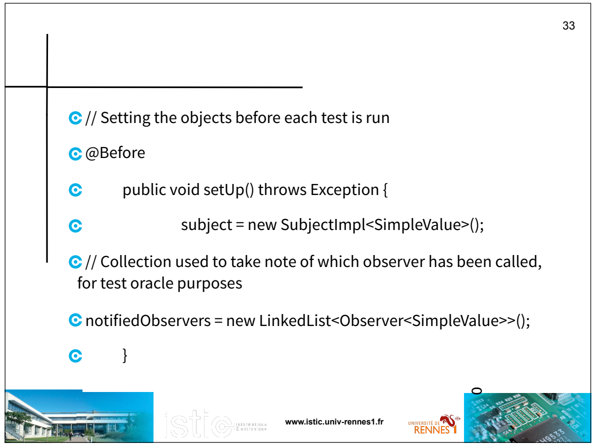
```
package fr.istic.nplouzeau.observer.test;
import fr.istic.nplouzeau.observer.Observer;
import fr.istic.nplouzeau.observer.Subject;
import fr.istic.nplouzeau.observer.SubjectImpl;
import junit.framework.Assert;
import org.junit.Before;
import org.junit.Test;
```



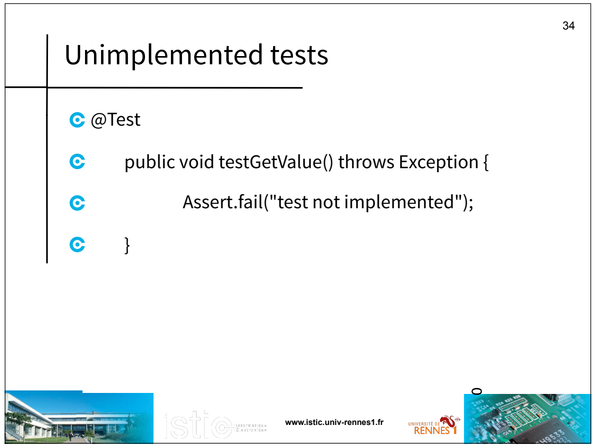
- public class SubjectImplTest {
  - private Subject<SimpleValue> subject;   // The default subject for the tests
  - private Collection<Observer<SimpleValue>> notifiedObservers;



```
private class SimpleObserver implements Observer<SimpleValue> {
                private String name;
                SimpleObserver(String name) {
                        this.name = name;
                }
                @Override
                public void update(Subject<SimpleValue> s) {
        System.err.println("This is " + name + " with value " +  s.getValue().getInteger());
                        // Now register this call in notifiedObservers
                        notifiedObservers.add(this);
                }
        }
```

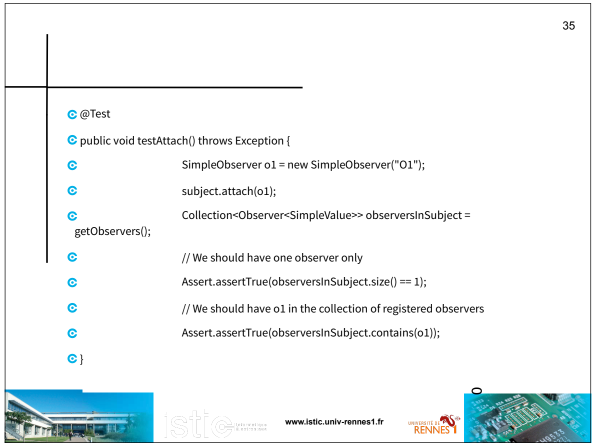


```
// Setting the objects before each test is run
@Before
        public void setUp() throws Exception {
                subject = new SubjectImpl<SimpleValue>();
// Collection used to take note of which observer has been called, for test oracle purposes
notifiedObservers = new LinkedList<Observer<SimpleValue>>();
        }
```

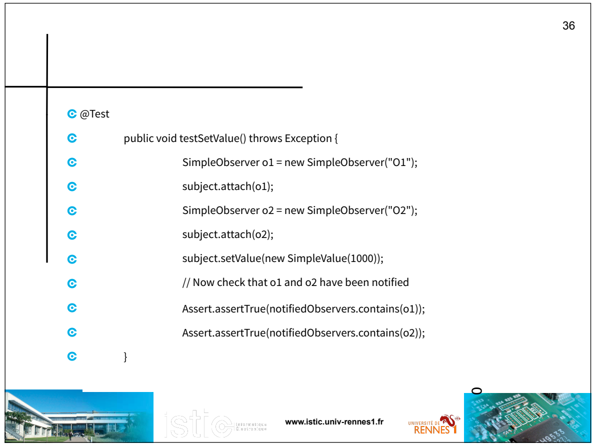


Unimplemented tests

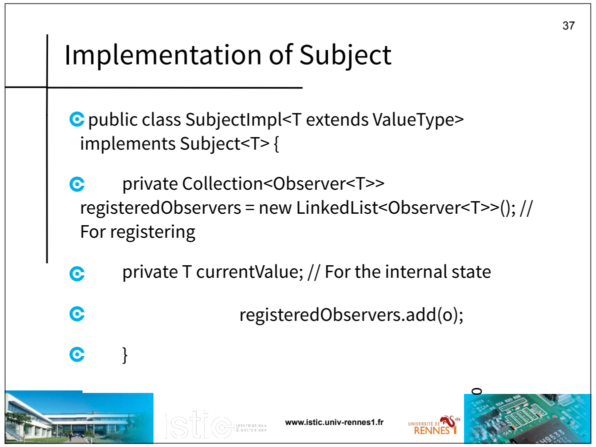
```
@Test
      public void testGetValue() throws Exception {
                  Assert.fail("test not implemented");
      }
```



```
@Test
public void testAttach() throws Exception {
                        SimpleObserver o1 = new SimpleObserver("O1");
                        subject.attach(o1);
                        Collection<Observer<SimpleValue>> observersInSubject =
  getObservers();
                        // We should have one observer only
                        Assert.assertTrue(observersInSubject.size() == 1);
                        // We should have o1 in the collection of registered observers
                        Assert.assertTrue(observersInSubject.contains(o1));
}
```

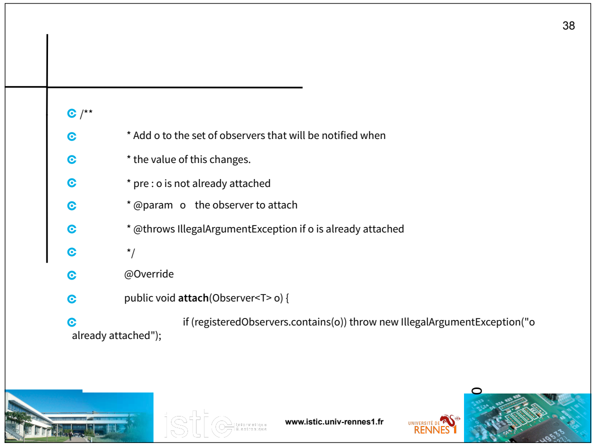


```
@Test
            public void testSetValue() throws Exception {
                        SimpleObserver o1 = new SimpleObserver("O1");
                        subject.attach(o1);
                        SimpleObserver o2 = new SimpleObserver("O2");
                        subject.attach(o2);
                        subject.setValue(new SimpleValue(1000));
                        // Now check that o1 and o2 have been notified
                        Assert.assertTrue(notifiedObservers.contains(o1));
                        Assert.assertTrue(notifiedObservers.contains(o2));
            }
```

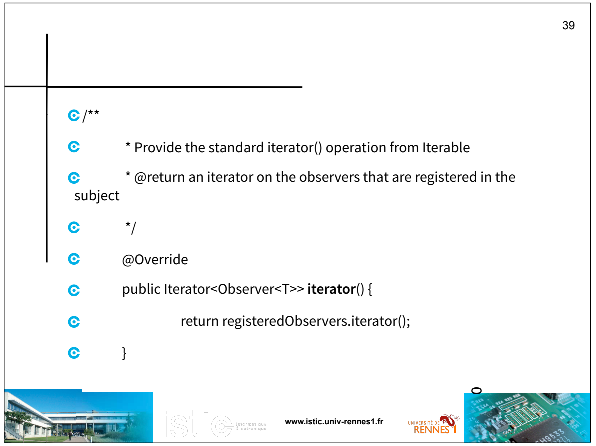


Implementation of Subject

```
public class SubjectImpl<T extends ValueType>
implements Subject<T> {
        private Collection<Observer<T>>
registeredObservers = new LinkedList<Observer<T>>(); //
For registering
        private T currentValue; // For the internal state
                              registeredObservers.add(o);
        }
```



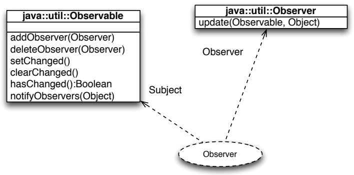
```
/**
         * Add o to the set of observers that will be notified when
         * the value of this changes.
         * pre : o is not already attached
         * @param   o   the observer to attach
         * @throws IllegalArgumentException if o is already attached
         */
        @Override
        public void attach(Observer<T> o) {
                        if (registeredObservers.contains(o)) throw new IllegalArgumentException("o
already attached");
```



```
/**
         * Provide the standard iterator() operation from Iterable
         * @return an iterator on the observers that are registered in the
subject
         */
        @Override
        public Iterator<Observer<T>> iterator() {
                return registeredObservers.iterator();
        }
```

```
/**
     * Write accessor to the subject state.
     * A copy of newValue will be retained in this
     * @param newValue
     */
    @Override
    public void setValue(T newValue) throws CloneNotSupportedException {
            currentValue = (T) newValue.clone();
            notifyObservers();
    }
```

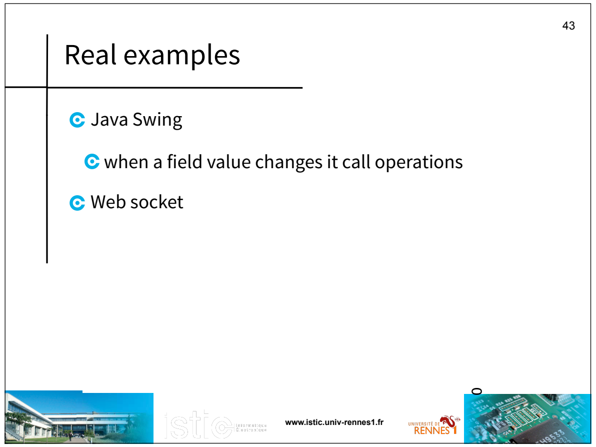
About the Observer implementation in the standard library



This is what is defined in the Java standard library

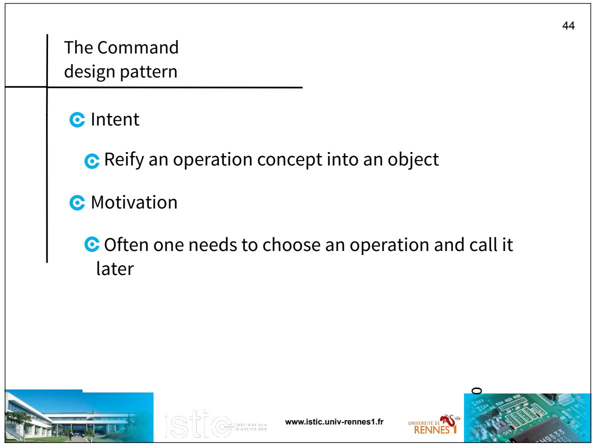
Good and bad choices

- Observer is an interface: good
- Observable (= Subject) is a class: not so good, as method inheritance is consumed
  - a concrete subject cannot inherit methods from problem-related classes
  - implementation is mentioned in parameters and attributes (see rule #EJ52)



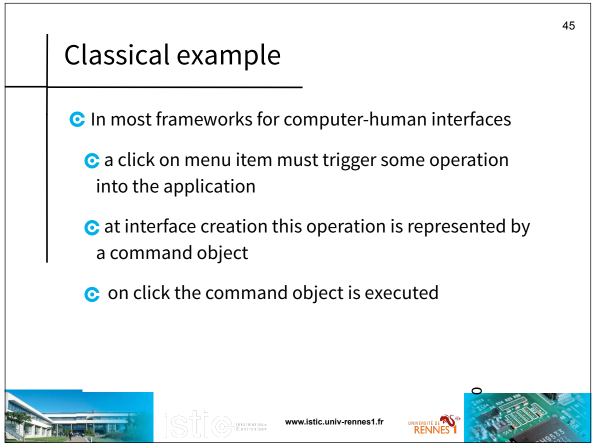
Real examples

- Java Swing
  - when a field value changes it call operations
- Web socket



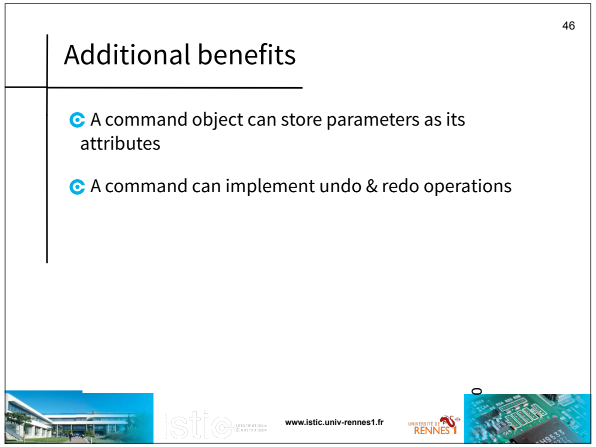
The Command design pattern

- Intent
  - Reify an operation concept into an object
- Motivation
  - Often one needs to choose an operation and call it later



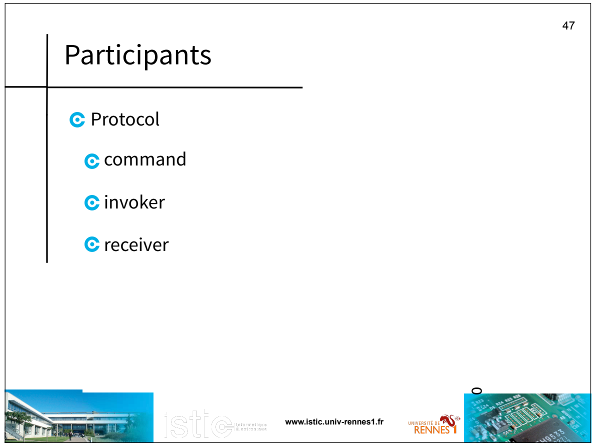
Classical example

- In most frameworks for computer-human interfaces
  - a click on menu item must trigger some operation into the application
  - at interface creation this operation is represented by a command object
  - on click the command object is executed



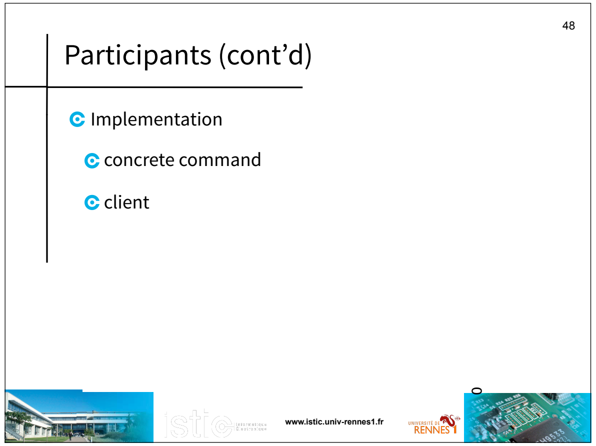
Additional benefits

- A command object can store parameters as its attributes
- A command can implement undo & redo operations



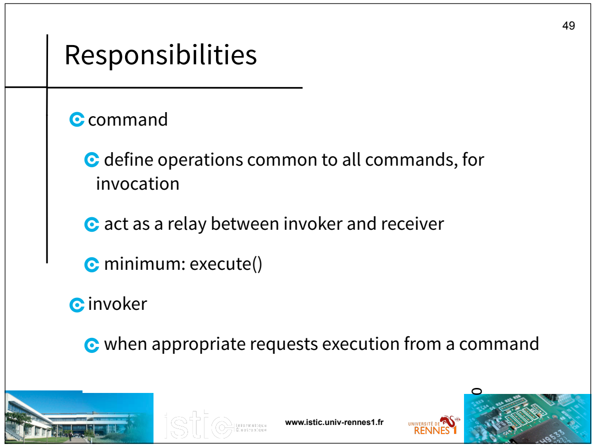
Participants

- Protocol
  - command
  - invoker
  - receiver



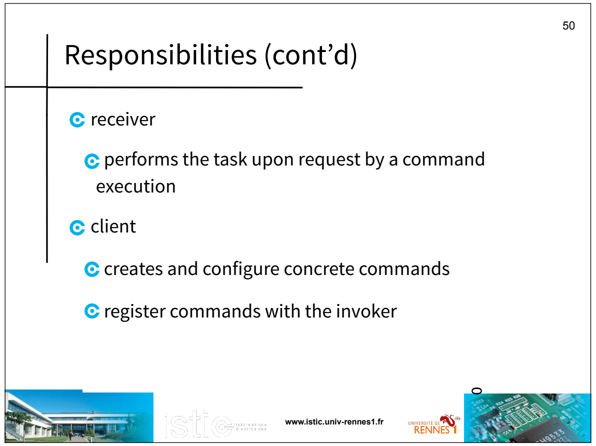
Participants (cont’d)

- Implementation
  - concrete command
  - client



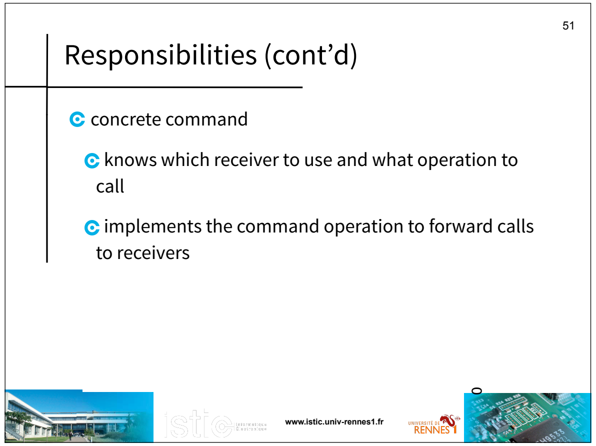
Responsibilities

- command
  - define operations common to all commands, for invocation
  - act as a relay between invoker and receiver
  - minimum: execute()
- invoker
  - when appropriate requests execution from a command



Responsibilities (cont’d)

- receiver
  - performs the task upon request by a command execution
- client
  - creates and configure concrete commands
  - register commands with the invoker



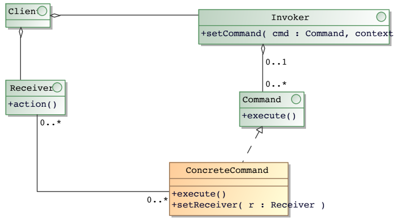
Responsibilities (cont’d)

- concrete command
  - knows which receiver to use and what operation to call
  - implements the command operation to forward calls to receivers

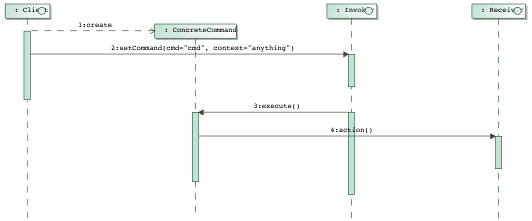
Collaborations

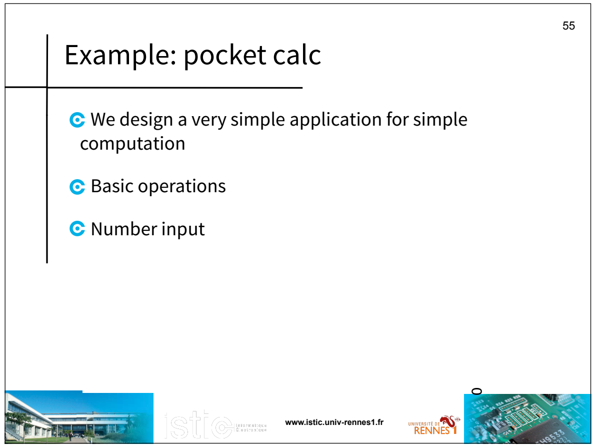
- the client creates command and set up the invoker
- when the time comes
  - the invoker detects this
  - retrieves the command set up by the client
  - call the execute() operation of this command
- the execute() method calls a given operation on the receiver
- the receiver executes the given operation

Structure



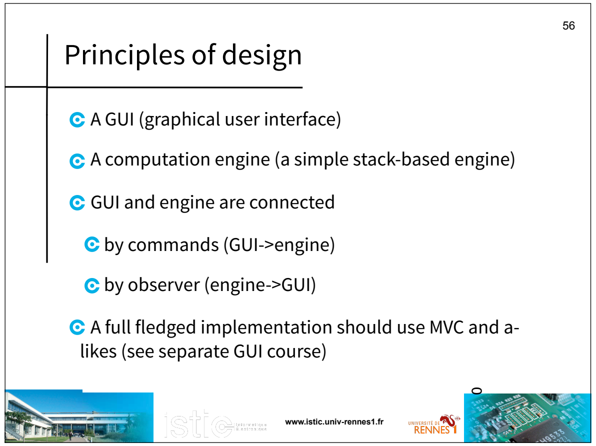
Collaboration as a sequence diagram





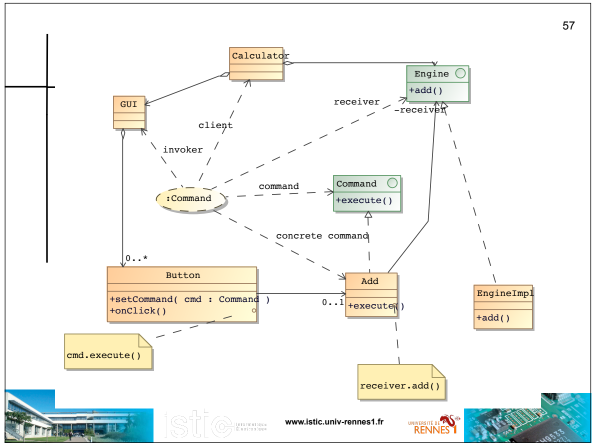
Example: pocket calc

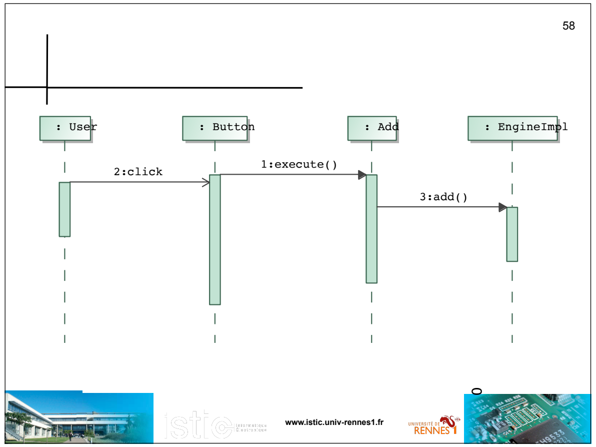
- We design a very simple application for simple computation
- Basic operations
- Number input

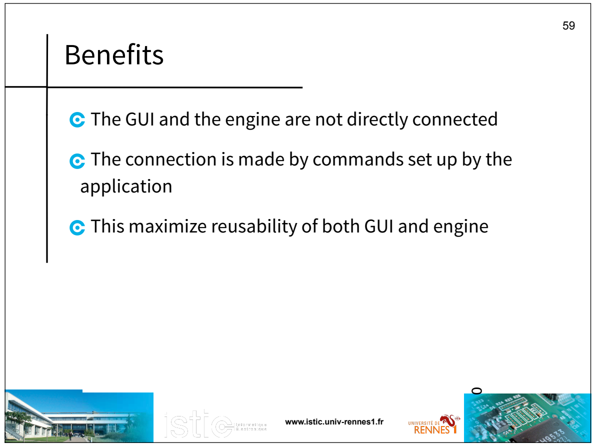


Principles of design

- A GUI (graphical user interface)
- A computation engine (a simple stack-based engine)
- GUI and engine are connected
  - by commands (GUI->engine)
  - by observer (engine->GUI)
- A full fledged implementation should use MVC and a-likes (see separate GUI course)







Benefits

- The GUI and the engine are not directly connected
- The connection is made by commands set up by the application
- This maximize reusability of both GUI and engine