IN1010 Uke 9, vår 2022
Programmeringsmønstre
Design Patterns

Eric Jul
Programmeringsteknologi gruppe
Inst. for informatikk
UiO

Who am I?

- **Professor** & Head of Programming Technology Group, **IfI**
- Masters degree 1980 University of Copenhagen in Computer Science & Math
- Ph.D. CS, **University of Washington**, 1988
- 1981-2009 University of Copenhagen, professor 2000
- 2009-2015 Member **Bell Labs**, Dublin, Ireland
- 2009-2016 Professor II, IfI, UiO
- 2016- Professor, IfI, UiO
- Research in Distribution/Concurrency/Objects since 1974
- Co-Designer of the Object Oriented Language Emerald Mobility (consider taking IN5570 ;-))
- Fine-grained Object Mobility
- Self-migration of Operating Systems

Who am I?

For fun, I fly glider (*Seilfly) - sometimes even inverted ;-)*

Programmeringsmønstre
Design Patterns

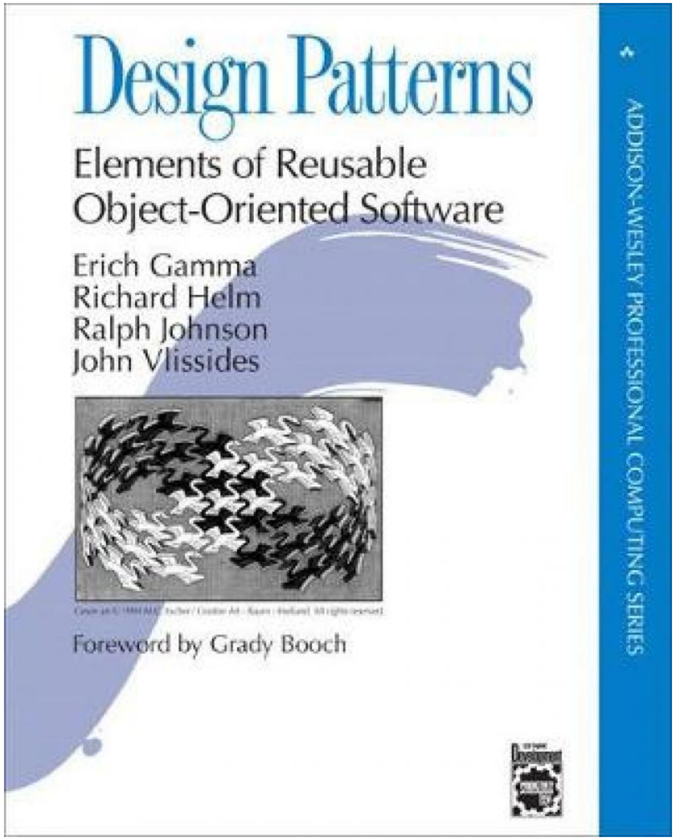
- A **design pattern** is the re-usable form of a solution to a design problem.
- An algorithm describe specific steps to achieve something: e.g., sorting.
- A design pattern is more general: prescribes a way to build something, but not the details
- Inspired by Christofer Alexander, an architect
- Example from arcitecture: A window.

[WHITEBOARD]

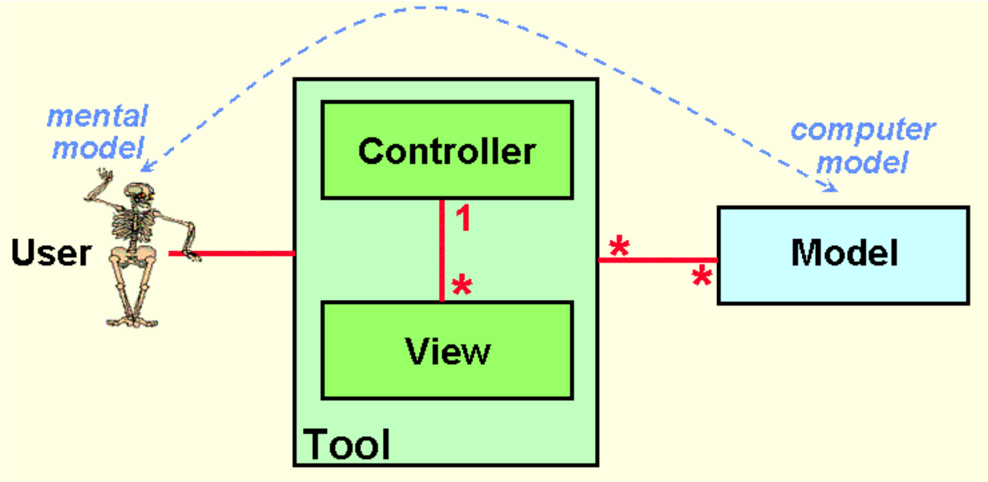
Design Patterns History

- The **design pattern** concept was clear formulated and presented in the seminal, so-called Gang of Four book in 1995
- Before that many used individual design patterns without thinking of them as one instance of a general idea

The «Bible» of Design Patterns



Modelling: Trygve Reenskaug
Professor Emeritus IfI, UiO
Inventor of Model-View-Control, 1979



Modelling

Modelling is about mapping some real-world entity into a simple version where irrelevant details have been removed and one or more essential aspects are high-lighted.

Often we go from:

real-world entity

to: *mental model*

to: `UML model`

to: **`Java program {}`**

Example: Modelling A Steam Engine

- Think of a steam engine
- Want to model it: just the temperature
- Build a SIMPLE model of the temperature
- Want to write a program to
  - Model the temperature: keep track of updates
  - Display the state of the model, *i.e.*, the current temperature

[WHITEBOARD]

Split Model and the Printing

- Separate the code for **maintaining** the model from the code that does the printing.
- *Separation of Concerns principle*

Two parts:

- Model part – represents the model itself
- View part – in charge of «viewing», *i.e.*, the printing

[WHITEBOARD]

A look at the code for our Steam Engine Example

Steam Engine

```
package steamEngineObserverPackage;
public class SteamEngineC extends SimpleObservableC {
        int temp;

        public SteamEngineC() {
                temp = 20; // arbitrary value (!)
        }

        public void setTemp(int t) {
                temp = t;
                System.out.println("New temperature " + temp);
        }

        public int getTemp() {
                return temp;
        }
}
```

Steam Engine Revisited

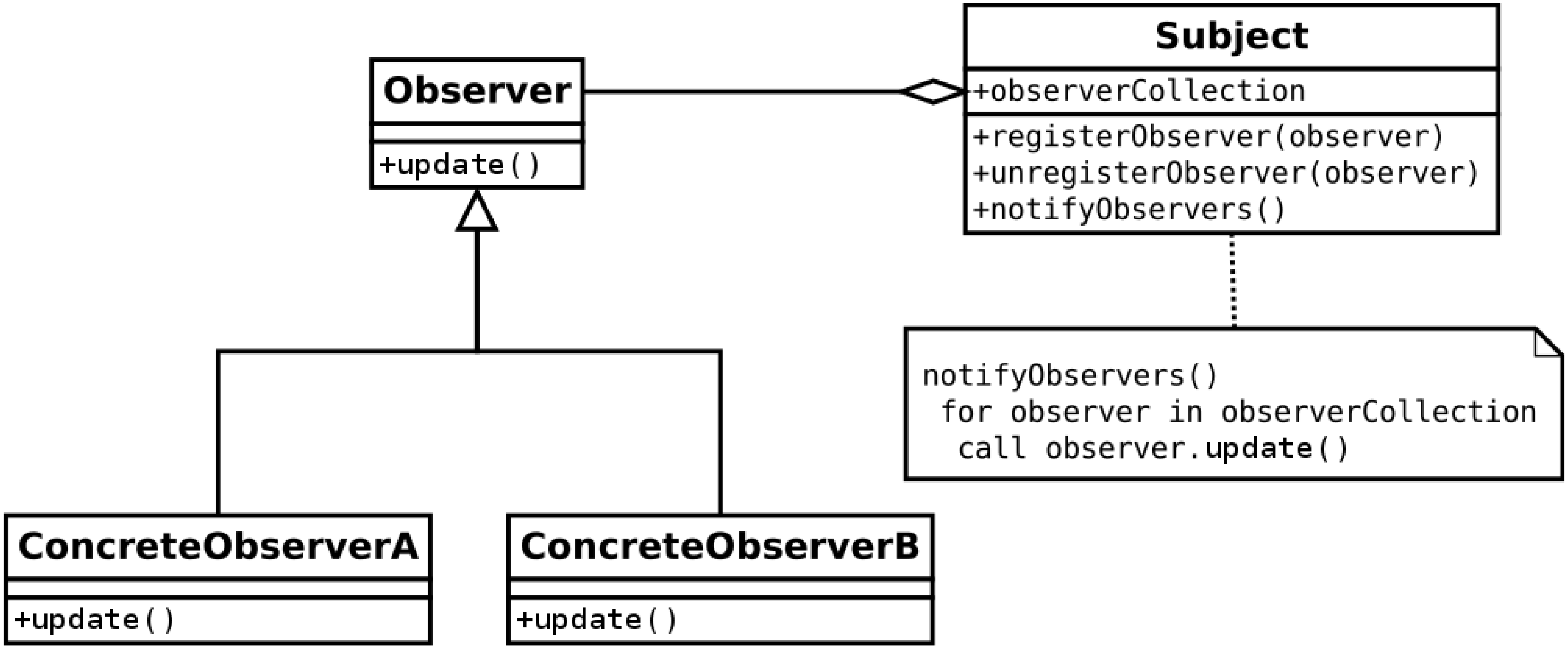
```
package steamEngineObserverPackage;
public class SteamEngineC extends SimpleObservableC {
	int temp;

	public SteamEngineC() {
		temp = 20; // arbitrary value (!)
	}

	public void setTemp(int t) {
		temp = t;
		notifyAllObs();
		% System.out.println("New temperature " + temp);
	}

	public int getTemp() {
		return temp;
	}
}
```

Observer Pattern UML (from GangOf4)



Steam Engine Observable Interface

```
public interface SimpleObservableI {
        public void add(SimpleObserverI o);
        public void notifyAllObs();
}
```

Steam Engine Simple Observable Superclass

```
public class SimpleObservableC implements SimpleObservableI {
        Set<SimpleObserverI> obsSet = new HashSet<SimpleObserverI>();

        @Override
        public void add(SimpleObserverI o) {
                obsSet.add(o);
                o.update();  // as to display current value
        }

        @Override
        public void notifyAllObs() {
                Iterator<SimpleObserverI> i = obsSet.iterator();
                while (i.hasNext()) {
                        SimpleObserverI o = i.next();
                        o.update();
                }

        }
}
```

Steam Engine Simple Observer Interface

```
package steamEngineObserverPackage;

public interface SimpleObserverI {
        public void update();
}
```

Steam Engine Print Observer

```
public class SteamEngineObserverC implements SimpleObserverI {
        SteamEngineC mySteamEngine;

        SteamEngineObserverC(SteamEngineC myEngine) {
                mySteamEngine = myEngine;
                myEngine.add(this);
        }

        public void update() {
                System.out.println("Current Steam Temperature is " +
                        mySteamEngine.getTemp());
        }
}
```

Steam Engine Alarm Observer

```
public class SteamEngineObserverAlarm implements SimpleObserverI {
        int myAlarmTemp;

        SteamEngineC mySteamEngine;

        SteamEngineObserverAlarm(SteamEngineC myEngine, int
           initialAlarmTemp) {
                mySteamEngine = myEngine;
                myAlarmTemp = initialAlarmTemp;
                myEngine.add(this);
        }

        public void update() {
                if (mySteamEngine.getTemp() > initialAlarmTemp) {
                        System.out.println("**** ALARM ****");
                }
        }

}
```

Steam Engine Simple Test

```
public class TestSteamEngine {
        public static void main(String arg[]) {
                SimpleObserverI myObs, myObs2, myAlarmObs;
                SteamEngineC myEngine;

                myEngine = new SteamEngineC();
                myObs = new SteamEngineObserverC(myEngine);
                myAlarmObs = new SteamEngineObserverAlarm(myEngine, 85);

                for(int i = 30; i <= 100; i += 10) {
                        myEngine.setTemp(i);
                }
        }
}
```

Observer available in Java Library

Observer is used so often that it is one of the Design Patterns that is built into the Java system.

A note on Overriding and the use of @override

The annotation «@override» in Java is placed just before every method that overrides a method in a superclass.

Strictly speaking, it is not necessary, however, it is considered good practise to use it.

By doing so, we achieve:

- Better error messages, *e.g.*, if you missspell a method name, you will get an error message.
- Better documentation: you explicitly say that this is a method that overrides a virtual method in a superclass.

If interested, you can find more on this here:

https://beginnersbook.com/2014/07/override-annotation-in-java/

A Design Pattern that you have ALREADY SEEN: Iterator

(Beware: a *very* common Design Pattern developed independently by many, so method names vary.)

Example: ***we have already seen it in slide 16***

Another Design Pattern: Proxy
Example: Displaying a large image loaded from a file

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

[WHITEBOARD]

Displaying a large image loaded from a file

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

Proxy Object: loads file on demand

```
public class ProxyImage implements Image{
   private RealImage realImage;
   private String fileName;
   public ProxyImage(String fileName){
      this.fileName = fileName;
   }
   public void display(){
      if(realImage == null){
         realImage = new RealImage(fileName);
      }
   realImage.display();
   }
}
```

Proxy Patterns

In this particular case, proxy is used to implement *lazy evaluation.*

Proxy can also be used for:

- Remote method calls where the proxy encapsulates all the work as to do the remote call, *e.g.*, Java RMI code.
- Access control.

References

The original Gang of Four Design Patterns seminal book:
https://www.oreilly.com/library/view/design-patterns-elements/0201633612/

Presented at the OOPSLA conference in Portland, Oregon, USA in October 1994.

A quick start guide to Design Patterns in Java:
https://www.tutorialspoint.com/design_pattern/design_pattern_quick_guide.htm

Proxy example:
https://www.tutorialspoint.com/design_pattern/proxy_pattern.htm

Summary

In today's lecture:

- Intro to Design Patterns
- Presentation of three different design patterns:
  - Example of using Observer part of MVC: Steam Engine
  - Iterator
  - Proxy

Feedback

- Did you learn anything?
  1. Very little
  2. Some
  3. Good
  4. A lot
  5. An awful lot

- Was it hard?
  1. Too easy
  2. A little Easy
  3. Fine
  4. A little hard
  5. Too hard