

20 Design patterns

Main concepts to be covered

- Why design patterns matters
- Classification of patterns
- Some common patterns
 - Composite
 - Decorator
 - Singleton
 - Factory method
 - Observer

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 20 2

Why design patterns matters

- Inter-class relationships are important, and can be complex.
- Some relationships recur in different applications.
- Design patterns help clarify relationships, and promote reuse.
- Don't reinvent the wheel!

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 20 3

Small and large Patterns

- Architectural patterns (large scale)
- Design patterns (medium scale)
- Idioms (small scale)

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 20 4

Architectural Pattern

- Fundamental structural organization template for a whole software system.
- It provides
 - a set of predefined subsystems
 - responsibilities of subsystems
 - relationships between subsystems
- Example Model View Controller

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 20 5

Design pattern

- Medium scale organization scheme for components of a software system.
- It provides
 - a scheme for refining components and their relationships.
 - communication rules for cooperating components.
- It is independent of programming language.
- Example Observer

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 20 6

Idiom

- Low level design pattern.
- Programming language specific implementation techniques.
- Example: The string copy loop idiom for the C/C++ programming language

```
- while ((*t++ = *s++) != '\0') ;
```

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 20 7

Design pattern categories

- **Creational patterns:** Object creation.
 - Singleton, Factory, Abstract Factory, Factory Method, ...
- **Structural patterns:** Static composition.
 - Composite, Decorator, Adapter, ...
- **Behavioral patterns:** Dynamic object interaction.
 - Iterator, Command, State, Template Method, Strategy, ...

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 20 8

Design pattern description

A design pattern description consists of

- **Pattern name.**
- **Category:**
 - Creational, Structural, or behavioral.
- **Intent:**
 - The problem addressed by it.
- **Structure:**
 - Class diagram showing participants and relationships.
- **Participants:**
 - A list of participating classes or objects and their collaboration.
- **Applicability:**
 - Situations in which it is useful.
- **Its consequences:**
 - Results, trade-offs.

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 20 9

Composite

- Composite defines a part-whole relationship between objects in a tree hierarchy.
- Simple objects and composite objects can be treated **uniformly** by clients
 - uniformly with respect to their **common interface**.

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 20 10

Composite (2)

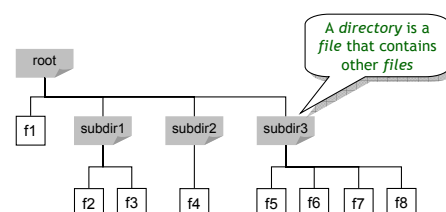
Example instances:

- Component, Container, JPanel, JButton, ...
- Hierarchical file systems have two kinds of files
 - "Ordinary" files (Leaf) contain data.
 - Directories (Composite) contain files.

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 20 11

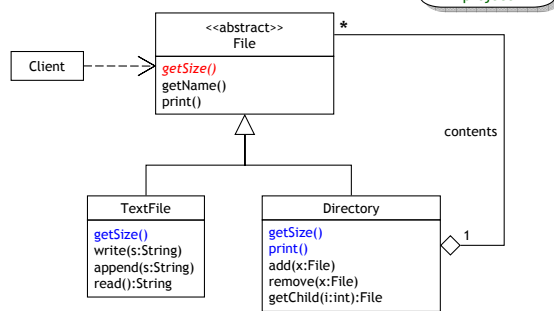
An instance of Composite: Hierarchical file systems



Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 20 12

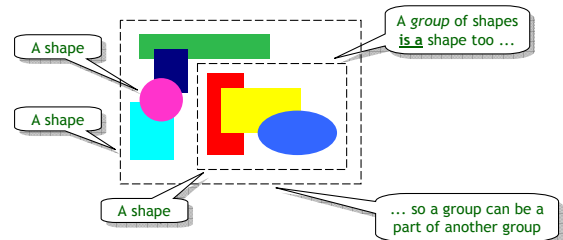
A file system



Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 20 13

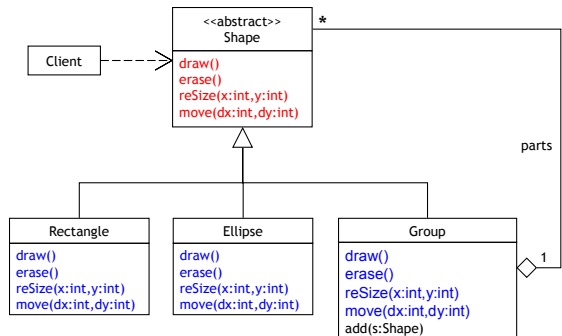
Another instance of Composite: Hierarchical geometrical shapes



Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 20 14

Geometrical shapes



Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 20 15

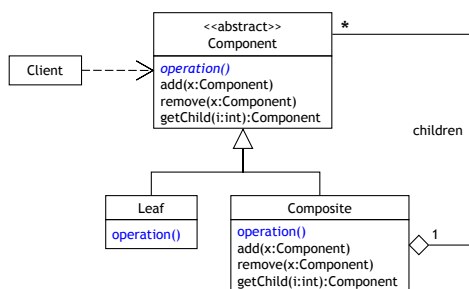
General Composite Pattern

- Participating classes:
 - Component (abstract class)
 - declares common operations and default behavior.
 - Leaf ("base case")
 - implements behavior for the primitive elements in the composition.
 - Composite ("recursion")
 - manages child components.
 - Client
 - manipulates objects in the composition via the Component interface.

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 20 16

General Composite Pattern



Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 20 17

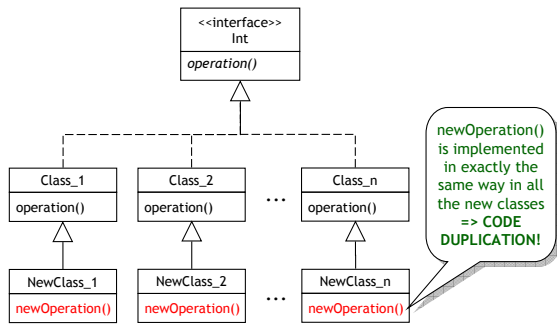
Decorator

- Augments the functionality of an **object**.
- Decorator object wraps another object.
 - The Decorator object has a similar interface.
 - Calls are relayed to the wrapped object ...
 - ... but the Decorator can interpolate additional actions.
- Example: `java.io.BufferedReader`
 - Wraps and augments an unbuffered **Reader** object.

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 20 18

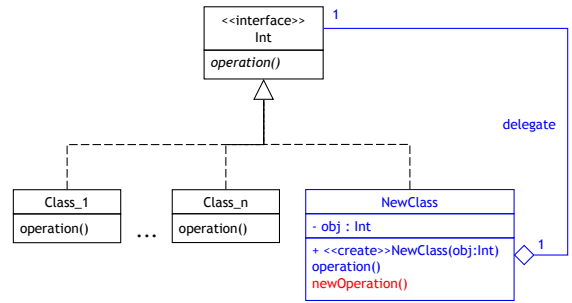
Extension by inheritance



Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 20 19

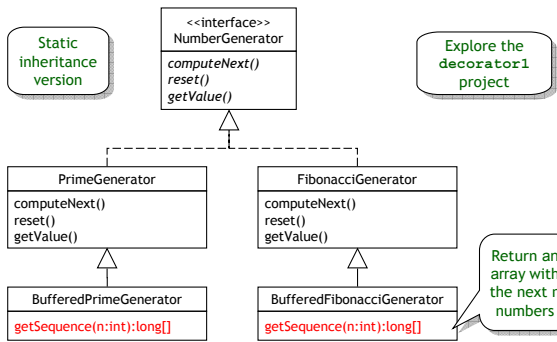
Extension using Decorator pattern



Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 20 20

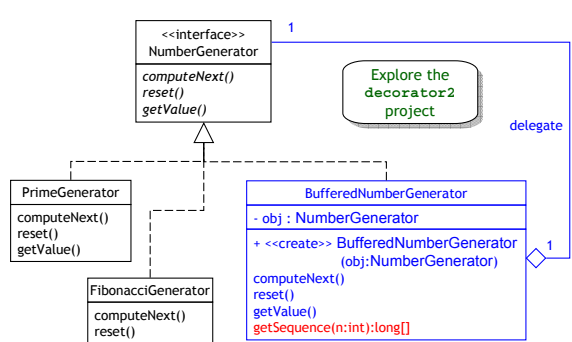
Example: Number generator



Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 20 21

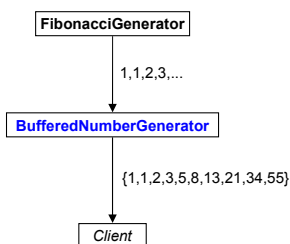
Decorated NumberGenerator



Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 20 22

Using the BufferedNumberGenerator



Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 20 23

Using the BufferedNumberGenerator

```

BufferedNumberGenerator bp =
    new BufferedNumberGenerator(new PrimeGenerator());

BufferedNumberGenerator bf =
    new BufferedNumberGenerator(new FibonacciGenerator());

...
bp.computeNext();
long x = bp.getValue();
...
bp.reset();
long[] array = bp.getSequence(10);
                {2, 3, 5, 7, 11, 13, 17, 19, 23, 29}
    
```

NumberGenerator methods work for buffered generators too

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 20 24

Java implementation

```
public class BufferedNumberGenerator implements NumberGenerator {
    private NumberGenerator decoratedObject;

    public BufferedNumberGenerator(NumberGenerator decoratedObject) {
        this.decoratedObject = decoratedObject;
    }

    public void computeNext() { decoratedObject.computeNext(); }
    public void reset() { decoratedObject.reset(); }
    public long getValue() { return decoratedObject.getValue(); }

    public long[] getSequence(int n) {
        long[] numArray = new long[n];
        for (int i = 0; i < n; i++) {
            numArray[i] = decoratedObject.getValue();
            decoratedObject.computeNext();
        }
        return numArray;
    }
}
```

Delegation

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 20 25

Singleton

- Ensures only a **single instance** of a class exists.
 - All clients use the same object.
- Constructor is private to prevent external instantiation.
- Single instance obtained via a static **getInstance** method.
- Example: Canvas in *shapes* project.

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 20 26

Singleton Java implementation template

```
public class Singleton {
    private static Singleton instance = null;

    // Private prevents external object creation
    private Singleton() {}

    // Return the single object of this class
    // create (lazily) if necessary
    public static synchronized Singleton getInstance() {
        if (instance == null)
            instance = new Singleton();

        return instance;
    }
}
```

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 20 27

Singleton: an example

```
public class TicketMachine {
    private static TicketMachine instance = null;
    private int count;

    // This forbids external object creation
    private TicketMachine() { count = 0; }

    // Return the single object of this class,
    // create if necessary
    public static synchronized TicketMachine getInstance() {
        if (instance == null)
            instance = new TicketMachine();
        return instance;
    }

    public synchronized int getTicket() {
        return ++count;
    }
}
```

Ticket numbers are guaranteed to be **unique** as there can only exist one object of this class.

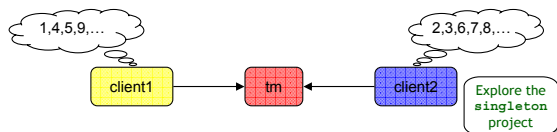
Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 20 28

Using the ticket machine

```
//Client 1
TicketMachine tm = TicketMachine.getInstance();
int ticket = tm.getTicket();
...

//Client 2
TicketMachine tm = TicketMachine.getInstance();
int ticket = tm.getTicket();
...
```



Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 20 29

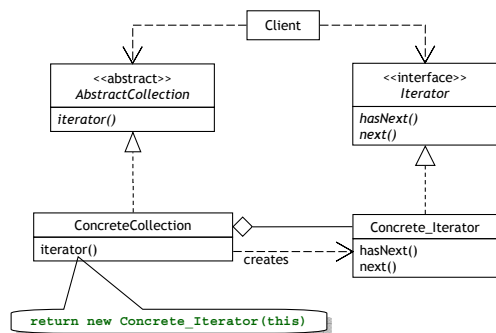
Factory method

- A creational pattern.
- Clients require an object of a particular interface type or superclass type.
- A factory method is free to return an implementing-class object or subclass object.
- Exact type returned depends on context.
- Example: **iterator** methods of the Collection classes.

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 20 30

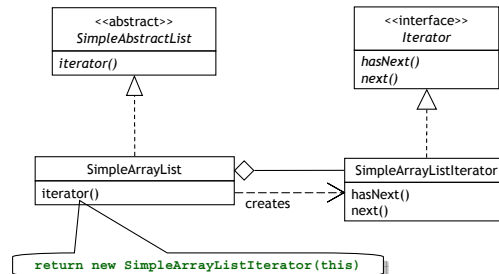
Factory method (2)



Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 20 31

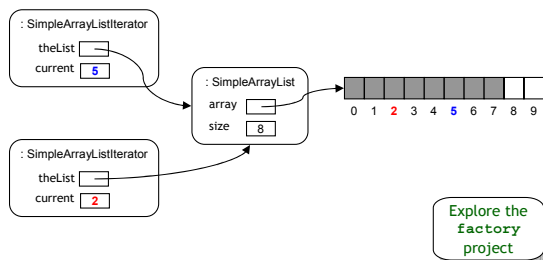
Example: SimpleArrayList



Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 20 32

Example: SimpleArrayList (2)



Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 20 33

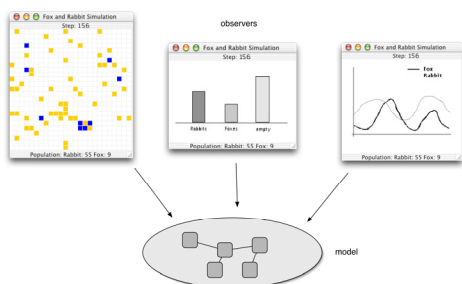
Observer

- Supports separation of internal model from a view of that model.
- Observer defines a one-to-many relationship between objects.
- The object-observed notifies all Observers of any state change.
- Example SimulatorView in the *foxes-and-rabbits* project.

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 20 34

Observers



Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 20 35

Review

- Design flexible, extendible and maintainable class structures.
- Being aware of existing design patterns will help you to do this.

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 20 36