Lesson 4
Design Patterns

Software Analysis and Design

2nd course of Computer Science

Universidad Autónoma de Madrid

Content

- **Introduction**
  - **Design Patterns**
  - **Describing design patterns**
  - **Example: Design patterns in Smalltalk MVC**
  - **The design pattern catalogue**
- Creational Design Patterns
- Structural Design Patterns
- Behavioural Design Patterns
- Conclusions
- Bibliography

Introduction

- Recurrent patterns of classes and communication between objects found in many design solutions
- Specific design for a problem, but generic enough to be adapted to future requirements and problems
- Reusing solutions that have been useful in the past
  - Avoid creating new designs from zero as much as possible
  - Avoid solving each problem from scratch

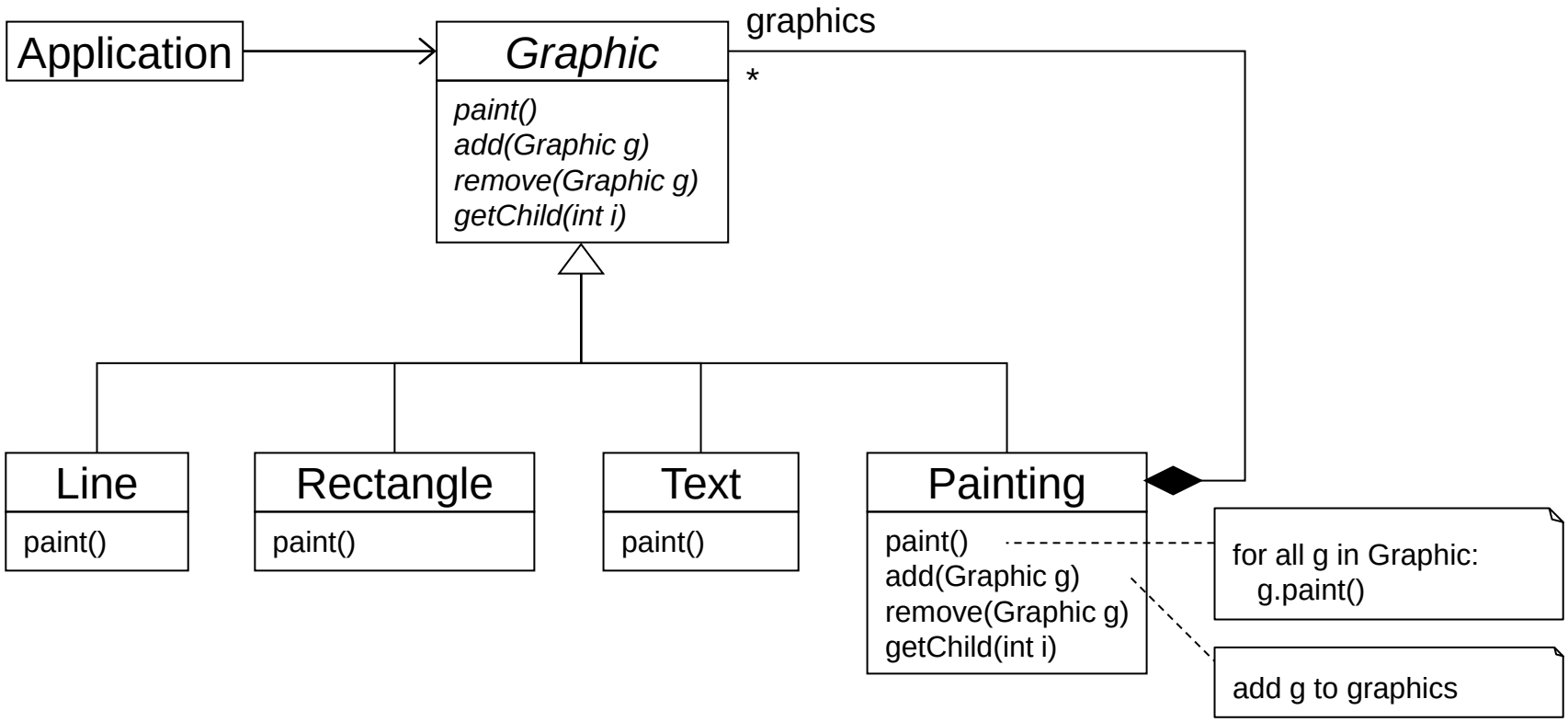
Pattern

- A schema that is followed to solve a problem
- The schema has been tested extensively and effectively. There is experience in its use
- Patterns exist in many domains:
  - Literature and cinema: "*tragic hero*", "*romantic comedy*", etc.
  - Art.
  - Engineering.
  - Architecture.
    - Christopher Alexander. "*A Pattern Language: Towns, Buildings, Construction". 1977.*
    - http://www.patternlanguage.com

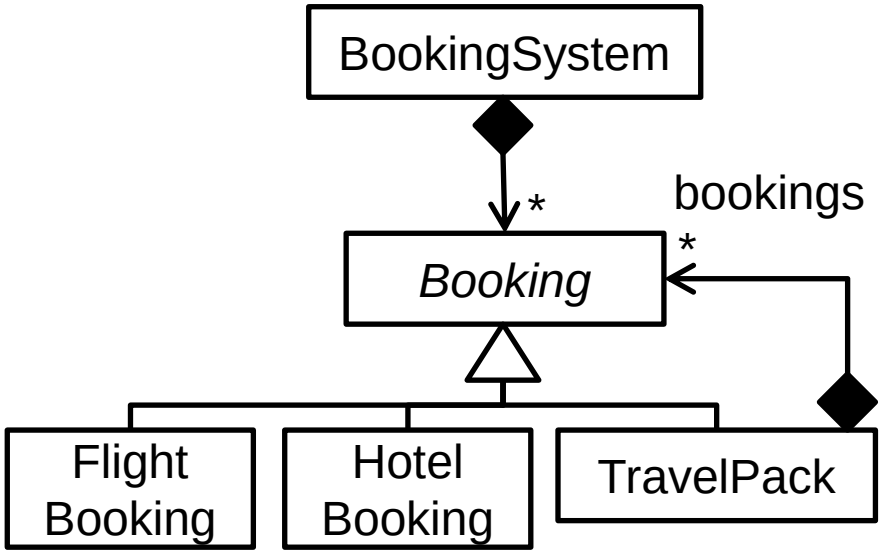
Design Patterns

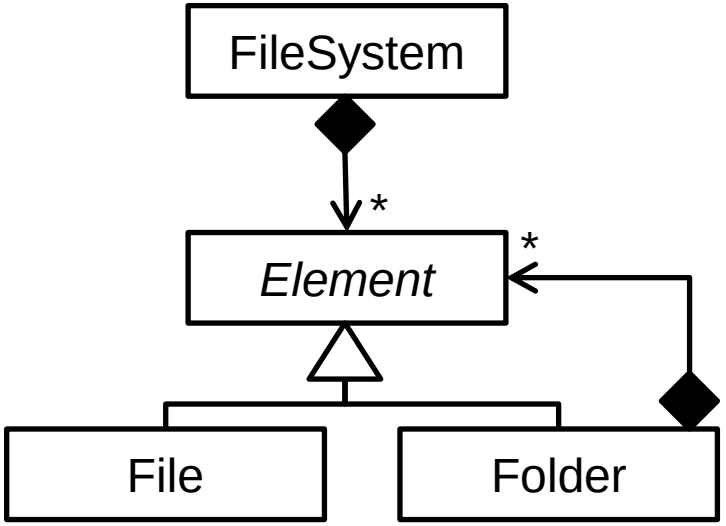
- Reusing abstract designs that do not include implementation details
- A pattern is a problem description and the essence of its solution, which can be reused in other cases
  - It is an appropriate solution for a common problem
  - It is associated to the object oriented paradigm, but in principle, patterns can be used with all software design approaches

Example


Application
Graphic
paint()
add(Graphic g)
remove(Graphic g)
getChild(int i)
graphics
*
Line
paint()
Rectangle
paint()
Text
paint()
Painting
paint()
add(Graphic g)
remove(Graphic g)
getChild(int i)
for all g in Graphic:
g.paint()
add g to graphics

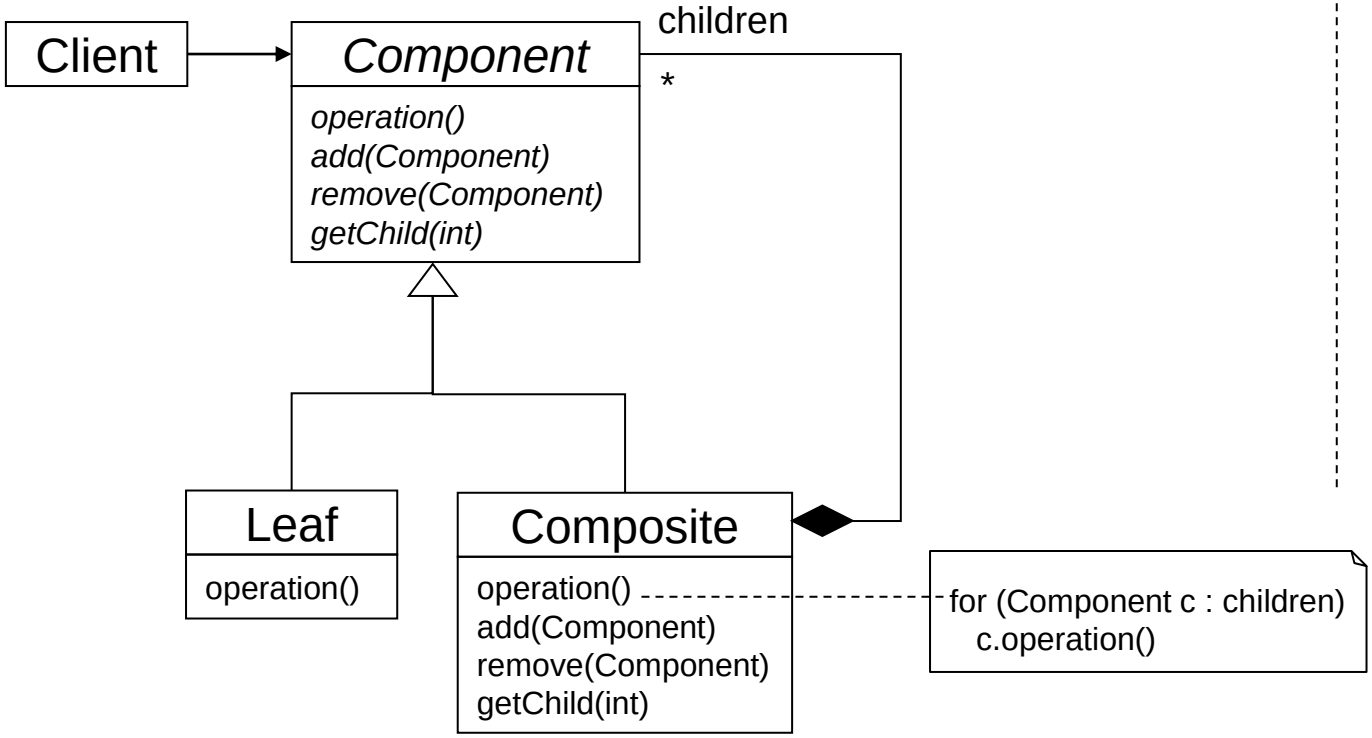
Examples


BookingSystem
*
bookings
*
Booking
Flight Booking
Hotel Booking
TravelPack

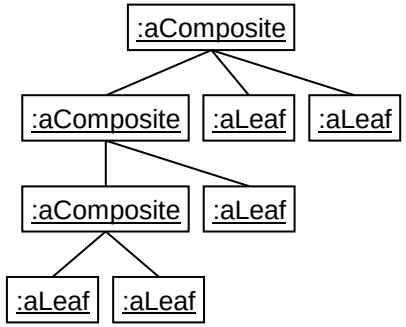

FileSystem
*
*
Element
File
Folder

Composite

Structure



Example of object structure:



- GoF's contributions in design pattern
  - Gang of Four was a team of four members: Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides.

(L-R) Ralph, Erich, Richard and John

Design Patterns

- Documenting design experience as a catalogue of patterns
- Pattern categories:
  - **Creational**: instantiation of objects
  - **Structural**: composition of objects
  - **Behavioural**: how objects communicate, cooperate and distribute their responsibilities to achieve their goals.

Design Patterns

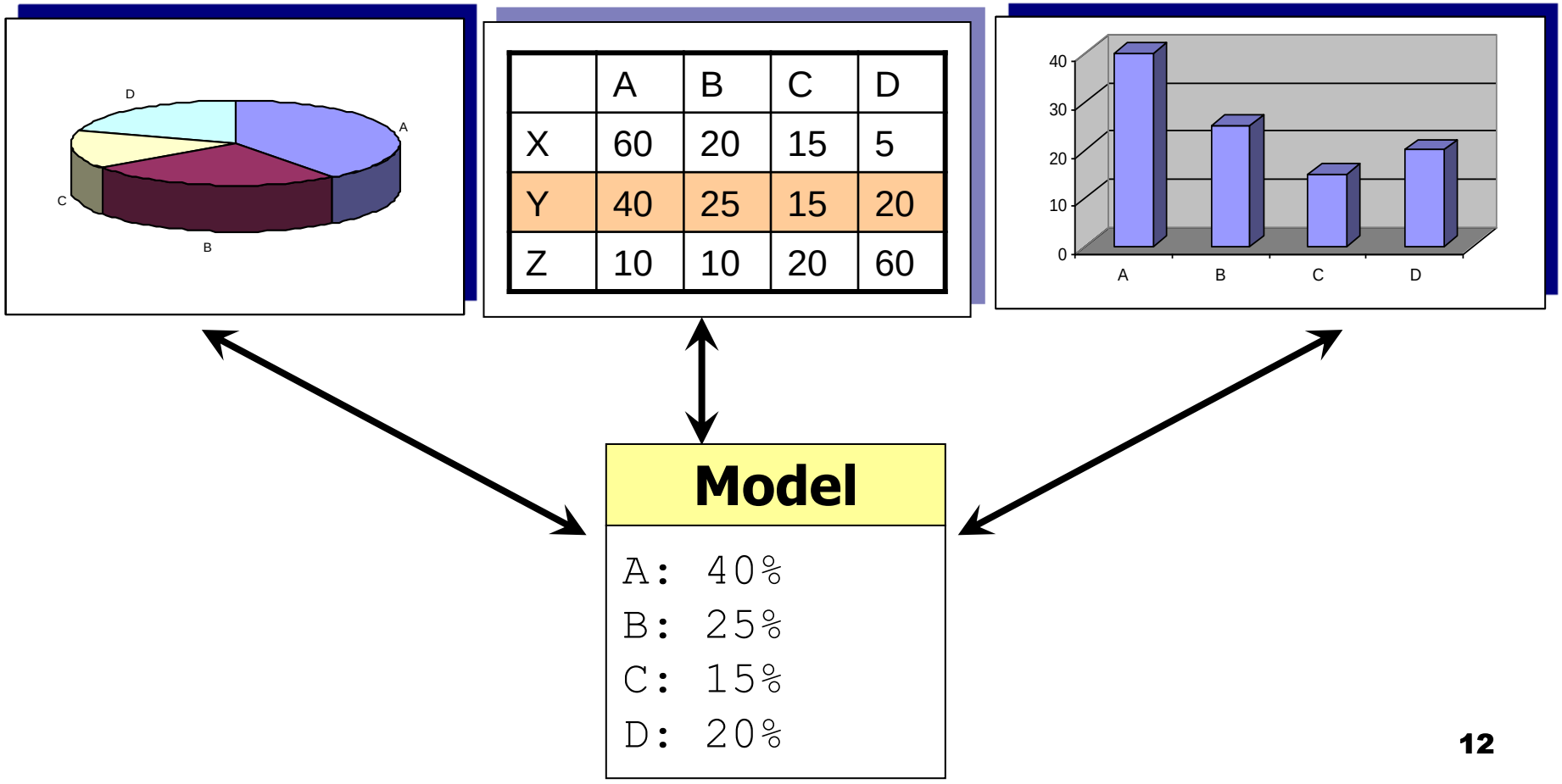
Pattern structure

- ***Pattern name.***
  - Description of the design pattern, as well as its solutions and consequences
  - Design vocabulary
- ***Problem.***
  - Description of when to apply the pattern
  - Explanation of the problem and its context
- ***Solution.***
  - Elements that form the design, relationships and responsibilities
  - Not a concrete design, but a template that can be applied in many different situations
- ***Consequences.***
  - Results, advantages and disadvantages of the pattern application
  - E.g.: relation between memory and computational time costs; implementation issues, etc

Design Patterns

Example: Smalltalk MVC

- Model–View–Controller



Design Patterns

Example: Smalltalk MVC

- Separating objects from the data (model), visualizations (view) and ways in which the interfaces react to user input (controller)
- Separating these components to increase flexibility and reuse
- Decoupling models from views through a subscription-notification protocol
- Notifying (and updating) views about changes on data of related models

Design Patterns

Example: Smalltalk MVC

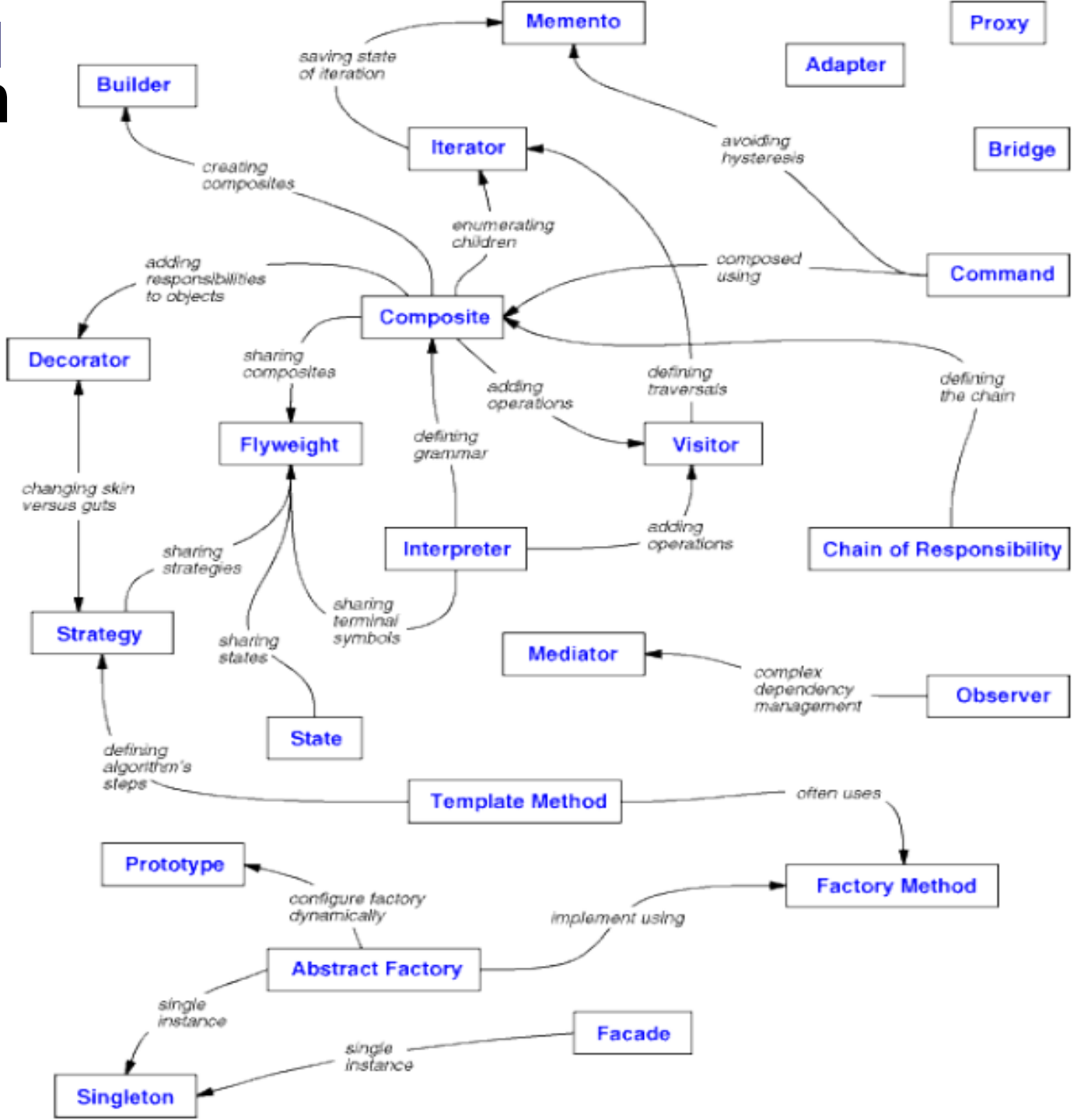
- *Observer* pattern, more generic (dependencies between objects)
- The views can be nested: simple and composite views
  - Generalization: *Composite* pattern
- Relationship between view and controller: *Strategy* pattern (an object that represents an algorithm)
- Other patterns:
  - *Factory Method*: it specifies the controller class predefined for a view
  - *Decorator*: it adds scrolling capacity to a view

The design pattern catalogue

		Purpose		
		Creational	**Structural**	**behavioural**
Scope	**Class**	Factory Method	Adapter (of classes)	Interpreter. Template Method.
	Objet	Abstract Factory Builder Prototype Singleton	Adapter (of objects). Bridge. Composite. Decorator. Facade. Flyweight. Proxy.	Chain of Responsibility. Command. Iterator. Mediator. Memento. Observer. State. Strategy. Visitor.

The pattern catalogue

Relationships between patterns.



Content

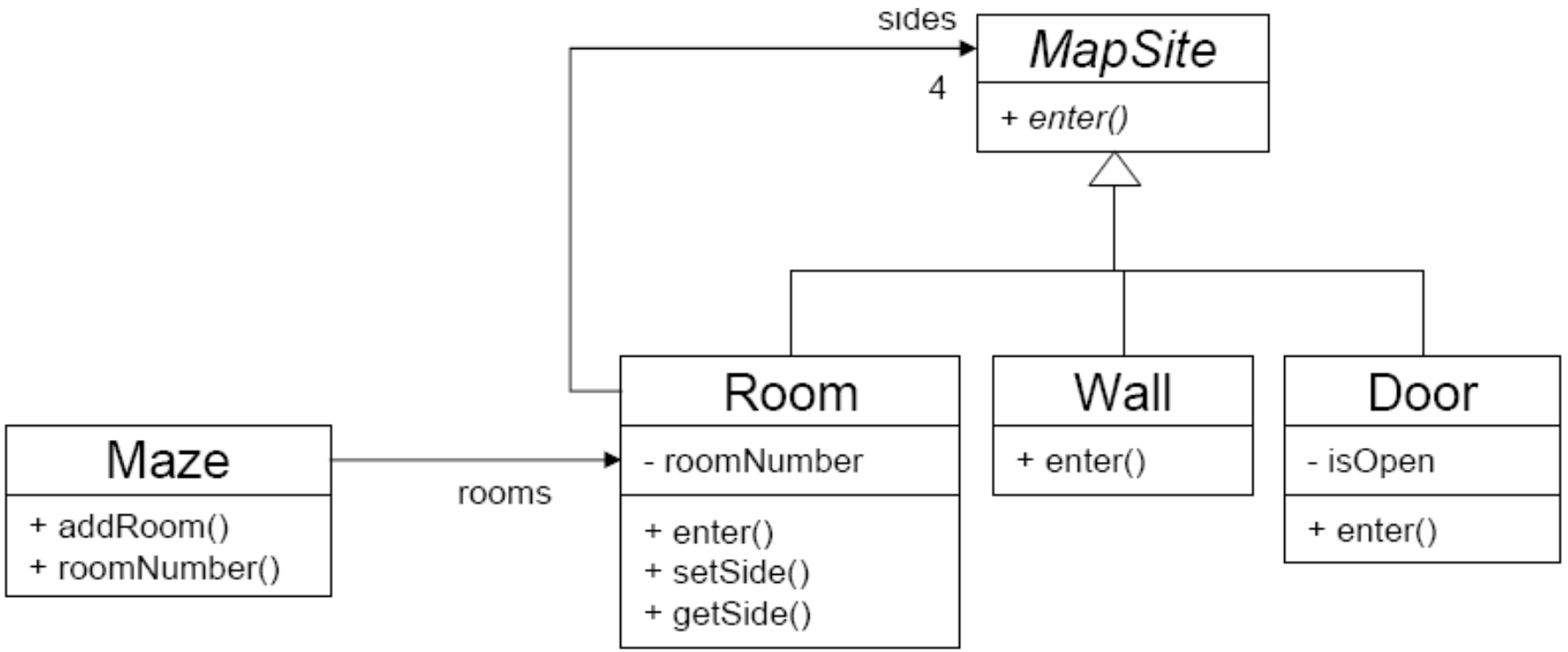
- Introduction
- **Creational Design Patterns**
  - **Factory Method**
  - **Abstract Factory**
  - **Singleton**
- Structural Design Patterns
- Behavioural Design Patterns
- Conclusions
- Bibliography

Creational Design Patterns

- Abstraction of the instantiation process
- Help in keeping the system independent of how its objects are created, represented and composed
- Flexibility on:
  - what is created
  - who creates it
  - how it is created
  - when it is created
- Allow configuring the system:
  - statically (compile-time)
  - dynamically (run-time)

Example

- Labyrinth:



Example

```
public enum Direction {
  NORTH,
  SOUTH,
  EAST,
  WEST
}
public abstract class MapSite {
  abstract void enter();
}

public class Room extends MapSite {
  private int roomNumber;
  private MapSite sides[]=new MapSite[4];
  Room () {}
  Room (int n) { roomNumber = n; }
  MapSite getSide (Direction dir) {
    return sides[dir.ordinal()];
  }
  void setSide (Direction dir, MapSite s){}
  void enter() {}
}
```

```
public class Wall extends MapSite {
  Wall () {}
  void enter() {}
}

public class Door extends MapSite {
  private Room room1;
  private Room room2;
  private boolean isOpen;
  Door (Room r1, Room r2) {}
  void enter() {}
  Room otherSideFrom (Room r1) {}
}

public class Maze {
  Maze() {}
  void addRoom (Room r) {}
  Room RoomNumber (int n) { }
}
```

Example

```
public class MazeGame {
  Maze createMaze () {
    Maze aMaze = new Maze();
    Room r1 = new Room(1);
    Room r2 = new Room(2);
    Door aDoor = new Door(r1, r2);
    aMaze.addRoom(r1);
    aMaze.addRoom(r2);
    r1.setSide(Direction.NORTH, new Wall());
    r1.setSide(Direction.EAST, aDoor);
    r1.setSide(Direction.SOUTH, new Wall());
    r1.setSide(Direction.WEST, new Wall());
    r2.setSide(Direction.NORTH, new Wall());
    r2.setSide(Direction.EAST, new Wall());
    r2.setSide(Direction.SOUTH, new Wall());
    r2.setSide(Direction.WEST, aDoor);
    return aMaze;
  }
}
```

Large: four invocations to `setSide` from Room.
We can initialize the room in the constructor.

Not very flexible:
Other forms of labyrinth?
- changing the method
- adding a new method

Other types of labyrinth?

Example

```
public class MazeGame {
  Maze createMaze () {
    Maze aMaze = new Maze();
    Room r1 = new Room(1);
    Room r2 = new Room(2);
    Door aDoor = new Door(r1, r2);
    aMaze.addRoom(r1);
    aMaze.addRoom(r2);
    r1.setSide(Direction.NORTH, new Wall());
    r1.setSide(Direction.EAST, aDoor);
    r1.setSide(Direction.SOUTH, new Wall(
    r1.setSide(Direction.WEST, new Wall()
    r2.setSide(Direction.NORTH, new Wall());
    r2.setSide(Direction.EAST, new Wall());
    r2.setSide(Direction.SOUTH, new Wall());
    r2.setSide(Direction.WEST, aDoor);
    return aMaze;
  }
}
```

Factory method: creation functions instead of constructors => changing the type of what is being created through redefinition

Abstract factory: object to create objects => changing the type of what is being created by receiving a different object

Singleton: a unique labyrinth object in the game

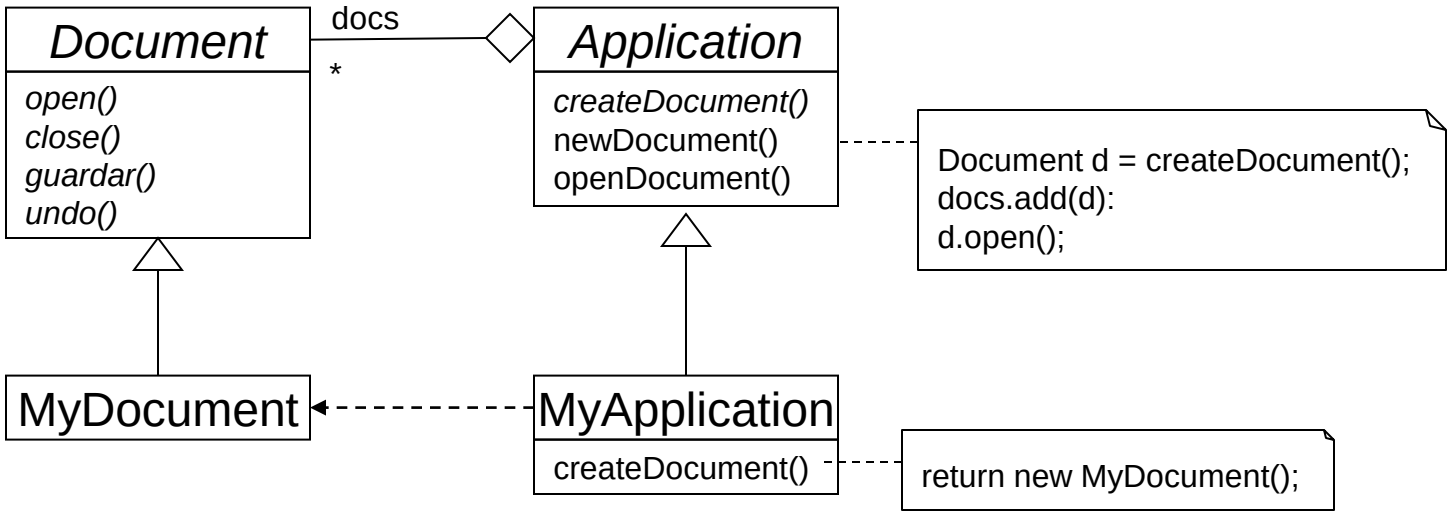
Factory Method

- **Purpose**
  - Defining an interface to create an object, but allowing subclasses to decide which class to instantiate
  - Enabling a class to delegate the creation of objects to its subclasses
- **Also known as**
  - Virtual Constructor.

Factory Method

- **Motivation**
  - An application framework should be able to present different types of documents.
  - The framework handles two abstractions
    - *Document*: its different types are defined as subclasses
    - *Application*: knows when to create a document, but not its type (it cannot predict the document type the programmer will define).
  - Solution
    - Encapsulating the knowledge about *Document* subclasses
    - Creating and moving such knowledge out of the framework

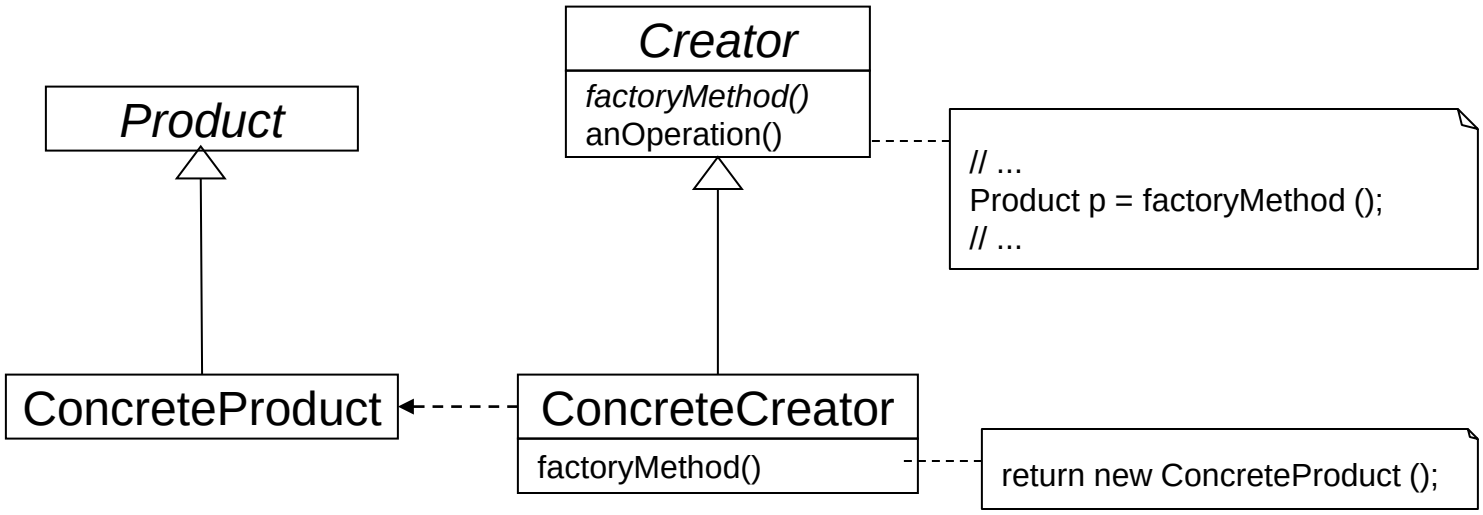
Factory Method



Factory Method

- Use the *Factory Method* pattern when:
  - A class cannot predict the class of the objects it has to create
  - A class wants its subclasses to decide which objects to create
  - The classes delegate responsibilities to one out of several auxiliary subclasses, and we want to identify which particular subclass has the creation responsibility

Structure


Creator
factoryMethod()
anOperation()
// ...
Product p = factoryMethod ();
// ...
Product
ConcreteProduct
ConcreteCreator
factoryMethod()
return new ConcreteProduct ();

Question

- How can we implement this pattern using lambda expressions?

Answer

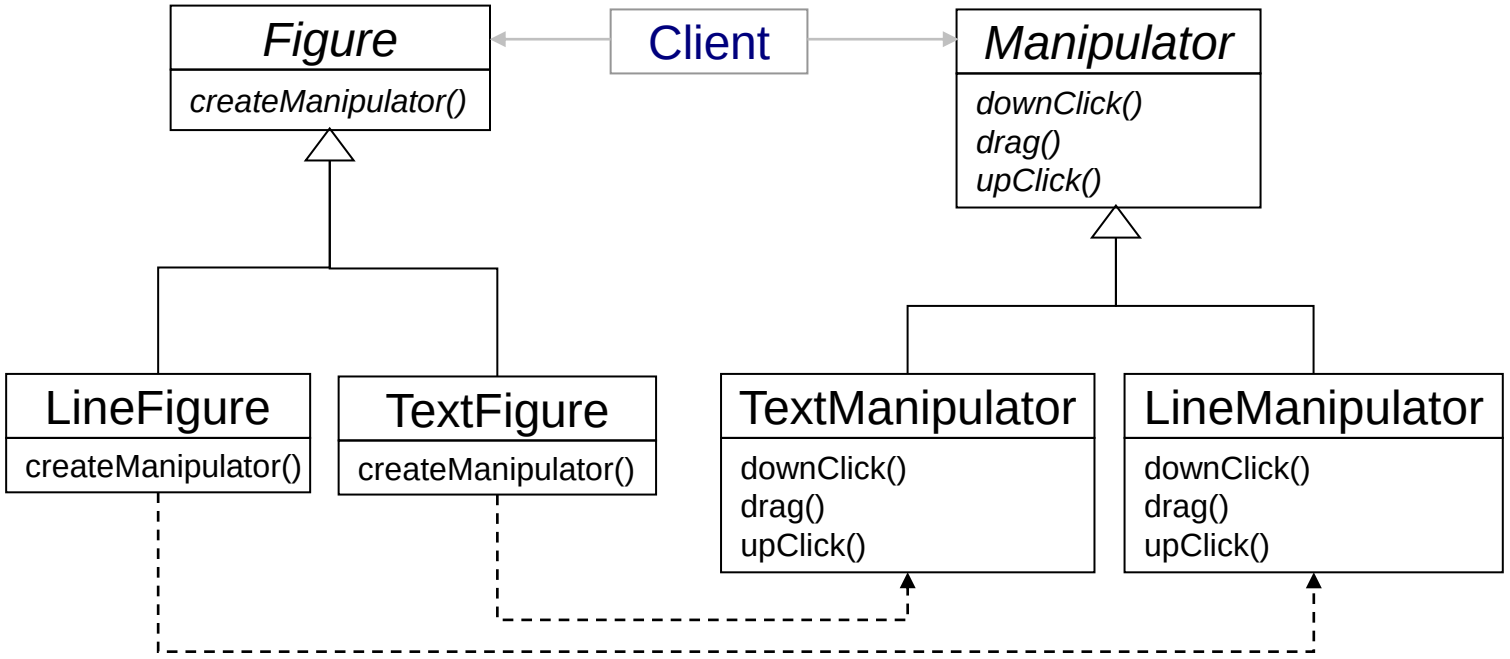
```
public class Application {
  private Supplier<? extends Document> docSupplier;
  private List<Document> docs = new ArrayList<Document>();

  public Document createDocument() {
    return this.docSupplier.get();
  }

  public void setProductSupplier(Supplier<? extends Document> sup) {
    this.docSupplier = sup;
  }
  public void newDocument() {
    Document d = this.createDocument();
    this.docs.add(d);
    d.open();
  }
}
```

Answer

```
public class Main {
  public static void main(String[] args) {
    Application ap = new Application();
    ap.setProductSupplier(MyDocument::new);
    // ap.setProductSupplier(() -> new MyDocument());
    // is equivalent
    ap.newDocument();
  }
}
```

Consequences

- Connection of hierarchies of parallel classes (delegation)



Example

```
public class MazeGame {
   // factory methods
   Maze makeMaze () { return new Maze(); }
   Wall makeWall () { return new Wall(); }
   Room makeRoom (int n) { return new Room(n); }
   Door makeDoor (Room r1, Room r2) { return new Door(r1, r2); }

   // create maze
   Maze createMaze () {
      Maze aMaze = makeMaze();
      Room r1 = makeRoom(1), r2 = makeRoom(2);
      Door aDoor = makeDoor(r1, r2);
      aMaze.addRoom(r1);
      aMaze.addRoom(r2);
      r1.setSide(Direction.NORTH, makeWall());
      r1.setSide(Direction.EAST,  aDoor);
      r1.setSide(Direction.SOUTH, makeWall());
      r1.setSide(Direction.WEST,  makeWall());
      r2.setSide(Direction.NORTH, makeWall());
      r2.setSide(Direction.EAST,  makeWall());
      r2.setSide(Direction.SOUTH, makeWall());
      r2.setSide(Direction.WEST,  aDoor);
      return aMaze;
}}
```

Exercises

- Using the “factory method” design pattern, create a generic class SortableStore that can be configured with a sort order (which will be the natural order by default).

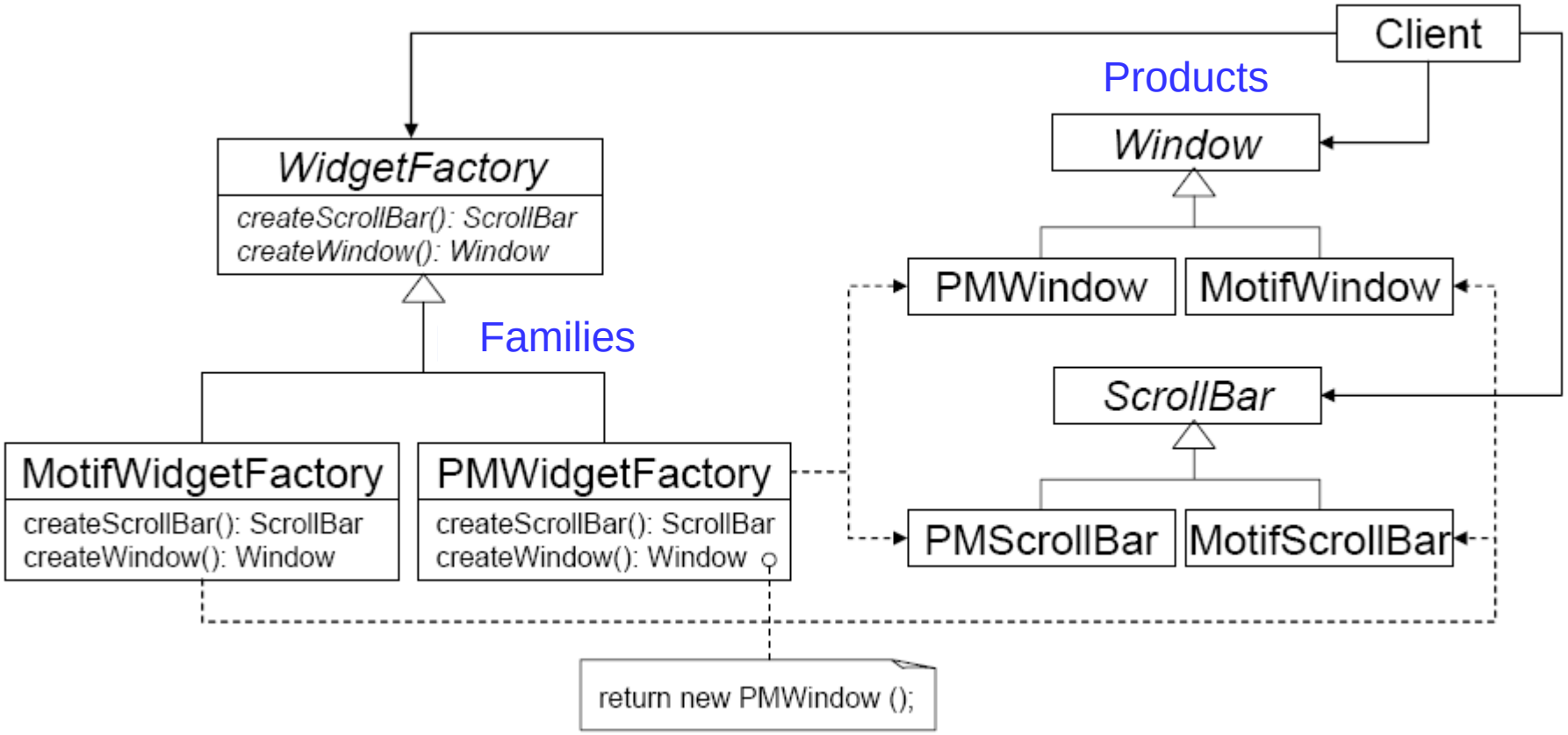
- Create a program that configures the class to sort a collection of Strings by size.

Abstract Factory

- **Purpose**
  - Providing an interface to create families of objects related or dependent to each other, without specifying their particular classes
- **Also known as**
  - Kit
- **Motivation**
  - E.g.: a framework for building user interfaces that allow several "*look and feel*" (e.g.: Motif, Presentation Manager)
    - An abstract class *WidgetFactory* with the interface to create each widget type
    - An abstract class for each widget type. Concrete subclasses that implement each particular widget
    - This way, the clients do not depend on a particular *look and feel*

Abstract Factory

- Motivation

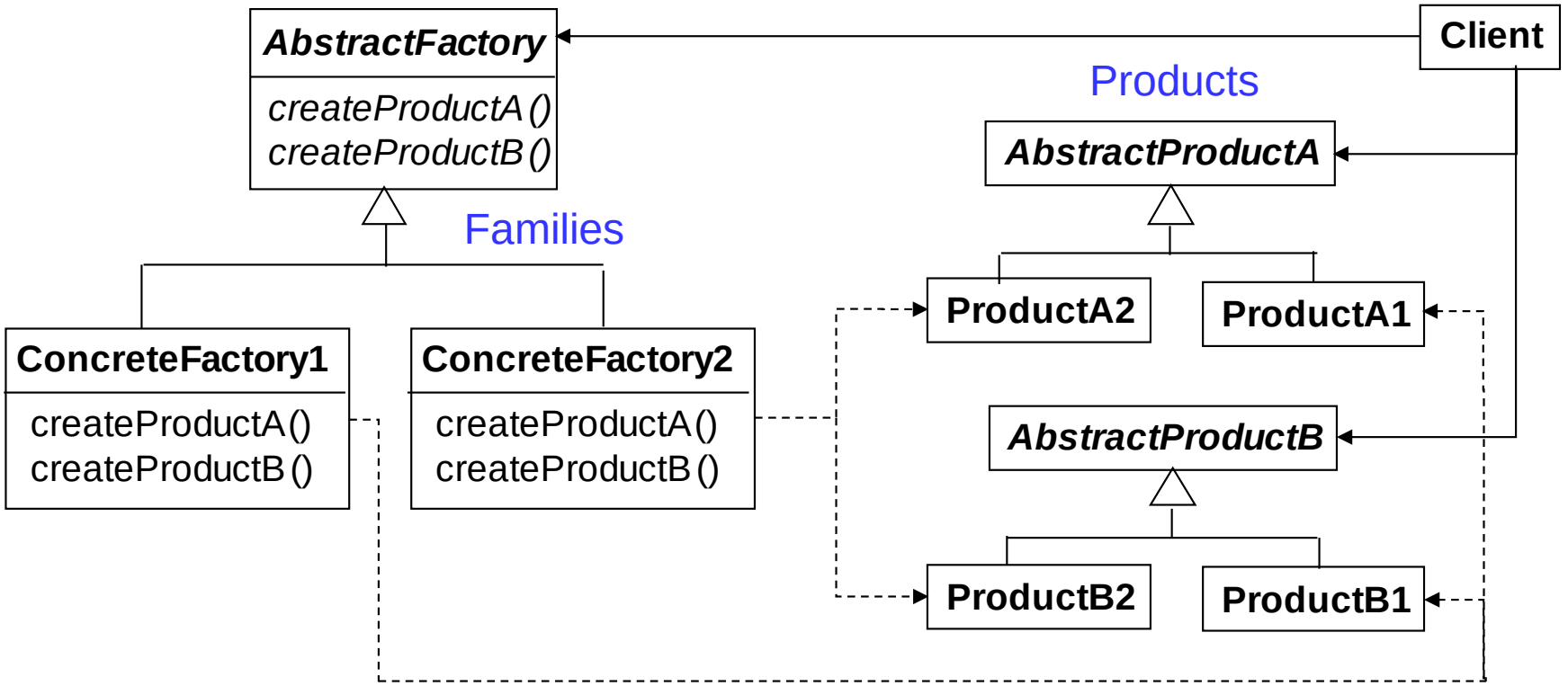


Abstract Factory

- **Application.** Use this pattern when
  - A system has to be independent of how its products are created, represented and composed
  - A system has to be configured with a family of products from several families
  - A family of related products has been designed to be used jointly, and it is required to satisfy such constraint
  - A library of classes has to be provided, revealing their interfaces and hiding their implementations

Abstract Factory

- Structure



Abstract Factory

- **Consequences**
  - Isolating concrete classes.
    - Helps in controlling the types of classes that are created in an application
    - Keeps clients independent from the implementation classes
  - Facilitates the replacement of product families
  - Promoting consistency between products (that is, the application uses objects of a single family at a time)
  - It is difficult to add a new product

Example. *Labyrinth.*

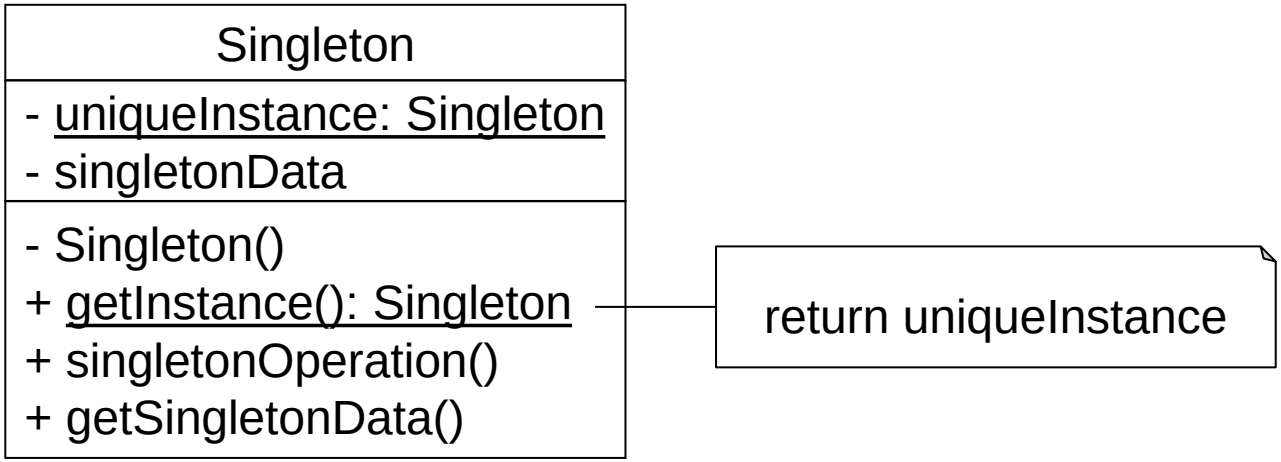
```
// abstract factory (it provides a default implementation)
public class MazeFactory {
  Maze makeMaze () { return new Maze(); }
  Wall makeWall () { return new Wall(); }
  Room makeRoom (int n) { return new Room(n); }
  Door makeDoor (Room r1, Room r2) { return new Door(r1, r2); }
}
public class MazeGame {
  Maze createMaze (MazeFactory factory) {
    Maze aMaze = factory.makeMaze();
    Room r1 = factory.makeRoom(1), r2 = factory.makeRoom(2);
    Door aDoor = factory.makeDoor(r1, r2);
    aMaze.addRoom(r1); aMaze.addRoom(r2);
    r1.setSide(Direction.NORTH, factory.makeWall());
    r1.setSide(Direction.EAST, aDoor);
    r1.setSide(Direction.SOUTH, factory.makeWall());
    r1.setSide(Direction.WEST, factory.makeWall());
    r2.setSide(Direction.NORTH, factory.makeWall());
    r2.setSide(Direction.EAST, factory.makeWall());
    r2.setSide(Direction.SOUTH, factory.makeWall());
    r2.setSide(Direction.WEST, aDoor);
    return aMaze;
}}
```

Singleton

- **Goal**
  - Ensuring that a class has a unique instance, and providing a single point access to such instance.
- **Motivation**
  - Sometimes it is important to ensure that a class has only one instance, e.g.:
    - A windows manager
    - A print queue
    - A single file system
    - A single log file or repository
  - How to ensure it? A global variable acting as the accessible object, but can be instantiated several times
  - Responsibility of the same class: manager of messages to create instances

Singleton

- **Application**
  - A unique instance that has to be accessed by clients from a single known access
  - A unique instance that has to be extensible by means of subclassification, and clients have to be capable of using an extended instance without modifying its code
- **Structure**



Singleton

- **Participants**
  - Singleton
    - It defines an operation over the class, called `getInstance()`, which allows clients to access to the unique instance
    - It can be responsible of creating its unique instance
- **Collaborations**
  - Clients access the instance of a Singleton through the `getInstance()` operation
- **Consequences**
  - Controlled access to the unique instance
  - Reduced name space. An improvement over the use of global variables
  - Allows operation refinement and representation. The Singleton can be subclassified and the application can be configured with an instance of such class
  - Easily modifiable to enable a large number of instances
  - More flexible than the class operations

Singleton

- Implementation schema in Java

```
public class Singleton {
  private static final Singleton instance = new Singleton();
  // Protected constructor to avoid external instantiations
  protected Singleton() {
     ...
  }
  public static Singleton getInstance() {
     return instance;
  }
}
```

Singleton

- On-demand initialization.

```
public class Singleton {
  private static Singleton instance = null;
  protected Singleton() {}

  public static Singleton getInstance () {
    // lazy instantiation
    if (instance==null) instance = new Singleton();
    return instance;
  }
}

// client code
Singleton mySingleton = Singleton.getInstance();
Singleton yourSingleton = Singleton.getInstance();
```

Singleton

Example. Labyrinth.

```
public class MazeFactory {
  private static MazeFactory instance = null;
  protected MazeFactory () {}
  public static MazeFactory getInstance () {
    if (instance==null) instance = new MazeFactory();
    return instance;
  }
  public static MazeFactory getInstance (String style) {
    if (instance==null) {
      if (style.equals(“bombed”) instance = new BombedMazeFactory();
      else if (style.equals(“enchanted”)) instance = new EnchantedMazeFactory();
      else instance = new MazeFactory();
    }
    return instance;
  }
  // methods make*
  // ...
}
```

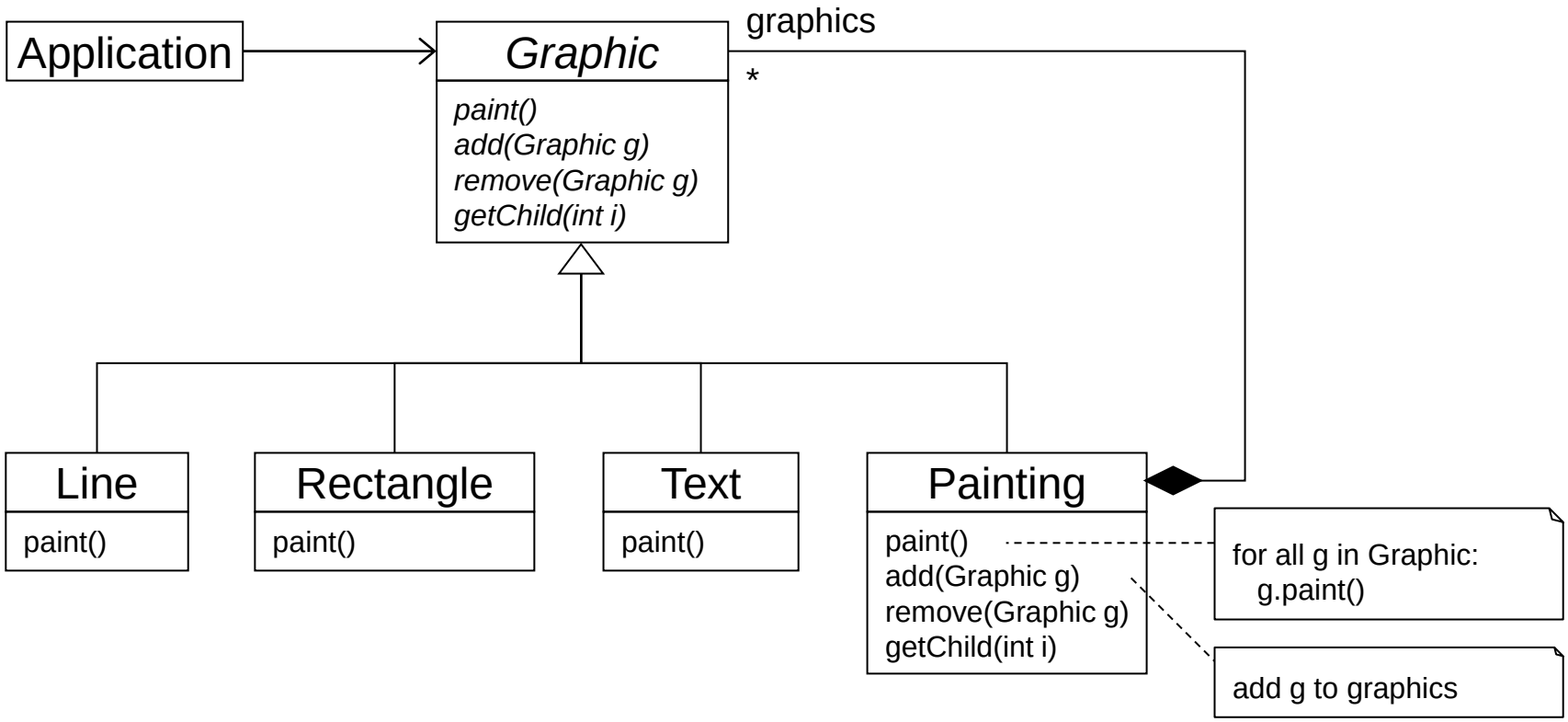
Content

- Introduction
- Creational Design Patterns.
- **Structural Design Patterns**
  - **Composite**
  - **Proxy**
- Behavioural Design Patterns
- Conclusions
- Bibliography

Composite

- **Purpose:**
  - Composing objects in tree structures to represent part-whole hierarchies. Manipulating all the objects of a tree uniformly
- **Motivation**
  - E.g.: graphical applications manage groups of figures composed of simple elements (lines, text, …)
- **Solution**
  - Having primitives for the simple components, and others for the containers? No, since they are not treated uniformly
  - Having an abstract class to represent components and containers and define their operations

Composite

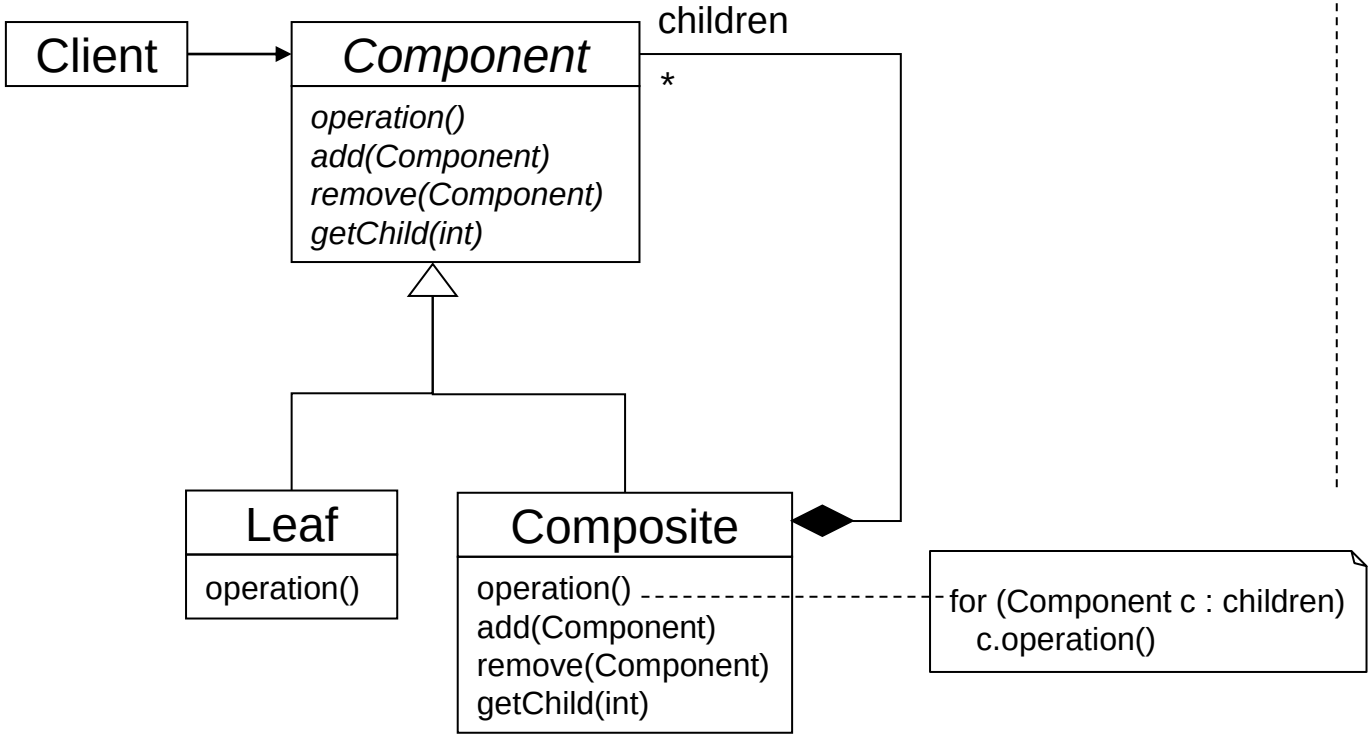

Application
Graphic
paint()
add(Graphic g)
remove(Graphic g)
getChild(int i)
graphics
*
Line
paint()
Rectangle
paint()
Text
paint()
Painting
paint()
add(Graphic g)
remove(Graphic g)
getChild(int i)
for all g in Graphic:
g.paint()
add g to graphics

Composite

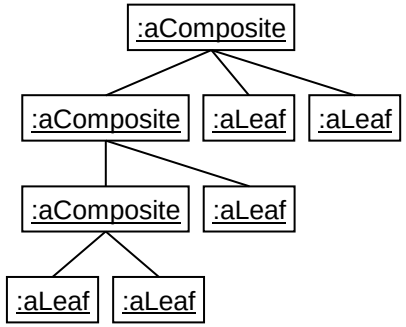
- Use the *Composite* pattern when
  - We want to represent part-whole object hierarchies
  - We want to be able to ignore the difference between individual and composed objects. Clients treat all the objects of the composed structure uniformly

Composite

Structure

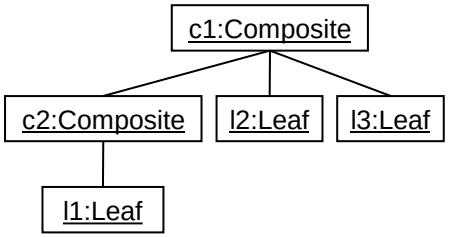


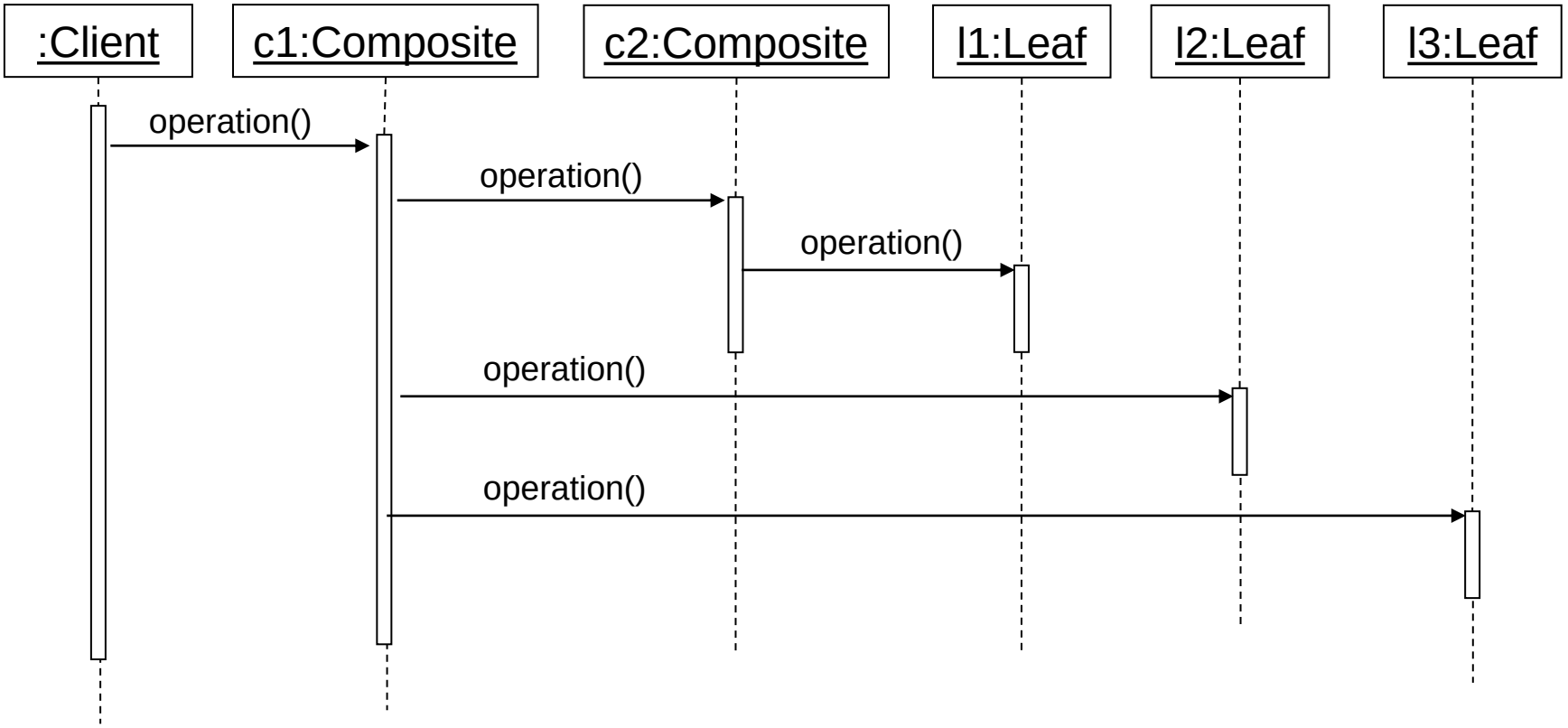
Example of object structure:



Composite

Collaborations. Example.

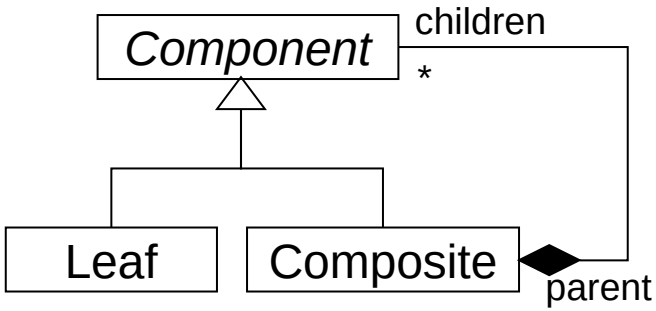




Composite

Implementation.

- Explicit references to parents
  - Simplify some operations in the composed structure
  - Defined in *Component* class
  - Managed by adding/removing elements in a *Composite*




```
public abstract class Component {
   protected Composite parent;
   public void setParent (Composite parent){
      this.parent = parent;
   }
   ...
}
public class Composite {
   protected List<Component> children;
   public void add(Component c) {
      children.add(c);
      c.setParent(this);
   }
...
}
```

- Sharing components
  - Useful for memory saving
  - More difficult management of a component with several parents

Composite

Example code

```
// Implementation with interfaces

public interface Component {
   public void add (Component c);
   public void remove (Component c);
   public Component getChild (int i);
}

public class Leaf implements Component {
   public void add (Component c)      {}  // it may throw an exception too
   public void remove (Component c)   {}  // it may throw an exception too
   public Component getChild (int i) { return null; }
}

public class Composite implements Component {
   private List<Component> children = new ArrayList<Component>();
   public void add (Component c)      { children.add (c); }
   public void remove (Component c)   { children.remove (c); }
   public Component getChild (int i) { return children.get(i); }
}
```

Composite

Example code

```
// Implementation with an abstract class

public abstract class Component {
   public void add (Component c)      {}  // it may throw an exception too
   public void remove (Component c)   {}  // it may throw an exception too
   public Component getChild (int i) { return null; }
}

public class Leaf extends Component {
}

public class Composite extends Component {
   private List<Component> children = new ArrayList<Component>();
   public void add (Component c)      { children.add (c); }
   public void remove (Component c)   { children.remove (c); }
   public Component getChild (int i) { return children.get(i); }
}
```

Composite

Example code

```
public interface Component {
   public Composite getComposite ();
}
public class Leaf implements Component {
   public Composite getComposite () { return null; }
}
public class Composite implements Component {
   private List<Component> children = new ArrayList<Component>();
   public void add (Component c)     { children.add(c); }
   public void remove (Component c)  { children.remove (c); }
   public Component getChild (int i) { return children.get(i); }
   public Composite getComposite ()  { return this; }
}

// client code
Composite aComposite = new Composite();
Leaf aLeaf = new Leaf();
Composite c = null;
Component aComponent = aComposite;
if ((c=aComponent.getComposite())!=null)
   c.add(new Leaf());  // will add the leaf
aComponent = aLeaf;
if ((c=aComponent.getComposite())!=null)
   c.add(new Leaf());  // it does not add the leaf
```

Question

When to use an abstract class instead of an interface?

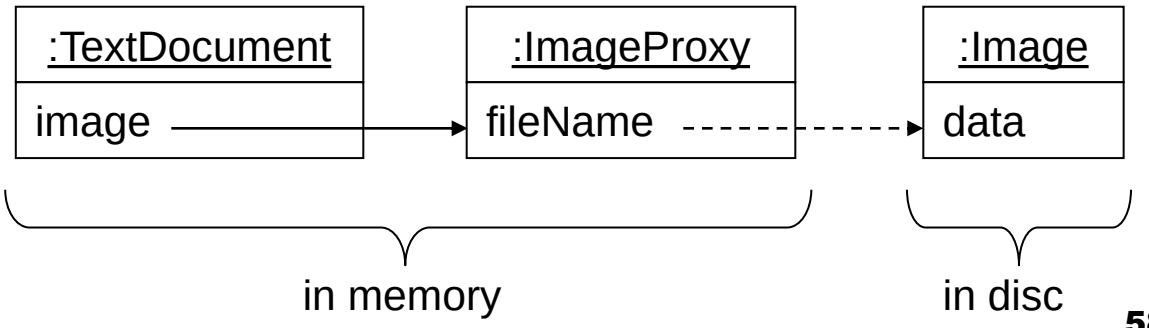
Composite

In Java...

- In the `java.awt.swing` package:
  - **Component**
    - Component
  - **Composite**
    - Container (abstract)
    - Panel (concrete)
    - Frame (concrete)
    - Dialog (concrete)
  - **Leaf**:
    - Label
    - TextField
    - Button

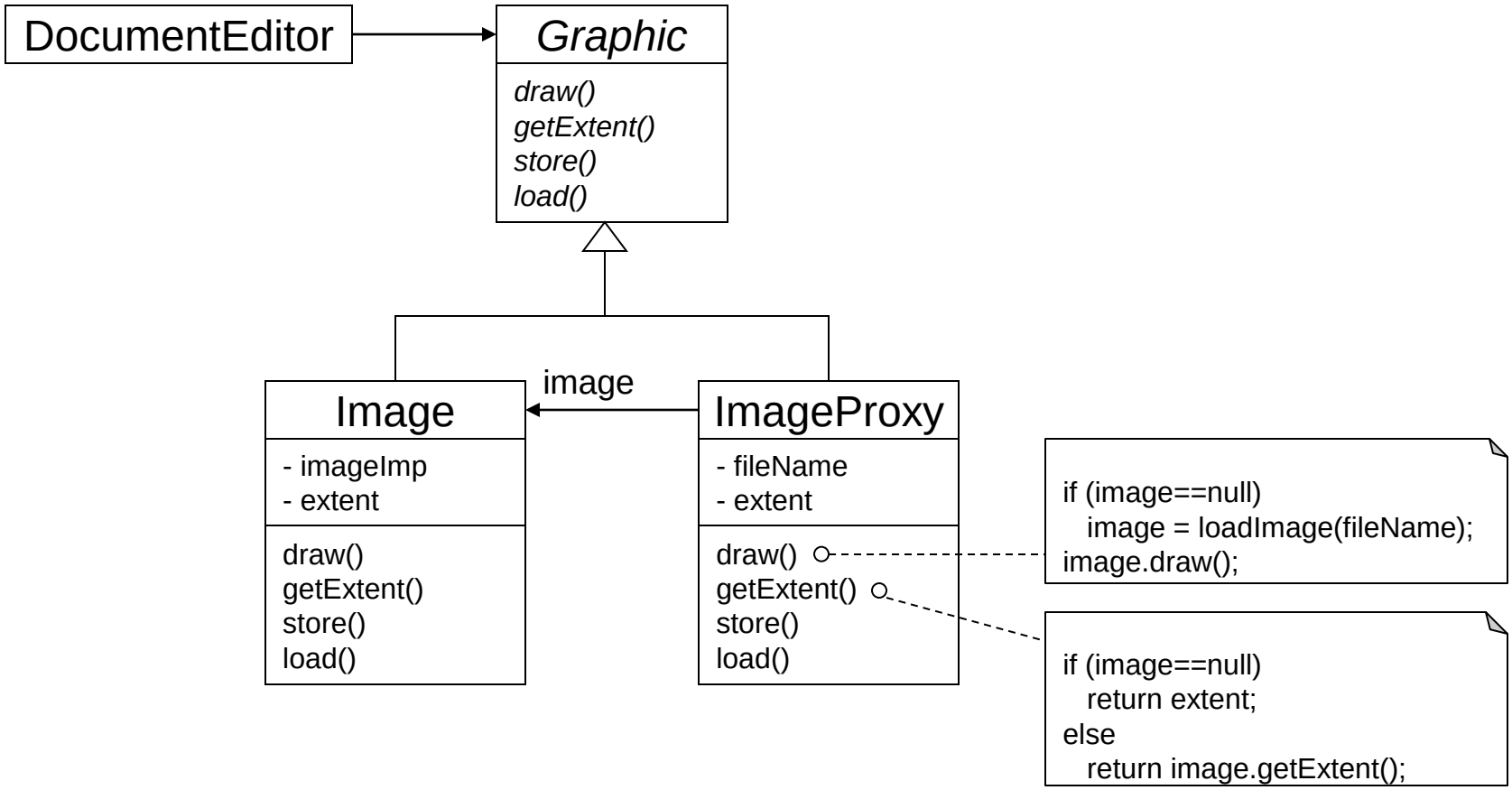
Proxy

- **Purpose**
  - Providing a representative or substitute for another object to control its access
- **Motivation**
  - Delaying the cost of creating and initializing an object until it is really needed. For instance, not opening the images of a document until all of them are visible
- **Solution**



Proxy

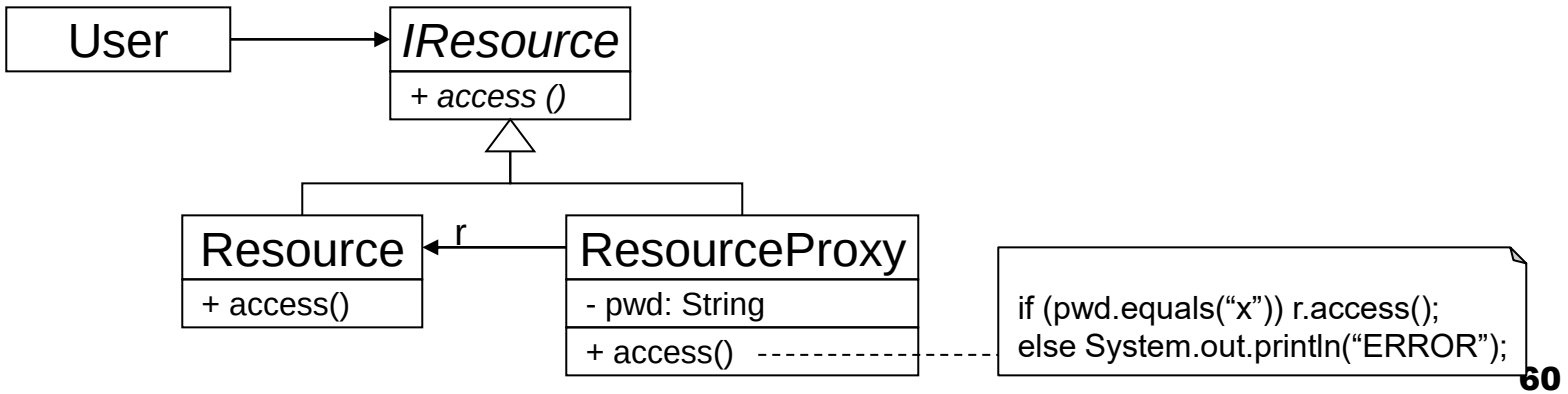
Motivation



Proxy

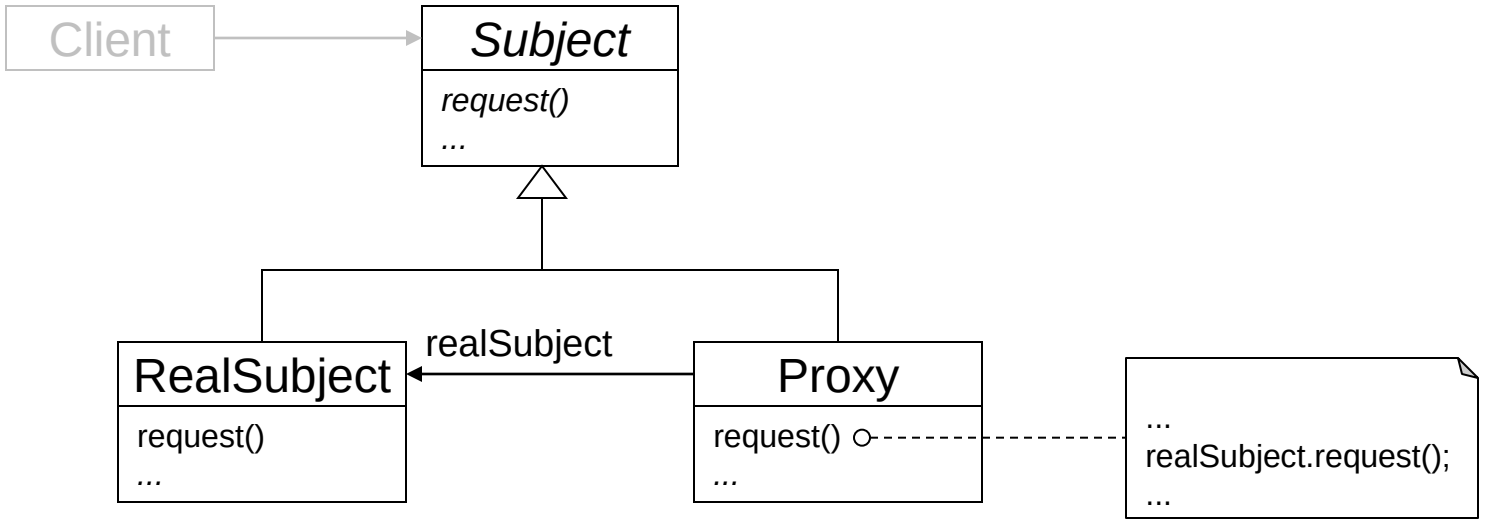
Application

- The *Proxy* pattern is used when we need an object reference that is more flexible and sophisticated than a pointer. For instance:
  - **Virtual proxy**: creates costly objects on demand (as the *ImageProxy* class in the motivation example)
  - **Remote proxy**: represents an object in another address space
  - **Intelligent reference**: substitutes a pointer that makes additional operations when accessing an object (e.g., counting the number of references, loading a persistent object in memory, blocking an object to deny concurrent access, etc.)
  - **Protection proxy**: controls the access to an object

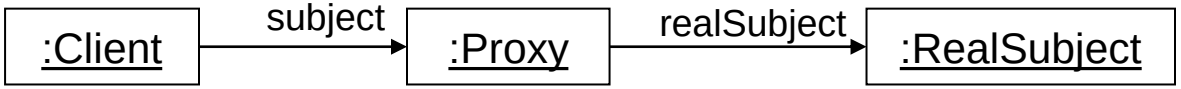


Proxy

Structure



Object diagram



Proxy

Participants

- **Proxy** (*ImageProxy*)
  - Maintains a reference to the real object
  - Provides an interface equal to the real object interface
  - Controls the access to the real object, and can be the responsible of its creation and removal
  - It may have other responsibilities that depend on the type of the proxy:
    - Remote proxies: they encode the requests and send them to the real object
    - Virtual proxies: they can store information of the real object (cache)
    - Protection proxies: they check if the client has the needed permissions to make the request
- **Subject** (*Graphic*): it defines a common interface for the proxy and the real objet, so that they can be used equally
- **RealSubject** (*Image*): class of the real object that the proxy represents

Proxy

Consequences

- Introduces a level of indirection with different uses
  - Remote proxies can hide the fact that an object belongs to other address space
  - Virtual proxies can make optimizations, such as on-demand object creation
  - Protection proxies and intelligent references allow performing management tasks, in addition to the object access
- Copy-on-write optimization
  - Copying a large object can be very costly
  - If the copy is not modified, the cost is not needed
  - The subject maintains a number of references, and only when an operation modifies the object, this is copied

Proxy

Implementation (Virtual Proxy)

```
public abstract class Graphic {
   public void draw();
}


public class Image extends Graphic {
   public void draw() { ... }
}


public class ImageProxy extends Graphic {
   private Image  image;
   private String fileName;
   public ImageProxy (String fileName) {
      this.fileName = fileName;
      this.image    = null;
   }
   public Image loadImage() { ... }
   public void draw () {
      if (image==null)
         image = loadImage(fileName);
      image.draw();
   }
}
```

```
public class TextDocument {
   public void insert (Graphic g) {
    ...
   }
}


// code to insert an ImageProxy into
// a document
TextDocument td = new TextDocument();
Graphic g =
   new ImageProxy("imagen.gif");
td.insertar(g);
```

Content

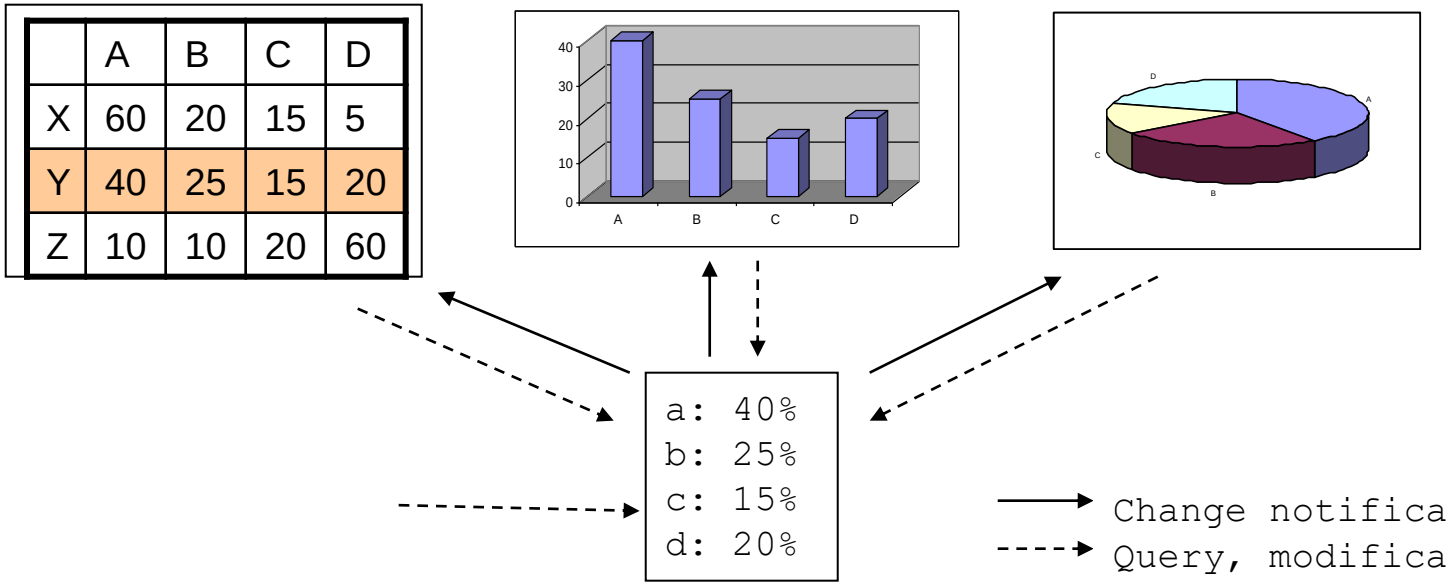
- Introduction
- Creational Design Patterns
- Structural Design Patterns
- **Behavioural Design Patterns**
  - **Observer**
  - **Iterator**
- Conclusions
- Bibliography

Observer

- **Purpose**
  - Defining a one-to-many dependency between objects so that only when an object status changes, the dependent objects are notified to be automatically updated.
  - Also known as *dependents*, *publish-subscribe*

Observer

- **Motivation**
  - Maintaining the consistency between related objects, without increasing the coupling between classes
  - E.g.: separation of the presentation layer and the data of the underlying applications in a user interface



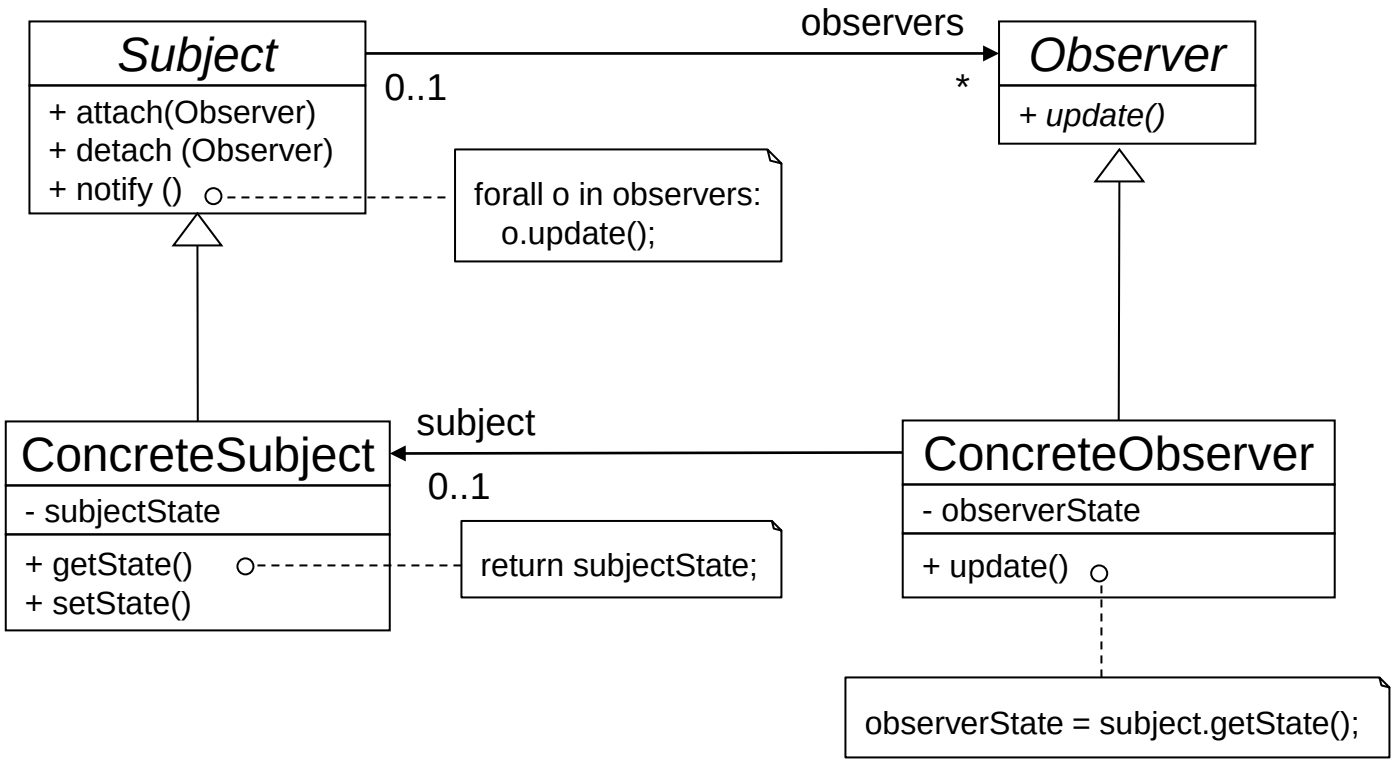
Observer

Application

- Use *Observer* pattern when:
  - An abstraction has two aspects, and one depends on the other. Encapsulating the aspects in distinct objects allows changing and reusing them
  - A change on an object implies changes on others, but we do not know how many of them have to be changed
  - An object has to be able to notify something to others without making assumptions about which are those objects; that is, when low coupling is required

Observer

Structure



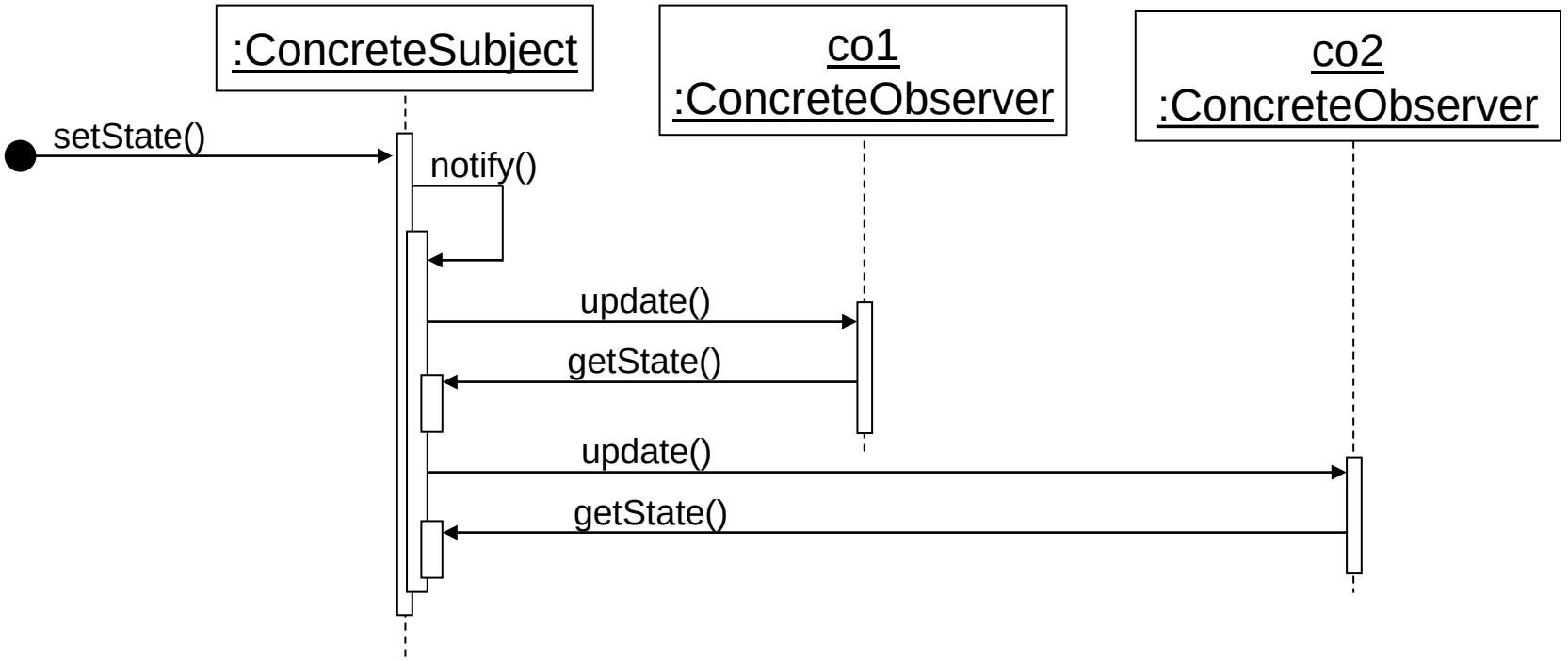
Observer

Participants

- **Subject**
  - Knows its observers, which can be a set of arbitrary size
  - Provides an interface to add and remove observer objects
- **Observer**
  - Defines the interface of the objects that should be notified about changes on the subject
- **ConcreteSubject**
  - Stores the status of interest for its observers
  - Sends notifications to its observers when its status changes
- **ConcreteObserver**
  - Maintains a reference to a *ConcreteSubject*
  - Stores the subject's status of interest
  - Omplements the interface of *Observer* to maintain its status consistent with the subject's status

Observer

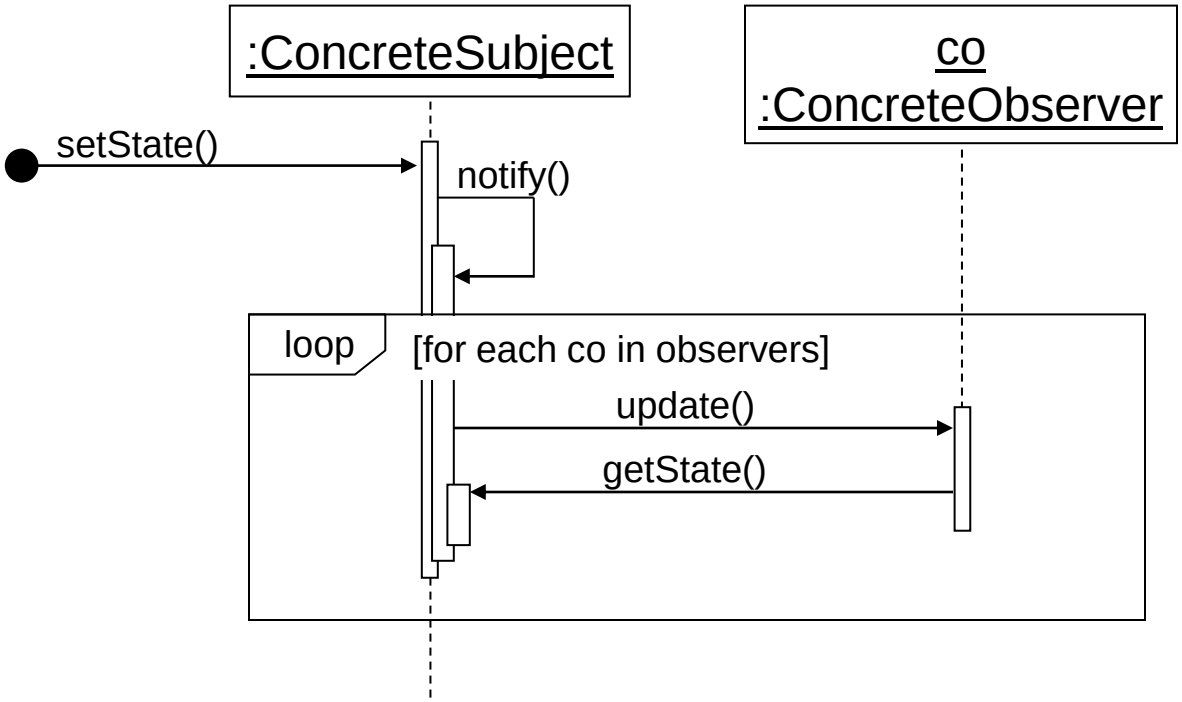
Collaborations



(subject with two observers)

Observer

Collaborations



(subject with an arbitrary number of observers)

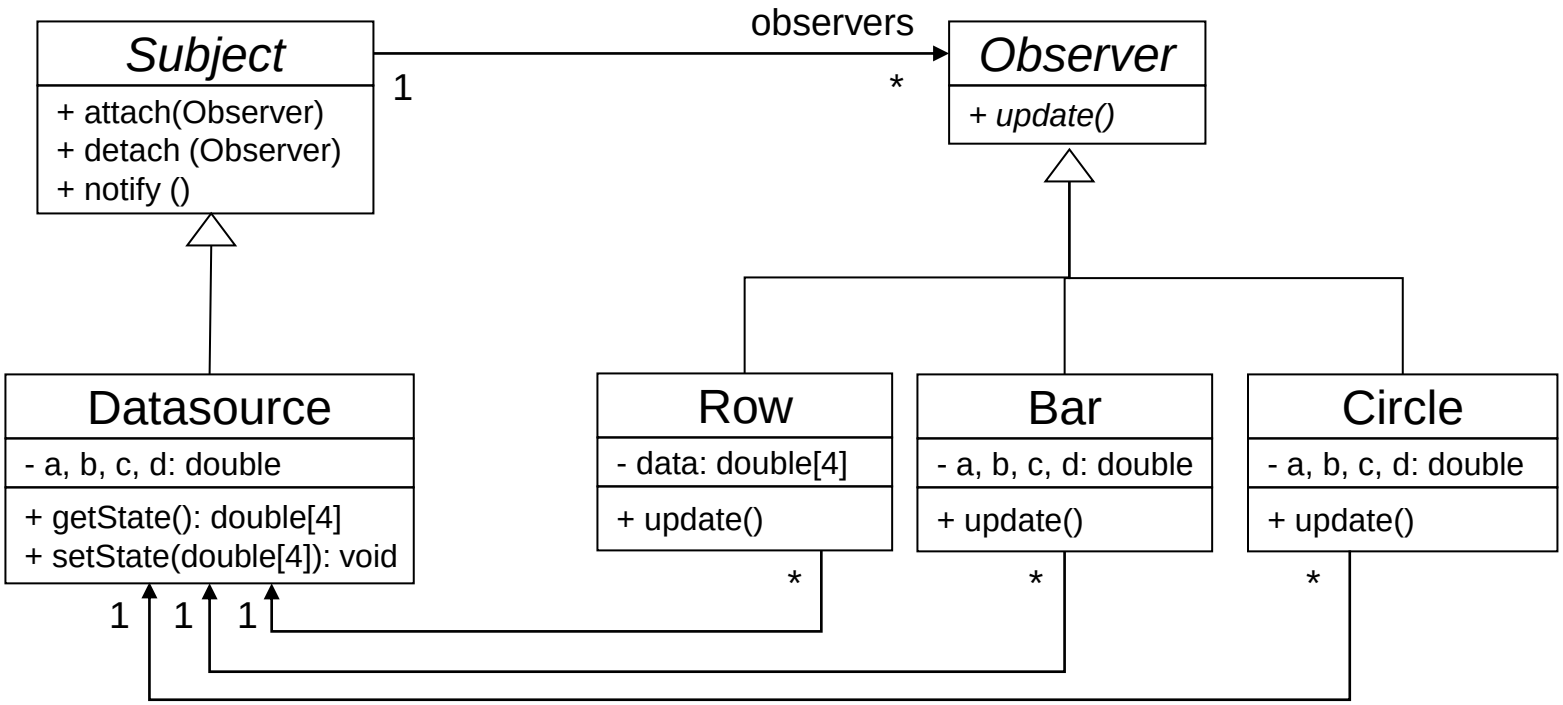
Observer

Consequences

- Allows modifying subjects and observers independently
- Allows reusing a subject without its observers, and vice versa
- Allows adding observers without the need of changing the subject and the remainder observers
- Abstract coupling between subject and observers. The subject does not know the concrete class of its observers (minimum coupling)
- Support for *broadcast*. The subject sends the notification to all subscribed observers. Observers can be added/removed
- Unexpected updates. An operation on the subject may entail a cascade of changes on its observers. The protocol does not offer details about what has changed

Observer

Example



Observer

Code of the example

```
public abstract class Subject {
 protected List<Observer> observers;

 public Subject() {
    observers =
      new ArrayList<Observer>();
 }
 public void attach(Observer o) {
    observers.add(o);
 }
 public void detach(Observer o) {
    observers.remove(o);
 }
 public void notify() {
    Iterator<Observer> it;
    it = observers.iterator();
    while (it.hasNext())
       it.next().update();
 }
}
```

```
public class Datasource
              extends Subject {
  private double a, b, c, d;
  public double[] getState ()
  {
    double[] data = new double[4];
    data[0] = a;
    data[1] = b;
    data[2] = c;
    data[3] = d;
    return data;
  }
  public void setState(double[] dd)
  {
    a = dd[0];
    b = dd[1];
    c = dd[2];
    d = dd[3];
    this.notify();
  }
}
```

Observer

Code of the example

```
public abstract class Observer {
  protected Subject subject;

  public Observer (Subject s) {
    subject = s;
    subject.attach(this);
  }

  public abstract void update();
}
```

```
public class Fila extends Observer {
  private double[] data;

  public Row (Subject s) {
    super(s);
    data = new double[4];
  }
  public void update () {
    double[4] data;
    data = ((Datasource)subject).
              getState();
    for (int i=0; i<4; i++)
       this.data[i] = data[i];
    this.redraw();
  }

  public void redraw () { ... }
}
```

Observer

In Java…

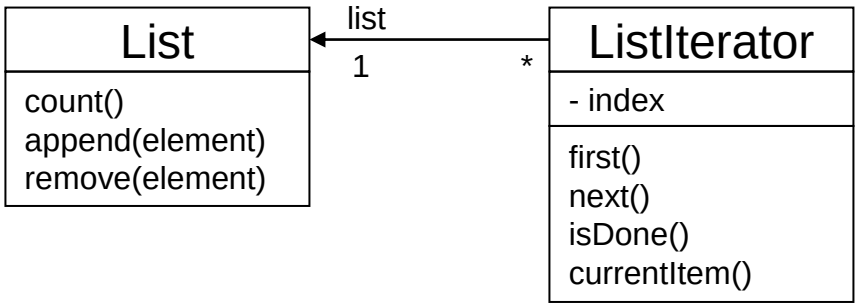
- The `java.util.Observer` interface
  - `void update (Observable o, Object arg)`
- The `java.util.Observable` class
  - `Observable()`
  - `void addObserver(Observer)`
  - `int countObservers()`
  - `void deleteObserver(Observer o)`
  - `void deleteObservers()`
  - `void notifyObservers()`
  - `void notifyObservers(Object arg)`
  - `boolean hasChanged()`
  - `void clearChanged()`
  - `void setChanged()`

Iterator

- **Purpose**
  - Providing sequential access (traversal) to the elements of a collection (aggregate), without exposing their internal representation
  - Also known as cursor
- **Motivation**
  - E.g.: A list should provide a mechanism to traverse its elements hiding their internal structure
  - The list can be traversed in several ways, but no more operations should be added for additional traversing strategies
  - We must be able to do several traversals simultaneously
- **Solution:**
  - Giving the list traversing responsibility to the iterator

Iterator

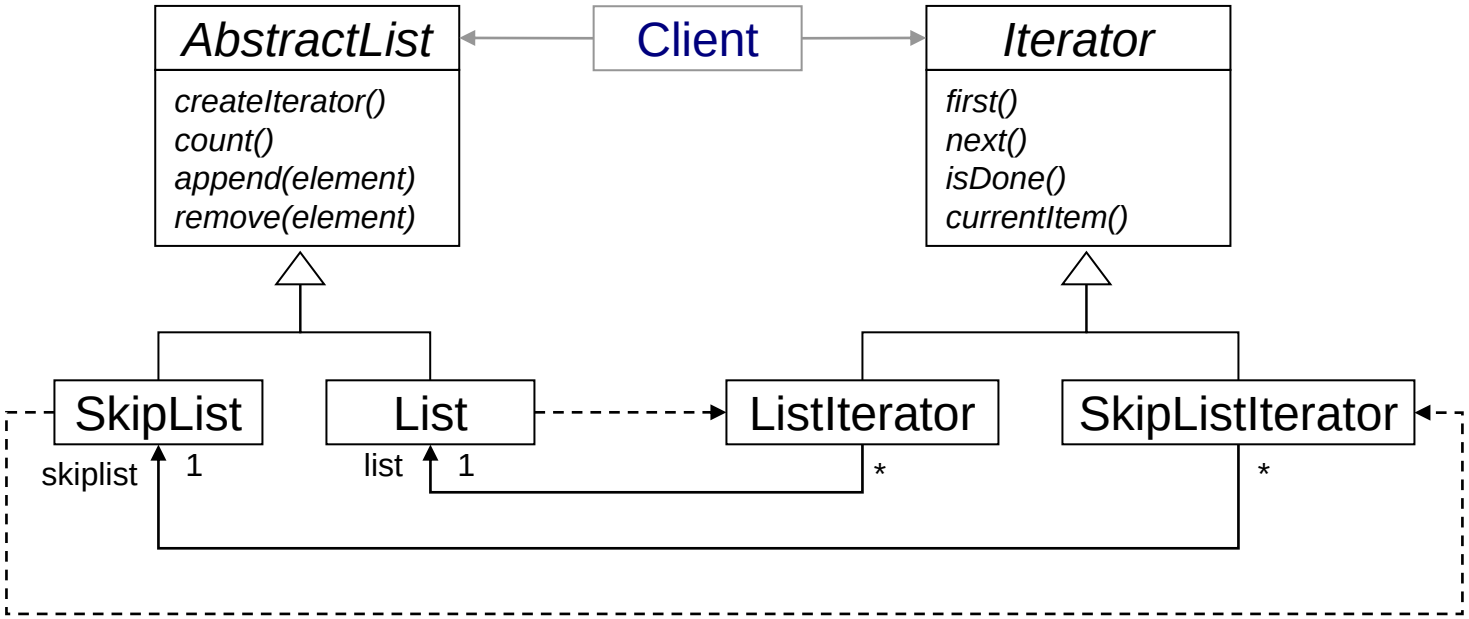
Motivation



- When instantiating *ListIterator*, the list has to be provided.
- Once the iterator has been instantiated, the elements of the list can be accessed.
- **Advantages**
  - Separates the traversing mechanism from the object, which allows defining iterators that follow different strategies, and performing several traversing processes at the same time.
- **Disadvantages**
  - A common interface for all iterators of a list is not ensured
  - Iterator and client are coupled. How do we know which iterator to use?
  - A common interface for list creation is not ensured

Iterator

Motivation

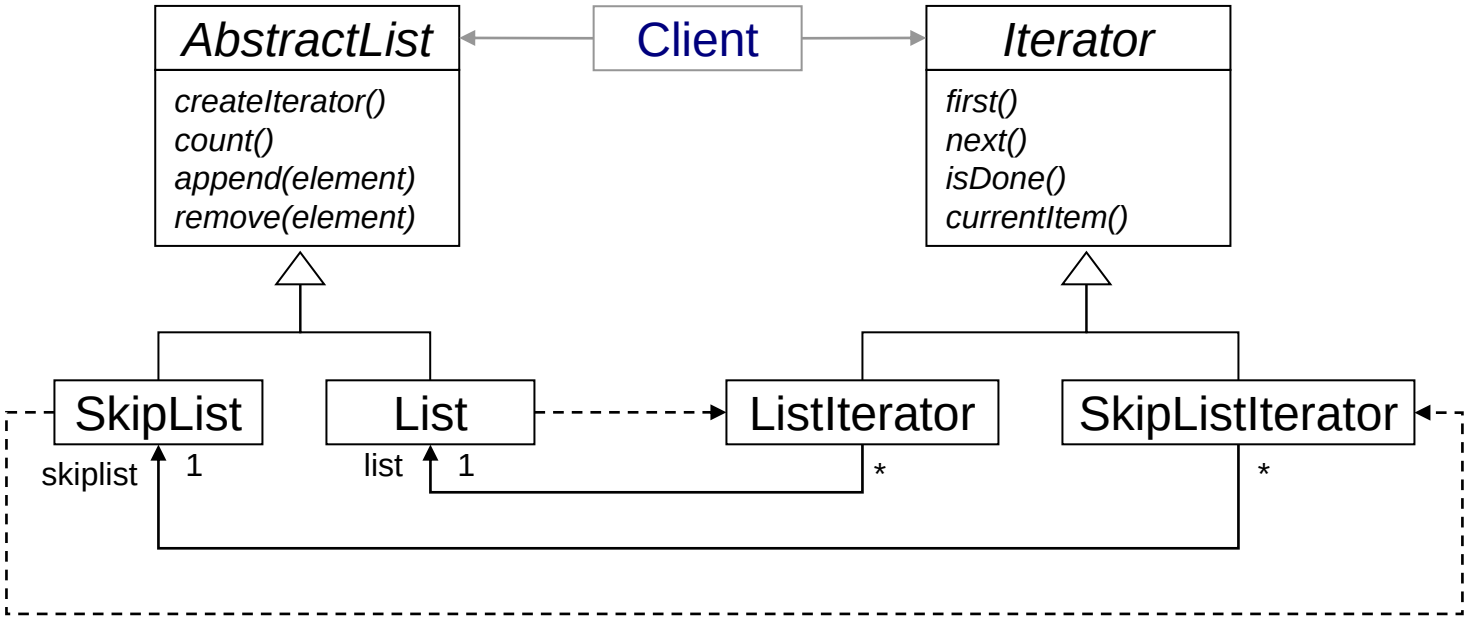


- Generalize the iterator to support polymorphic iteration
- The aggregate can be changed without modifying the client code
- The lists are responsible for creating their own iterators

Iterator

Motivation

Which other pattern are we using here?

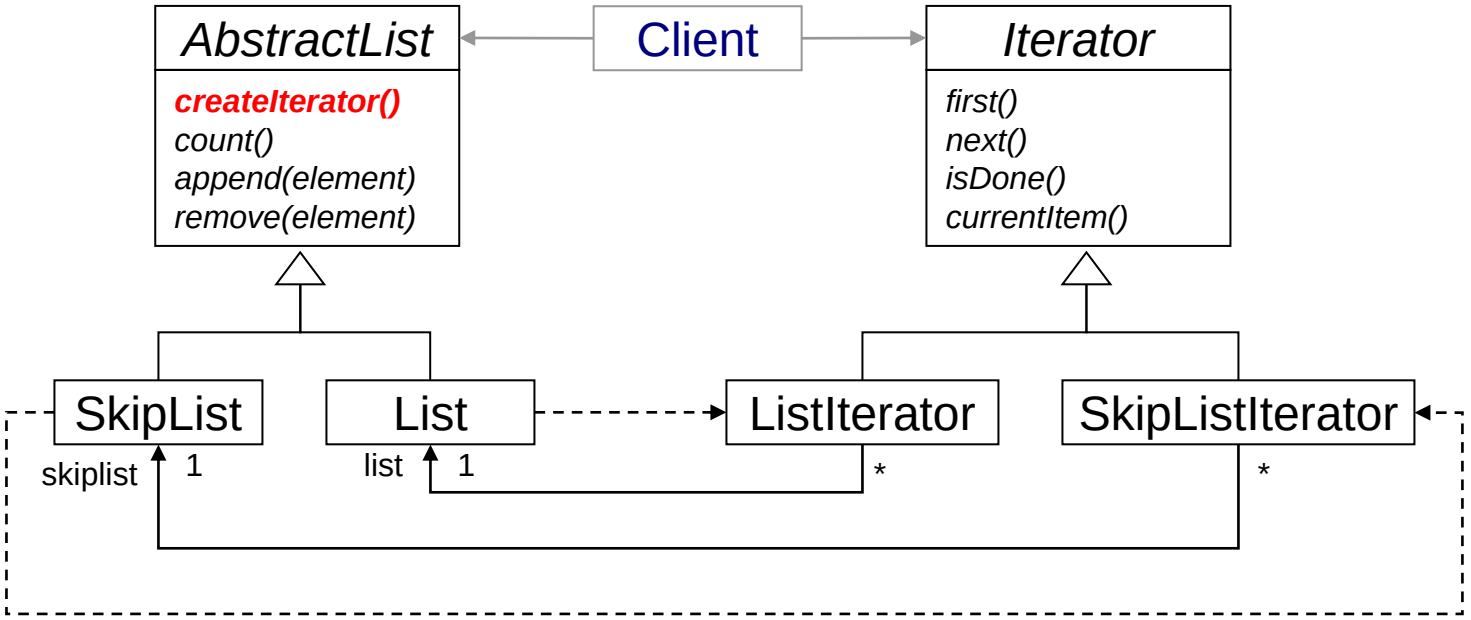


- Generalize the iterator to support polymorphic iteration
- The aggregate can be changed without modifying the client code
- The lists are responsible for creating their own iterators

Iterator

Motivation

Which other pattern are we using here?



- Generalize the iterator to support polymorphic iteration
- The aggregate can be changed without modifying the client code
- **The lists are responsible for creating their own iterators**

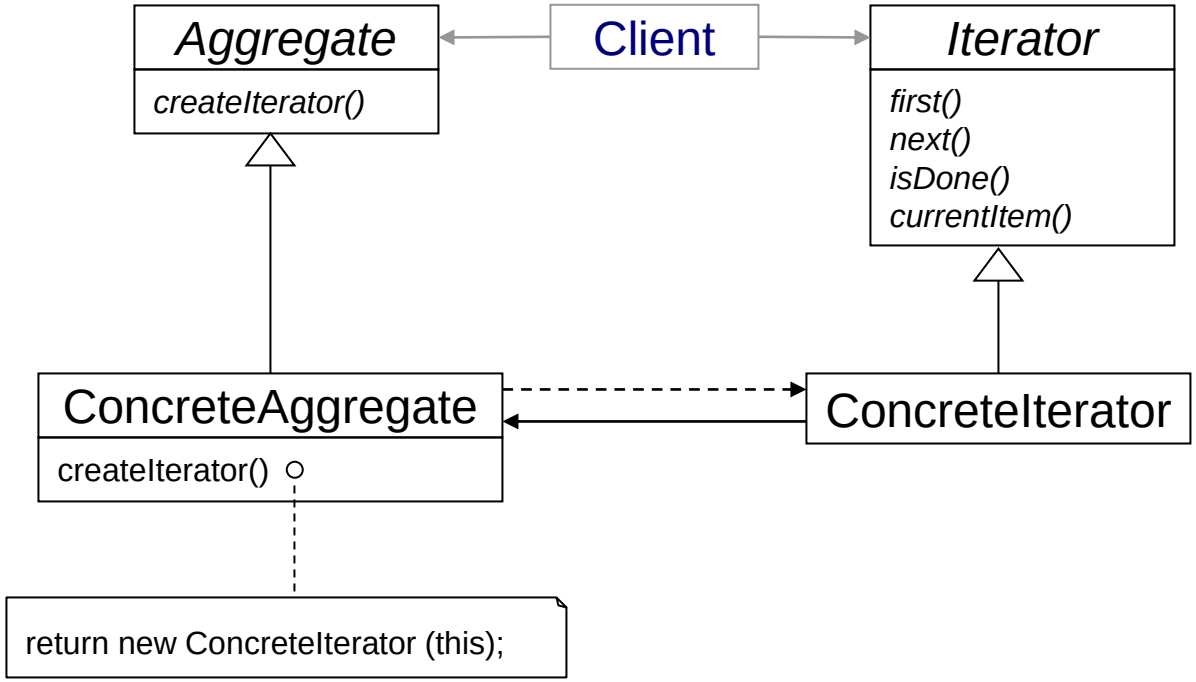
Iterator

Application

- Use the *Iterator* pattern to
  - Access the content of an aggregate without exposing its internal representation
  - Allow several traversal strategies on the aggregate
  - Provide a uniform interface to traverse different types of aggregates (that is, allow polymorphic iteration)

Iterator

Structure



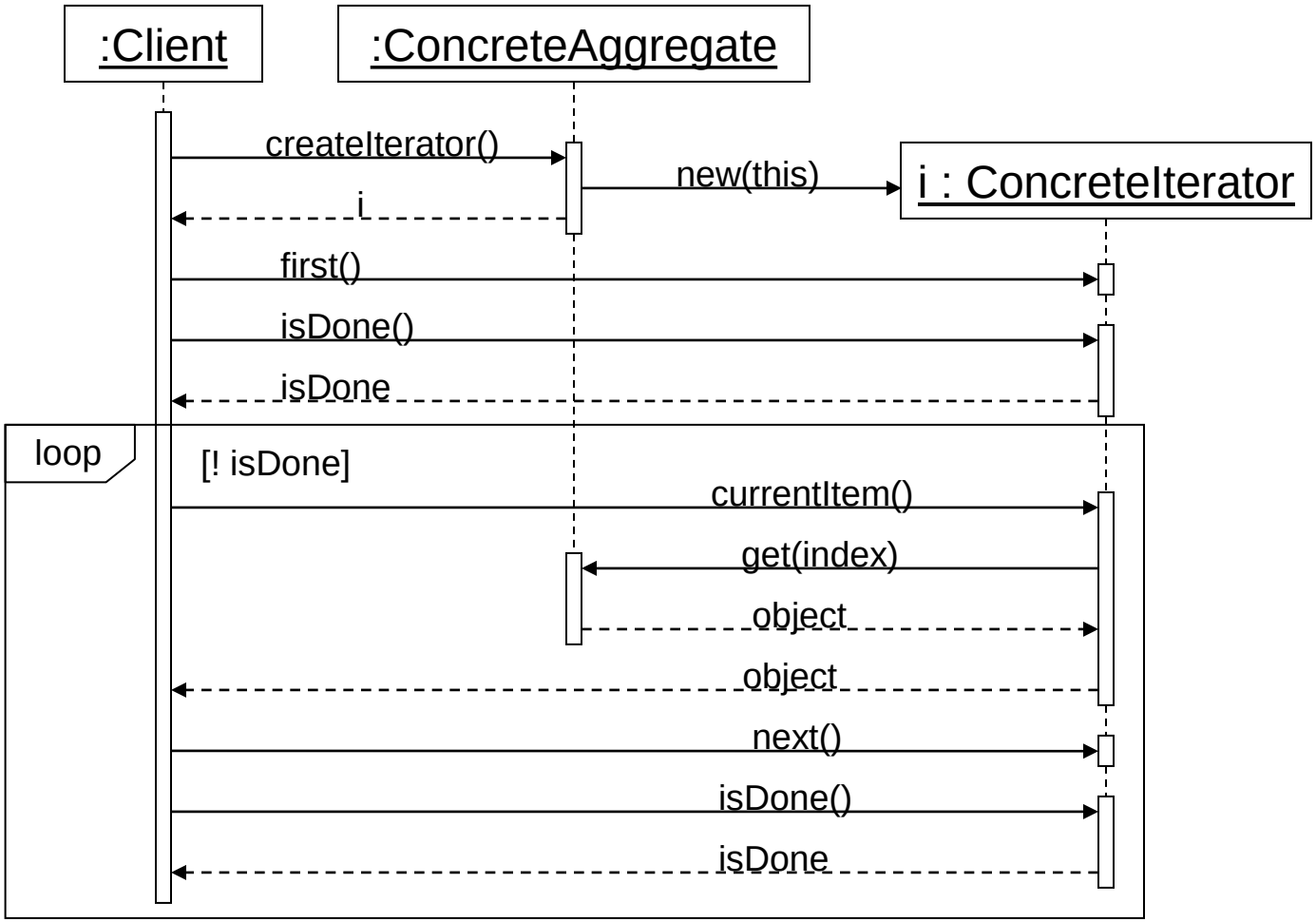
Iterator

Participants

- **Iterator**: defines the interface to access and traverse elements
- **ConcreteIterator**
  - Implements the *Iterator* interface
  - Maintains the current position on the traversal of the aggregate
- **Aggregate***:* it defines the interface to create an *Iterator* object
- **ConcreteAggregate***:* it implements the creation of the *Iterator* to return the appropriate *ConcreteIterator* instance

Iterator

Collaborations



Iterator

Code of the example

```
public interface Iterator {
   public void first();
   public void next();
   public boolean isDone();
   public Object currentItem();
}

public interface Aggregate {
   public Iterator createIterator();
   public Object get(int);
   public int count();
}

public class ListIterator implements Iterator{
   private Aggregate a;
   private int current;
   public ListIterator (Aggregate a); {
      this.a = a;
      current = 0;
   }
   public void first() { current = 0; }
   public void next() { current++; }
   public boolean isDone() { return current >= a.count(); }
   public Object currentItem() { return a.get(current); }
}
```

Iterator

Implementación

- Who controls the iteration?
  - The client: external iterator
    - More flexible: it allows comparing two collections:

```
Iterator it = list.createIterator();
it.first();
while (it.isDone() == false) {
   it.next();
   it.currentItem();
}
```

  - The iterator: internal iterator
    - The iterator receives a function to apply over the aggregate elements, and the traversal is automatic
    - The client code is simplified

```
Iterator it = list.createIterator(OPERATION);
it.traverse();
```

Iterator

In Java…

- Java Collection framework
  - **Aggregate**
    - Interfaces `Collection`, `Set`, `SortedSet`, `List`, `Queue` of `java.util`
    - A method `iterator()`, which returns a generic iterator
  - **ConcreteAggregate**: implementations of the above interfaces
    - Set is implemented by `HashSet`, `TreeSet`, `LinkedHashSet` classes
    - List is implemented by `ArrayList`, `LinkedList` classes
  - **Iterator**: interface `java.util.Iterator`
    - `boolean hasNext()`
    - `Object next()`
    - `void remove()`
  - **ConcreteIterator**: concrete implementations of `Iterator`
  - **Client example** :

```
java.util.Collection c = new java.util.LinkedList();
java.util.Iterator it = c.iterator();
while (it.hasNext()) {
        it.next();
}
```

Conclusions

- The design patterns describe the solution to problems that repeatedly appear in our systems, so that such solution can be used whenever it is needed
- They capture knowledge from design experts
- They help on the generation of “malleable” software, that is, software that supports and facilitates its change, reuse and improvement
- They are design guidelines, not strict rules

Bibliography

- “UML distilled: a brief guide to the standard Object Modelling Language”, Martin Fowler. Editorial Addison-Wesley.
- “Design patterns: elements of reusable object oriented software”, Erich Gamma et al. Editorial Addison- Wesley.
- “Design patterns in Java: a catalog of reusable design patterns illustrated with UML”, Mark Grand. Editorial John Wiley & sons.