Design Patterns

GL52

Yazan MUALLA

CIAD, Univ. Bourgogne Franche-Comté, UTBM, France

26 May 2020

Contents

Object-Oriented Modelling

- Object-oriented modelling methods emphasize the **design notation**
- But it is more than just drawing diagrams, it is about designing the best solution
- Most powerful reuse is the **design reuse** (match problem to design experience)
- Object-oriented modelling systems exhibit *recurring design structures* that promote:
  - Abstraction
  - Flexibility
  - Modularity
  - Elegance
  - Efficiency
  - …

Beginning of Patterns

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over."

Christopher Alexander, A Pattern Language, 1977

Why Study Patterns?

- Reuse tried and proven solutions
  - No need to reinvent the wheel
  - Provide a head start
  - Save time
  - Increase *efficiency*
- Establish common terminology
  - Increase inter-relations benefits in shared work
  - Avoid unanticipated things later
- Provide a higher level prospective
  - Free us from dealing with the details too early

Why Study Patterns? (cont.)

- Most design patterns make software systems more *modifiable*
  - We are using time-tested solutions
- Make software systems easier to update (more *maintainable*)
- Help increase the understanding of basic object-oriented design principles
  - *Encapsulation*
  - *Inheritance*
  - *Interfaces*
  - *Polymorphism*
  - *…*

Contents

Design Patterns

- A technique to repeat the designer success and intensify the design experience
- **(Problem, Solution) pair** that abstracts a recurring design structure
- Name & specify the design structure explicitly
- Four Basic Parts:
  - Name
  - Problem
  - Solution
  - Uses & trade-offs

Design Patterns Goals

- Codify good design
  - Generalized experience
  - Aid to novices & experts alike
- Give design structures explicit names
  - Common vocabulary
  - Reduced *complexity*
  - Greater *expressiveness*
  - Improve documentation
- Facilitate *refactoring*
  - Patterns are interrelated
  - Additional *flexibility*

Refactoring

- The **process of restructuring the code** in a series of small and semantics-preserving transformations without changing its external behavior

- It improves *nonfunctional* attributes of the software:
  - Improve source-code *readability* and reduce *complexity*
  - Improve source-code *maintainability*
  - Create a more expressive internal architecture or object model to improve *extensibility*

- **Extreme Programming** is a form of Agile Programming that emphasizes refactoring

- **Unit testing** is an essential component of Extreme Programming, and it is testing classes in isolation

Object-Oriented Design Patterns

- **Creational patterns:** deal with the process of object creation

• Abstract Factory • Builder	• Factory Method • Prototype	• Singleton

- **Structural patterns:** deal primarily with the static composition and structure of classes and objects

• Adapter • Bridge • Composite	• Decorator • Façade	• Flyweight • Proxy

- **Behavioral patterns:** deal primarily with the dynamic interaction among classes and objects

• Chain of Responsibility • Command • Interpreter • Iterator	• Mediator • Memento • Observer • State	• Strategy • Template Method • Visitor

Object-Oriented Design Patterns (cont.)

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Contents

Observer (object behavioral)

- **Problem:** Need to notify a changing number of objects (observers) about an update in an entity (subject)

- **Solution:** Define a **one-to-many dependency** so that when one entity updates its state, all its dependents (observers) are notified automatically

- **Uses:**
  - An abstraction has two aspects, one dependent on the other
  - A change to one entity requires informing others

- **Consequences:**
  - \+ *Modularity*: subject & observers may vary independently
  - \+ *Extensibility*: can define & add any number of observers
  - \+ *Customizability*: different observers offer different views of the subject
  - – Unexpected updates: observers don't know about each other

Observer (object behavioral)

- Whenever the subject undergoes a change in its state, it notifies all of its registered observers
- Upon receiving **notification** from the subject, each of the observers queries the subject to **synchronize** its state with that of the subject's
- The **subject** should provide an **interface for registering/unregistering and for change notifications**
- **Observers** should provide an **interface for receiving notifications** from the subject

Observer (object behavioral)

- Publisher-subscriber, one of the following two must be true:
  - In the **pull model**: The subject should provide an interface that enables observers to query the subject for the required state information to update their state
  - In the **push model**: The subject should send the state information that the observers may be interested in

- After applying the Observer pattern, **different observers can be added dynamically** without requiring any changes to the Subject class

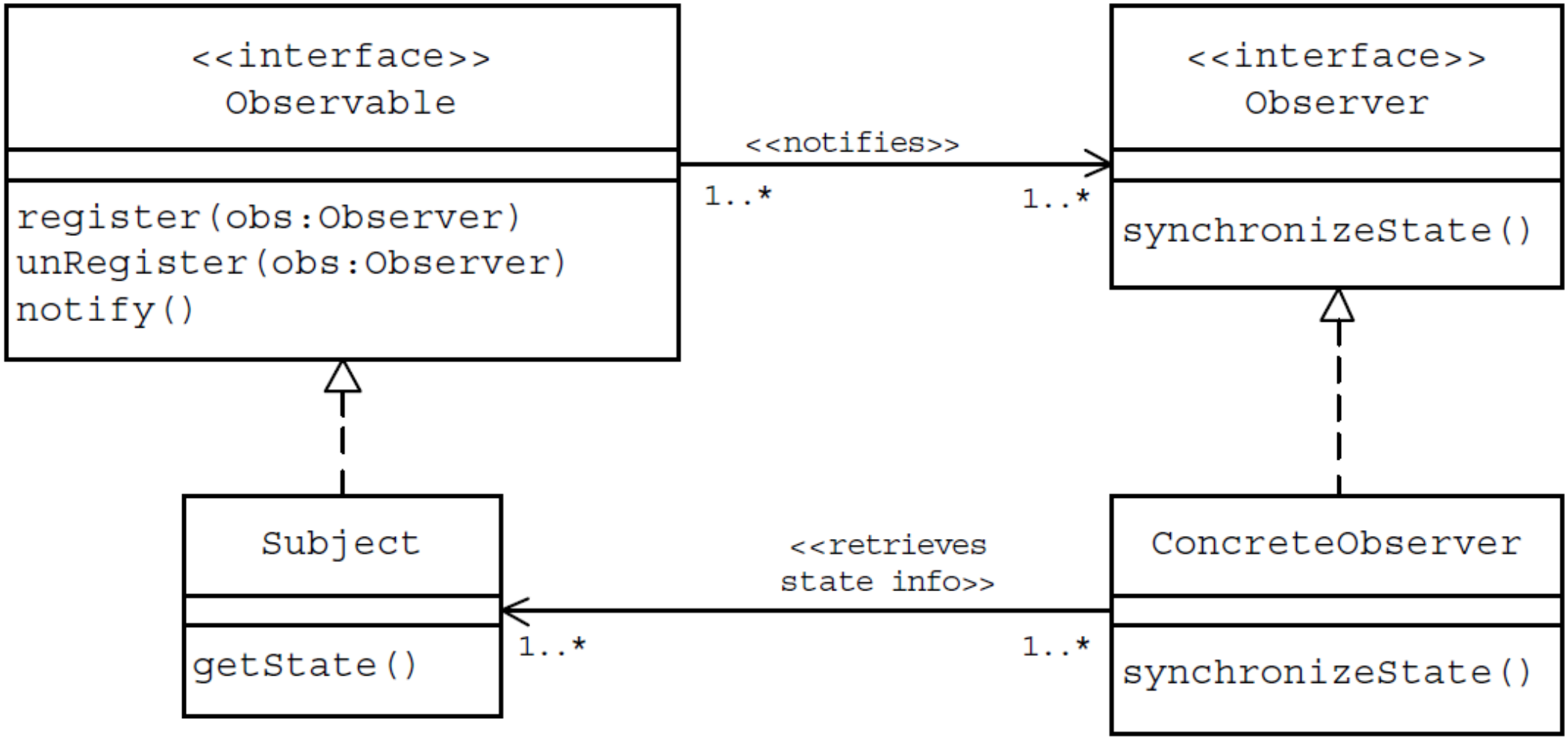
Observer (object behavioral)

- **Implementation:**
  - Subject-observer mapping
  - Update protocols: the push & pull models
  - Register modifications of interest explicitly

- **Examples:**
  - Smalltalk Model-View-Controller (MVC)
  - Pub/sub middleware (e.g., CORBA Notification Service, Java Messaging Service)
  - Mailing lists
  - Social media followers

Detailed Discussion and Examples

Observer (object behavioral)



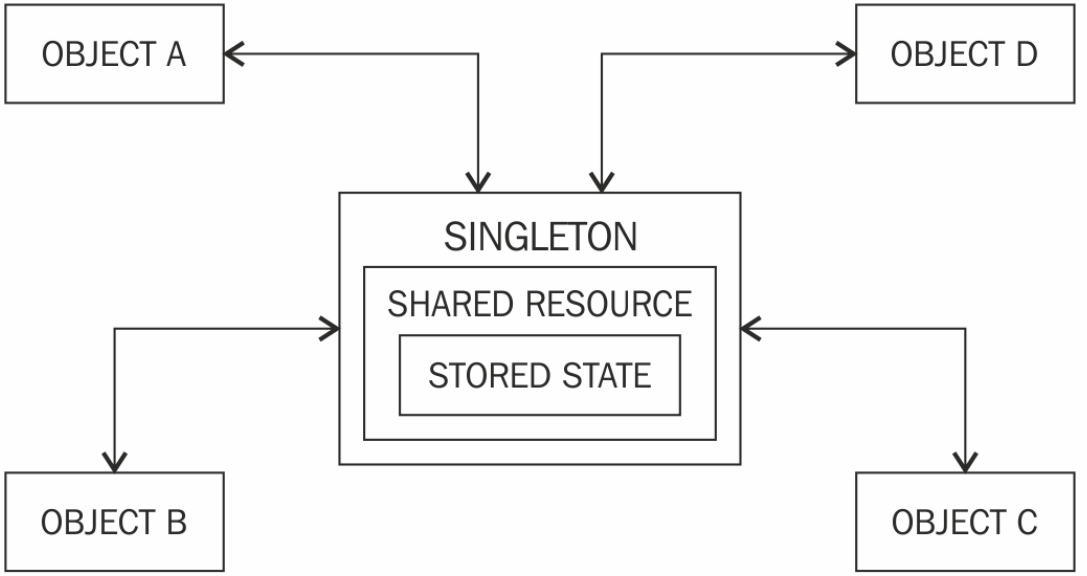
Contents

Singleton (object creational)

- **Problem:** Need to have **one and only one instance of a given class** during the lifetime of an application

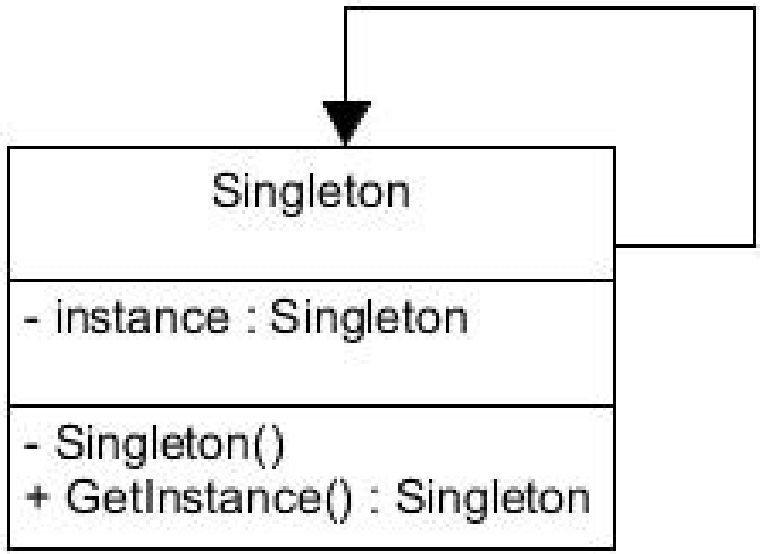
- **Solution:** Provides a **controlled object creation mechanism** to ensure that only one instance of a given class exists

- **Examples:**
  - AudioStream
  - A single database connection object



Singleton (object creational)

```
class Singleton {
    private static Singleton instance = new Singleton();
    // don't let Java give you a default public
constructor
    private Singleton() { }

    Singleton GetInstance() {
        return instance;
    }
    ...
}
```



Contents

Composite (object structural)

- **Problem:** Need to treat individual objects & multiple, recursively-composed objects uniformly

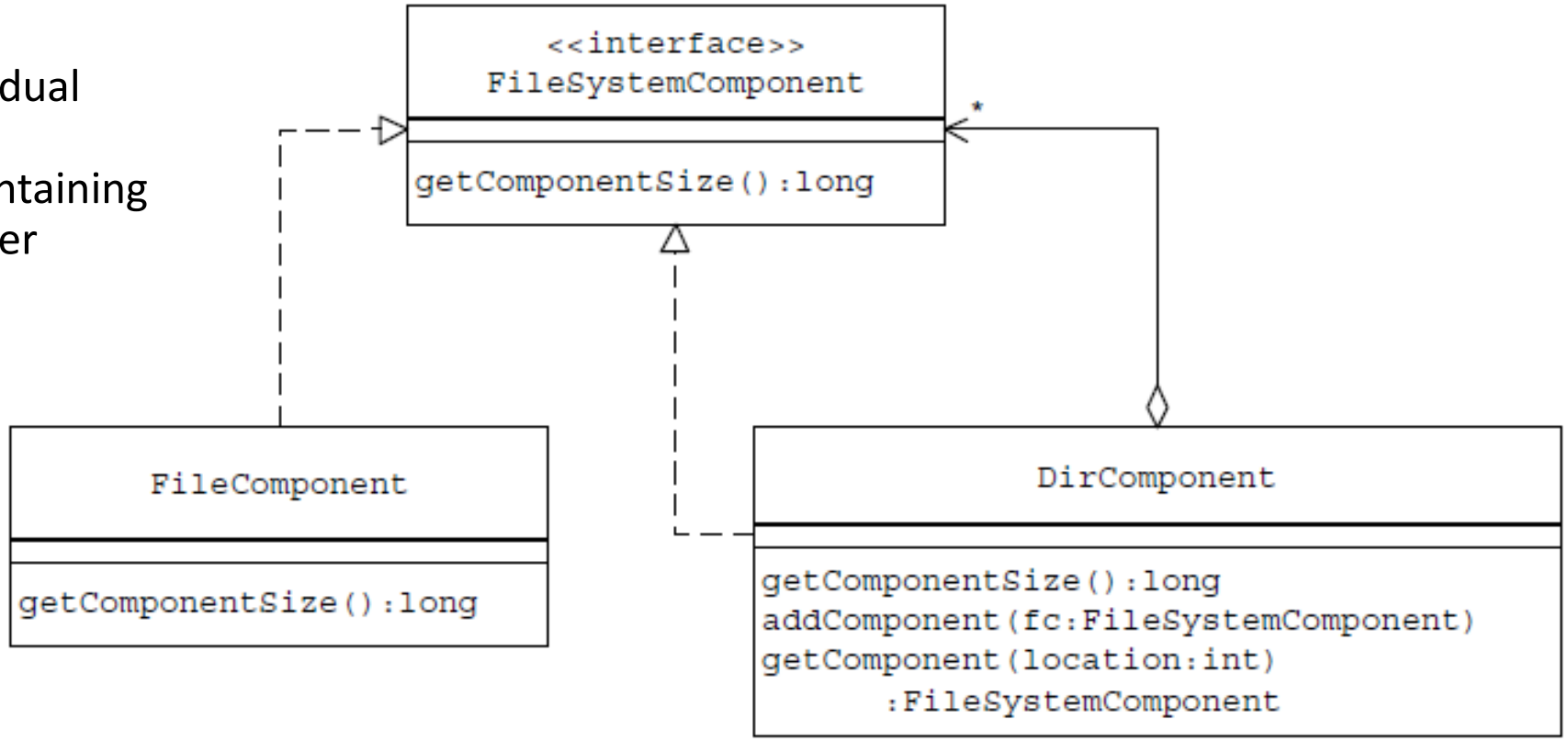
- **Solution:** The object **inherits** the abstract class or **realizes** the interface that is composed of or aggregates the object

- **Uses:**
  - No distinction between individual & composed elements
  - Objects in structure can be treated uniformly

- **Consequences:**
  - \+ *Uniformity*: treat components the same regardless of complexity
  - \+ *Extensibility*: new component subclasses work wherever old ones do
  - _ Ambiguity

Composite (object structural)

- You want to treat **uniformly**
  - files (individual objects)
  - folders (containing files or other folders)



Composite (object structural)

- **Implementation:**
  - Do components (children) know their composites (parents)?
  - Uniform interface for both components & composites?
  - Responsibility for deleting children

- **Examples:**
  - Directory structures in UNIX & Windows
  - Naming Contexts in CORBA
  - MIME types in SOAP

Contents

Template Method (class behavioral)

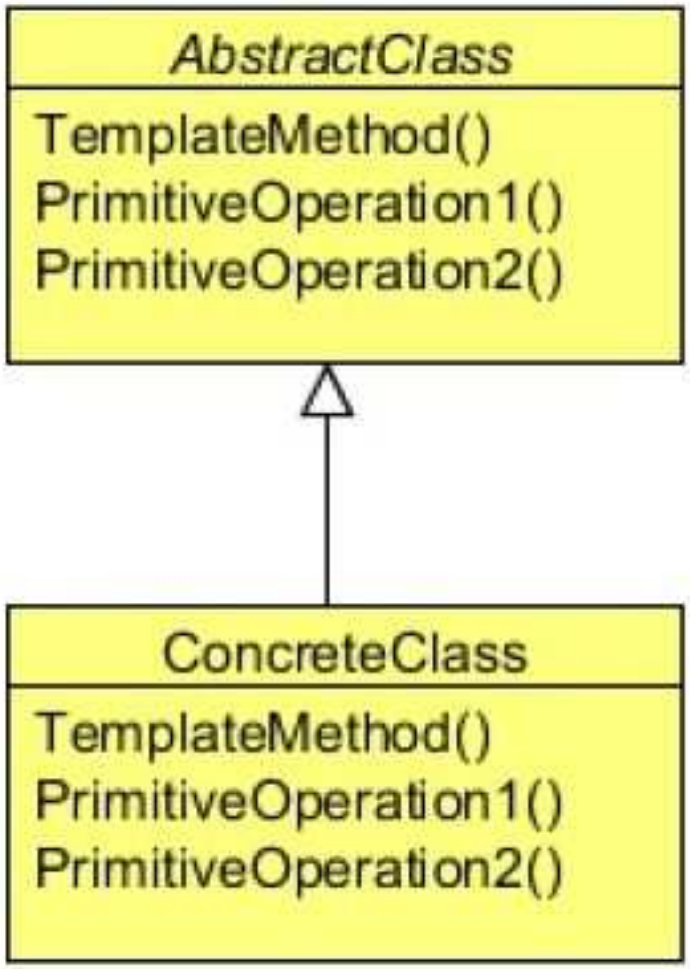
- **Problem:** Having similar implementations of the same functionality with some variant parts

- **Solution:** Provide an **abstract class** that defines way(s)/template(s) to execute its methods. Its subclasses can override the method implementation as per need but the invocation is to be in the same way as defined by the abstract class

- **Uses:**
  - When there is an algorithm that could be implemented in multiple ways, the template method pattern enables keeping the outline of the algorithm in a separate method (Template Method) inside a class (Template Class), leaving out the specific implementations of this algorithm to different subclasses
  - Keep the invariant part of the functionality in one place and allow the subclasses to provide the implementation of the variant part

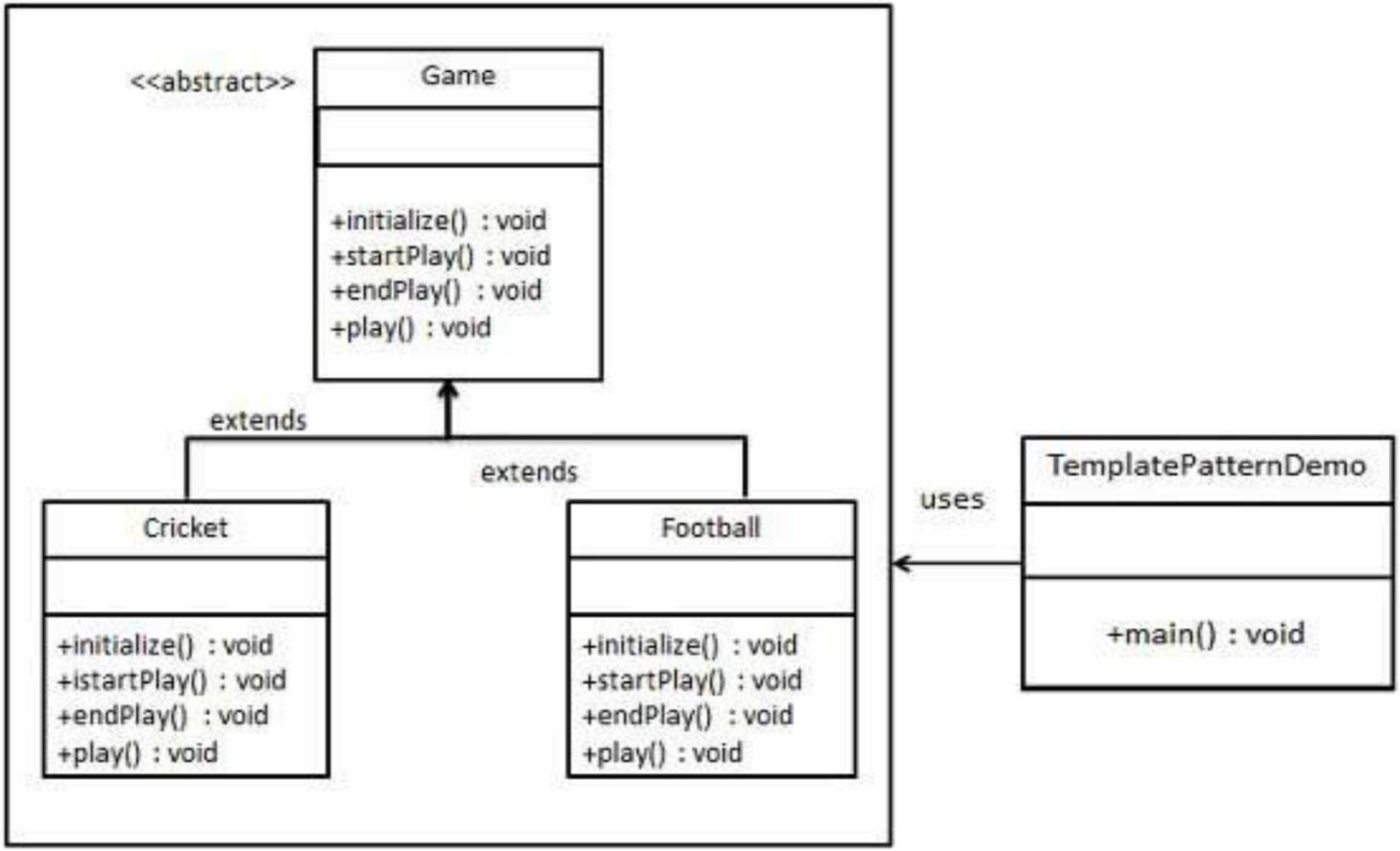
Template Method (class behavioral)

- Defines abstract primitive operations that concrete subclasses use to implement steps of an algorithm
- Implements a template method defining the skeleton of an algorithm.



Detailed Discussion and Examples

Template Method (class behavioral)



Contents

Factory Method (class creational)

- **Problem:** Need to **reduce coupling** between client objects and class hierarchy structure

- **Solution:** Define an interface to choose the right class from the hierarchy structure as per some criteria

- **Uses:**
  - A class can't anticipate the class of objects it must create
  - A class wants its subclasses to specify the objects it creates
  - Classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate

Factory Method (class creational)

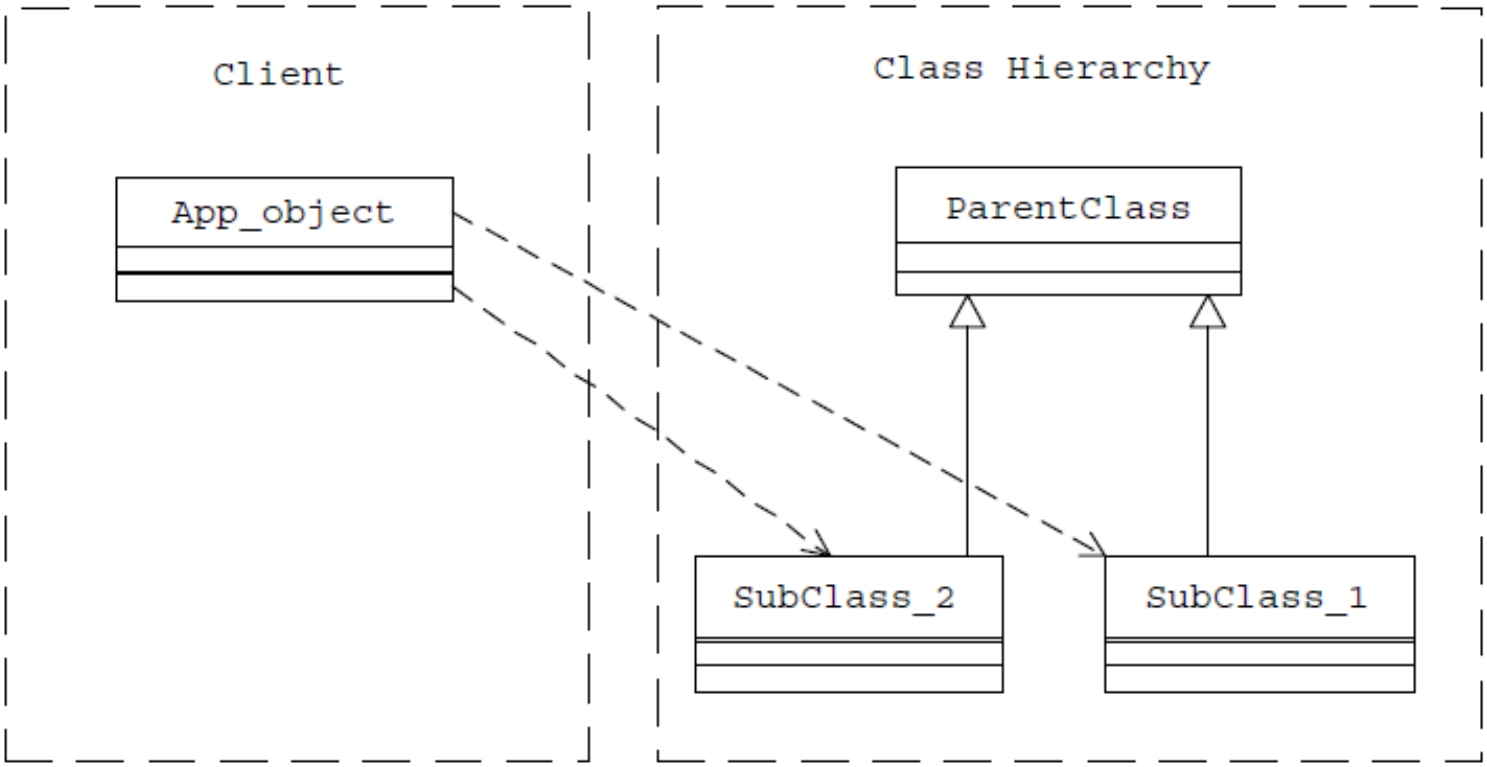
- **Examples:**
  - XML files handlers
  - Image creator:

```
Image createImage (String ext) {

    if (ext.equals("gif")) return new GIFImage();

    if (ext.equals("jpg")) return new JPEGImage();

    ...

}
```

Image example drawbacks: High coupling, Dependency, Readability, Scalability

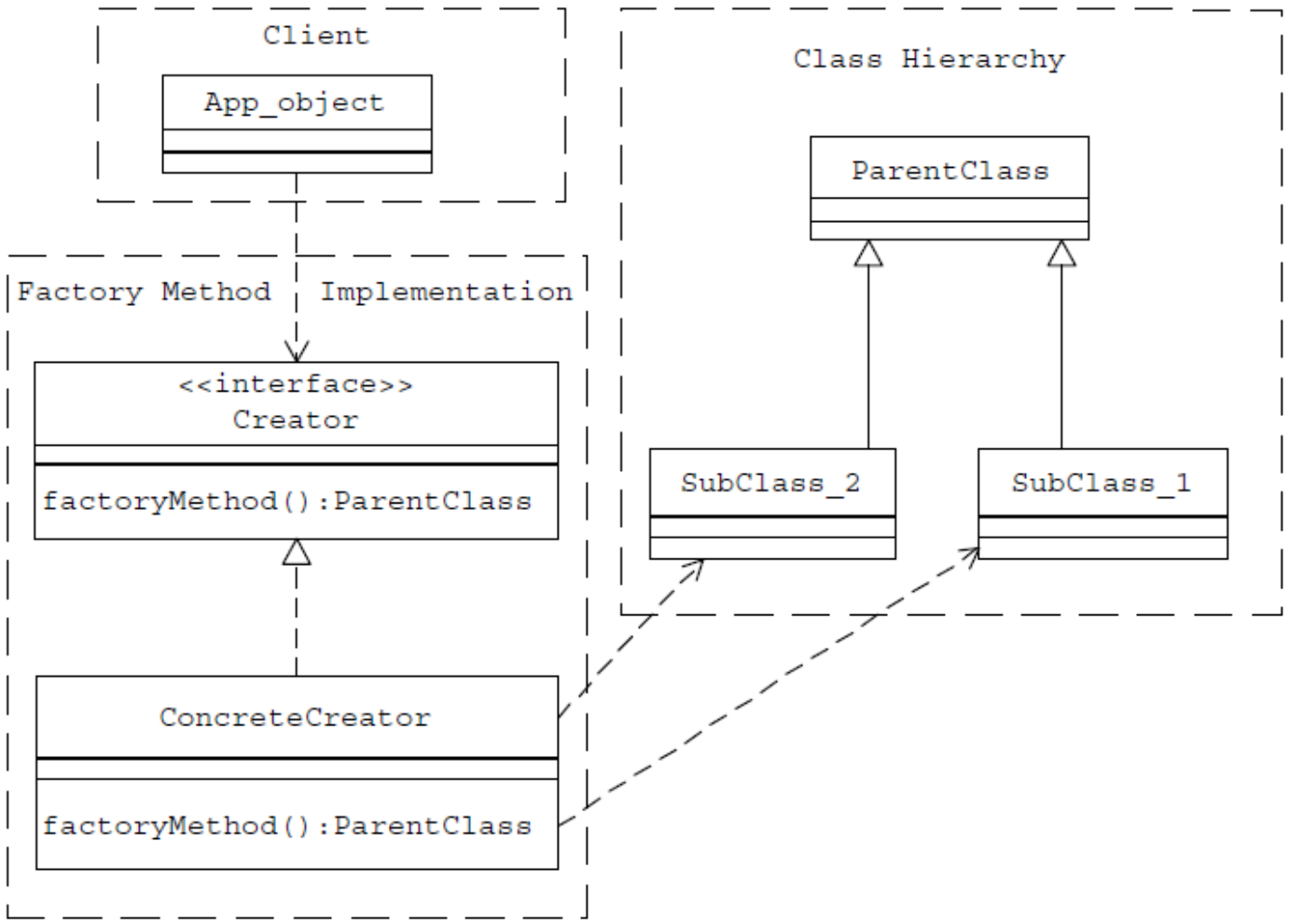
Factory Method (class creational)

- Problem



Factory Method (class creational)

- Solution



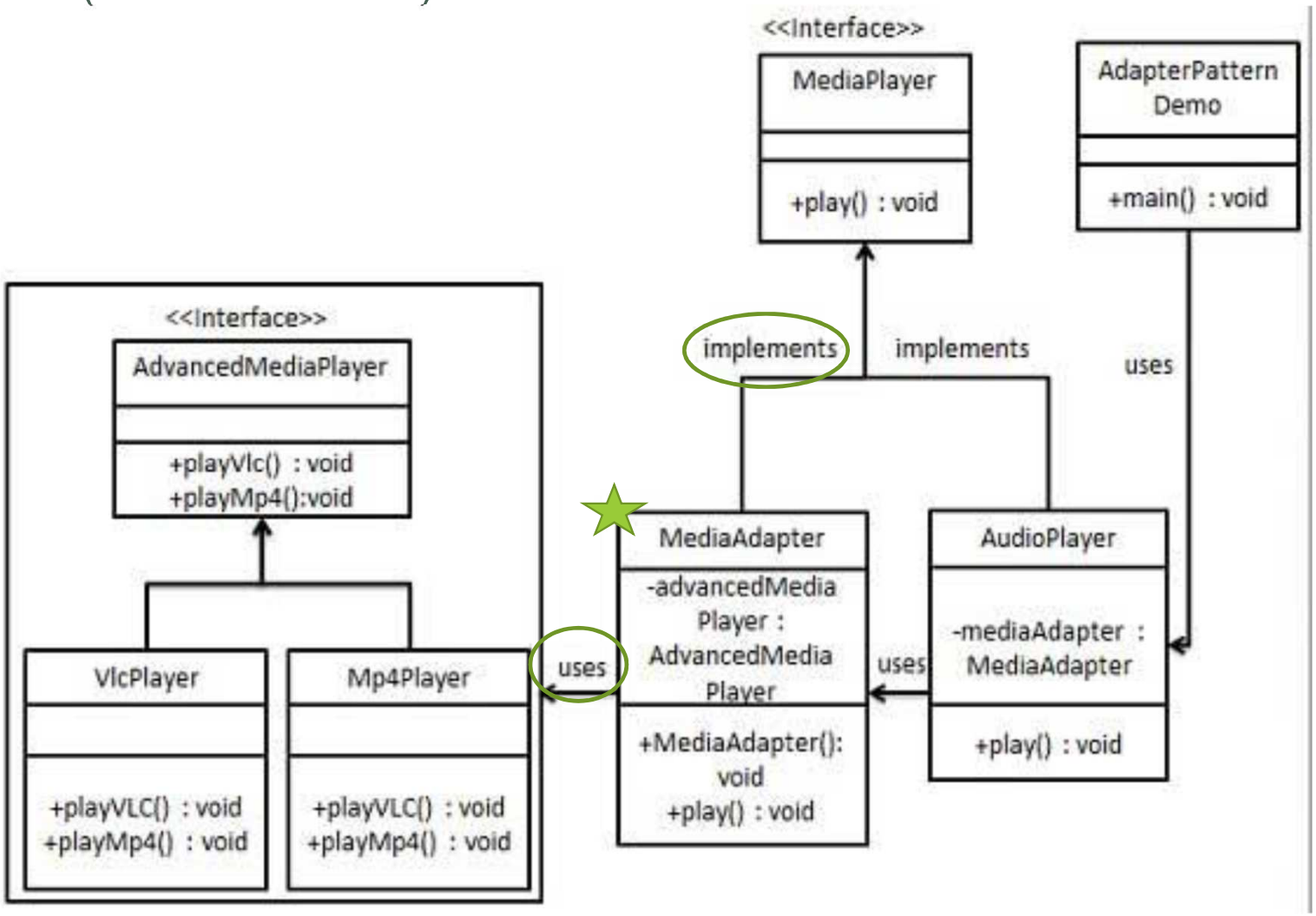
Contents

Adapter (class structural)

- **Problem:** Need to fill the gap between two interfaces or **connect legacy code with new code** with no changes to be made between the two parts
- **Solution:** Define a bridge between the two parts
- **Uses:**
  - This pattern works as a bridge between **two independent and incompatible entities**
  - This pattern involves a single class which is responsible to join functionalities of entities
- **Examples:**
  - Card reader
  - Audio player adapter

Detailed Discussion and Examples

Adapter (class structural)



Contents

Design Patterns

- Design patterns are applicable in all stages of the OO lifecycle
  - Analysis, design, & reviews
  - Realization & documentation
  - Reuse & refactoring
- Design patterns permit design at a more abstract level
  - Treat many class/object interactions as a unit
  - Often beneficial after initial design
  - Target for class refactoring
- Variation-oriented design
  - Consider what design aspects are variable
  - Identify applicable pattern(s)
  - Vary patterns to evaluate tradeoffs
  - Repeat

Object-Oriented Design Patterns with examples given

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Benefits of Design Patterns

- Design reuse
- Uniform design vocabulary/terminology
- Enhance understanding, restructuring, & team communication
- Basis for automation
- Abstract away many unimportant details

Liabilities of Design Patterns

- Require significant, tedious & error-prone human efforts to handcraft the reuse of implementation design
- Can be deceptively simple uniform design vocabulary
- May limit design options
- Leave some important details unresolved

References

- Books
  - **Design Patterns - Elements Of Reusable Object-Oriented Software**
  - Timeless Way of Building, Alexander, ISBN 0-19-502402-8
  - A Pattern Language, Alexander, 0-19-501-919-9
  - Design Patterns, Gamma, et al., 0-201-63361-2 CD version 0-201-63498-8
  - Pattern-Oriented Software Architecture, Vol. 5, Buschmann, et al., 0-471-48648-5
  - Analysis Patterns, Fowler; 0-201-89542-0
  - Concurrent Programming in Java, 2nd ed., Lea, 0-201-31009-0
  - Pattern Languages of Program Design
    Vol. 1, Coplien, et al., eds., ISBN 0-201-60734-4
    Vol. 2, Vlissides, et al., eds., 0-201-89527-7
    Vol. 3, Martin, et al., eds., 0-201-31011-2
    Vol. 4, Harrison, et al., eds., 0-201-43304-4
    Vol. 5, Manolescu, et al., eds., 0-321-32194-4
  - AntiPatterns, Brown, et al., 0-471-19713-0
  - Applying UML & Patterns, 2nd ed., Larman, 0-13-092569-1
  - Pattern Hatching, Vlissides, 0-201-43293-5
  - Design Patterns Explained, Shalloway & Trott, 0-201-71594-5

References

- Articles
  - “Object-Oriented Patterns,” P. Coad; Comm. of the ACM, 9/92
  - “Documenting Frameworks using Patterns,” R. Johnson; OOPSLA ’92
  - “Design Patterns: Abstraction & Reuse of Object-Oriented Design,” Gamma, Helm, Johnson, Vlissides, ECOOP ’93

Thank you for Your attention