Ingegneria del Software
Corso di Laurea in Informatica per in Management

Design Patterns
part 4

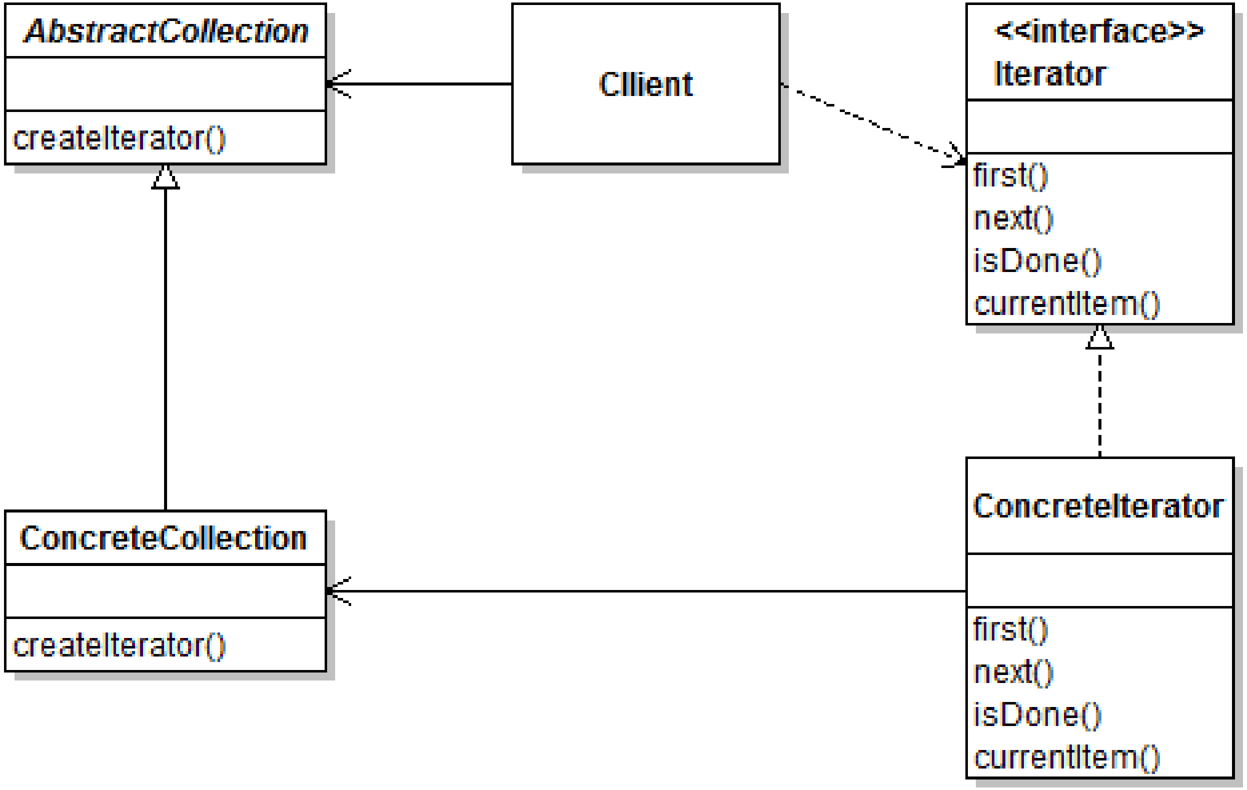
Davide Rossi
Dipartimento di Informatica – Scienze e Ingegneria
Università di Bologna

GoF: Memento

- Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
- A caretaker asks originator for mementos that can be stored and used to restore originator’s state.

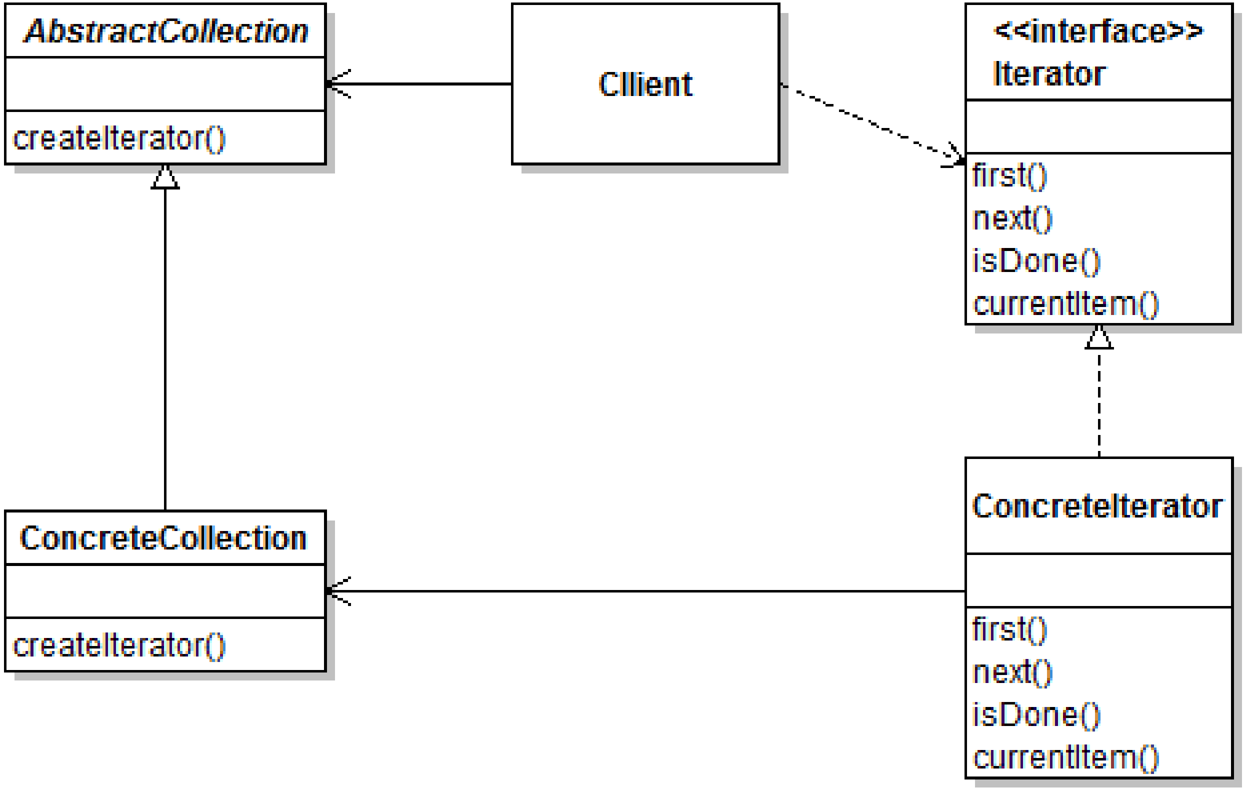
GoF: Iterator

- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.



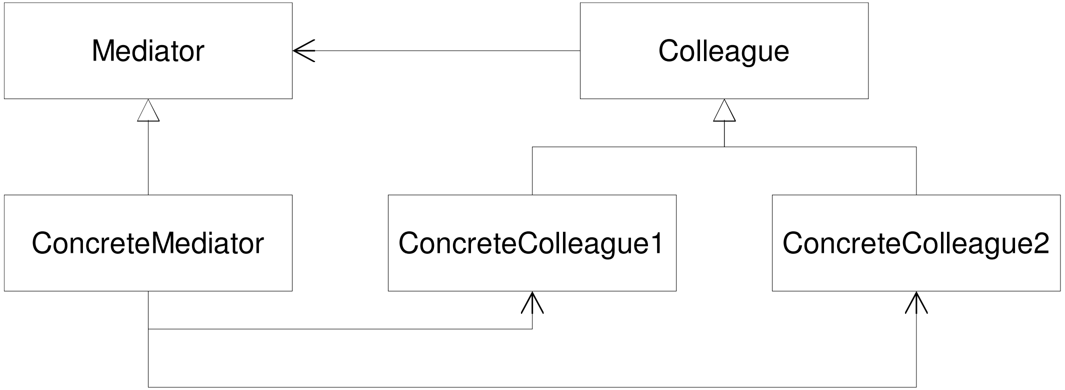
GoF: Iterator

- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.



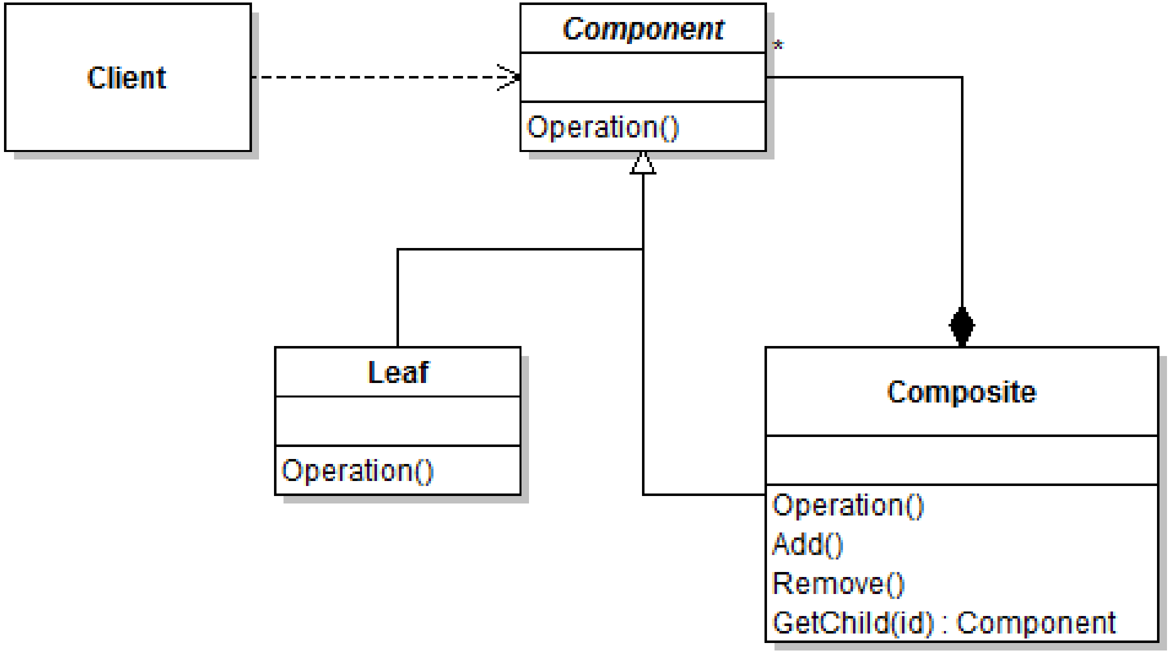
GoF: Mediator

- Define an object that encapsulates how a set of objects interact.
- Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and
- It lets you vary their interaction independently.



GoF: Composite

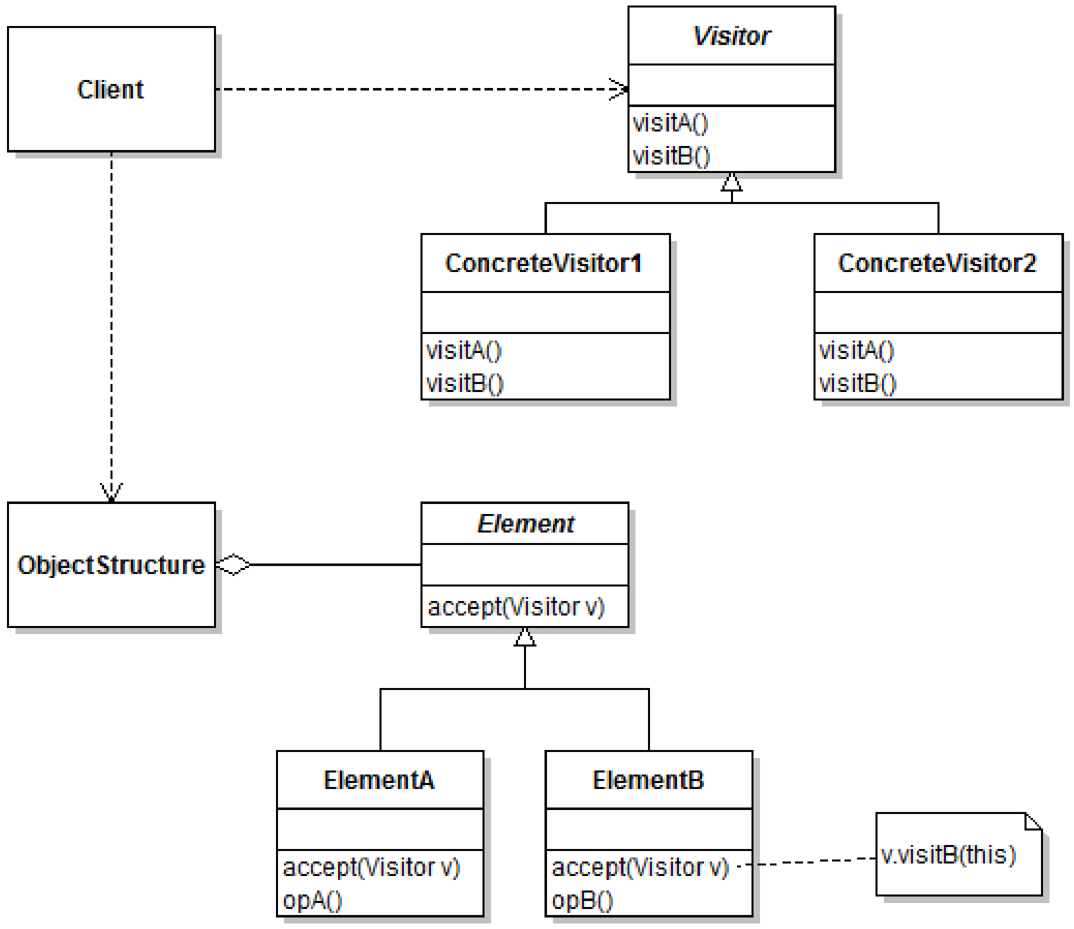
- Compose objects into tree structures to represent part-whole hierarchies
- Composite lets clients treat individual objects and compositions of objects uniformly



GoF: Visitor

- Represent an operation to be performed on the elements of an object structure.
- Visitor lets you define a new operation without changing the classes of the elements on which it operates.
- Based on inversion of control

Visitor


Client
Visitor
visitA()
visitB()
ConcreteVisitor1
visitA()
visitB()
ConcreteVisitor2
visitA()
visitB()
ObjectStructure
Element
accept(Visitor v)
ElementA
accept(Visitor v)
opA()
ElementB
accept(Visitor v)
opB()
v.visitB(this)

Visitor in the Java API

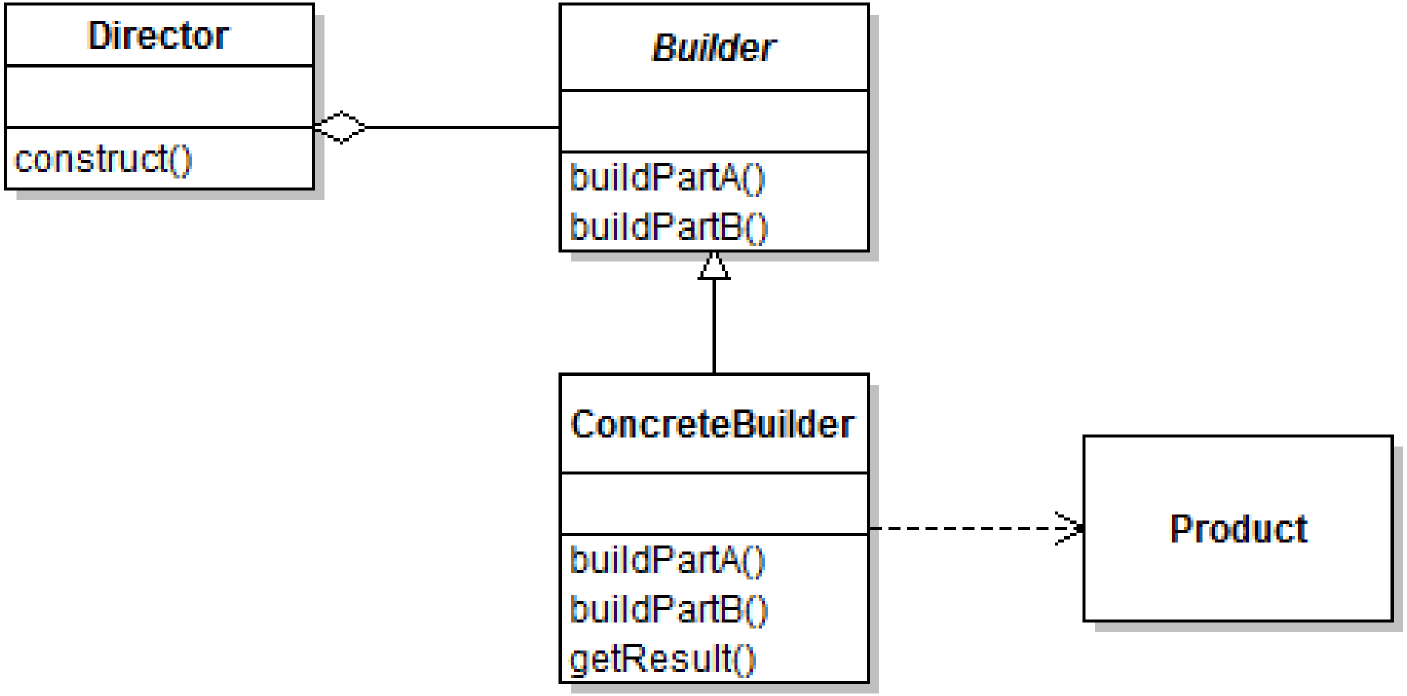
```
class java.nio.file.Files {
  public static Path walkFileTree(
          Path start,
        FileVisitor<? Super Path> visitor)
    …
}

interface java.nio.file.FileVisitor {
  … visitFile(T file, BasicFileAttributes attrs)
  …

}
```

GoF: Builder

- Separate the construction of a complex object from its representation so that the same construction process can create different representations.



Minor but frequent issue solved by Builder: ugly constructors

```
Foo foo = new Foo(a, b, null, null, c, null, d)
```

Builder in Java

```
Foo foo =
  Foo.builder().
  setWidth(a).setHeight(b).
  setDepth(c).setColor(d).build()
```

GoF: Command

- Encapsulate a request as an object, thereby letting you parametrize clients with different requests, queue or log requests, and support undoable operations.

GoF: Abstract Factory

- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

GoF: Prototype

- Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- Create new instances by *cloning* existing ones.

GoF: Flyweight

- Use sharing to support large numbers of fine-grained objects efficiently.

GoF: Chain of Responsibility

- Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request.
- Chain the receiving objects and pass the request along the chain until an object handles it.

GoF: Interpreter

- Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Resources

Books

- Eric Freeman & Elisabeth Robson, *Head First Design Patterns: Building Extensible and Maintainable Object-Oriented Software* (2nd Edition), O'Reilly

Online:

- http://www.vincehuston.org/dp/
- http://www.oodesign.com/
- https://refactoring.guru/design-patterns/
- http://www.informit.com/articles/article.aspx?p=1404056

Modern patterns

The Hollywood principle

- “Don't call us, we'll call you”
- Who controls who?
- Library or framework?
- IoC: inversion of control

Dependency Injection

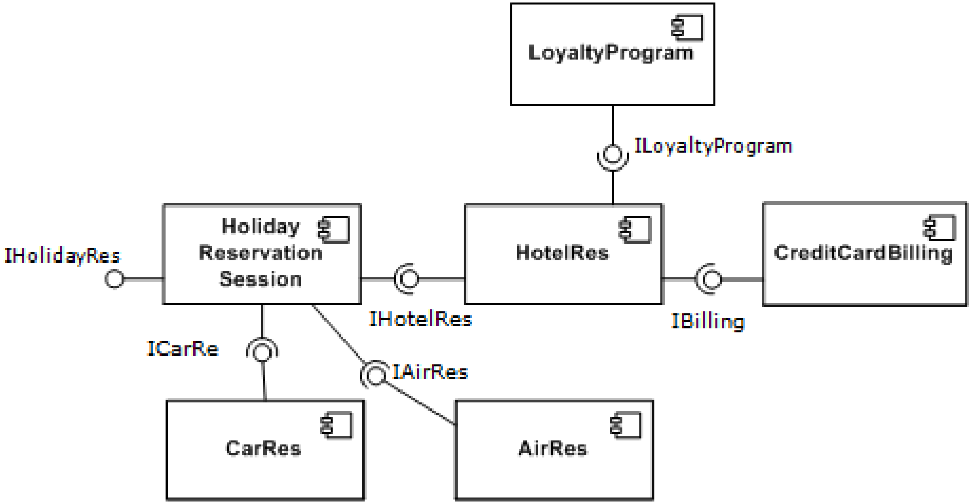
- A design pattern which is an application of IoC.
- Dependency injection creates a graph of dependencies by inverting the control to find associated objects, *pushing dependencies from the core to the edges* - up to a *composition root*.
- Break the dependency from new/factories (“the new new”).
- Greatly simplifies **testing**, improves **modularity**.

Types of injection

- Constructor injection
  - Dependencies are provided via constructor parameters
- Setter injection (also: field injection)
  - Dependencies are provided by calling specific setters
- Interface injection
  - Injection methods are declared in interfaces, a class implements an injection interface for each dependency to inject
- Method injection
  - Used when a dependency can be variously solved on a per-operation basis (perform this operation using this service)

Component based architectures

- In component-based software engineering (CBSE), software systems are built by gluing together software components on the basis of *provided* and *required* interfaces.



DI and CBSE

- By exposing the dependencies to be injected in its interface an object exposes both what it *provides* and what it *requires*, easing a component based approach to software design.

“Clean” DI

- Use constructor injection for dependencies always needed that do not change for the lifetime of the instance.
- Use method injection for dependencies that are needed only during the invocation of that method.

“Clean” DI

When binding has to be solved at run-time pass factories to constructors or methods.

- Try to isolate choice points in strategy-like structures.
- If the language allows, let the dependency emerge from the signature of the factory.
  For example, in Java, make factories implement a `Factory<Dependency>` interface.

DI in Java

- Pure DI, no framework, the composition root is close to the *entry point(s) of the application*.
- DI frameworks: Guice, Dagger, Spring, CDI, …
  - Annotations are used to mark dependencies that have to be injected

Example

```
public void postButtonClicked() {
  String text = textField.getText();
  if (text.length() > 140) {
      Shortener shortener = new TinyUrlShortener();
      text = shortener.shorten(text);
  }
  if (text.length() <= 140) {
      Tweeter tweeter = new SmsTweeter();
      tweeter.send(text);
      textField.clear();
  }
}
```

From Google I/O 09

Example

```
public class TweetClient {
  private final Shortener shortener;
  private final Tweeter tweeter;

  public TweetClient(Shortener shortener, Tweeter tweeter) {
        this.shortener = shortener;
      this.tweeter = tweeter;
  }

  public void postButtonClicked() {
      …
      if (text.length() <= 140) {
          tweeter.send(text);
          textField.clear();
  }
```

From Google I/O 09

Example with Guice

```
import com.google.inject.Inject;

public class TweetClient {
  private final Shortener shortener;
  private final Tweeter tweeter;

  @Inject
  public TweetClient(Shortener shortener,
     Tweeter tweeter) {
      this.shortener = shortener;
      this.tweeter = tweeter;
  }
```

Example with Guice

```
import com.google.inject.AbstractModule;

public class TweetModule extends AbstractModule {
  protected void configure() {
    bind(Tweeter.class)
      .to(SmsTweeter.class);
    bind(Shortener.class)
        .to(TinyUrlShortener.class);

  }

}
```

Example with Guice

```
public static void main(String[] args) {
  Injector injector =
      Guice.createInjector(new TweetModule());
  TweetClient tweetClient =
      injector.getInstance(TweetClient.class);
  tweetClient.show();
}
```