Programming Fundamentals 2

Pierre Talbot

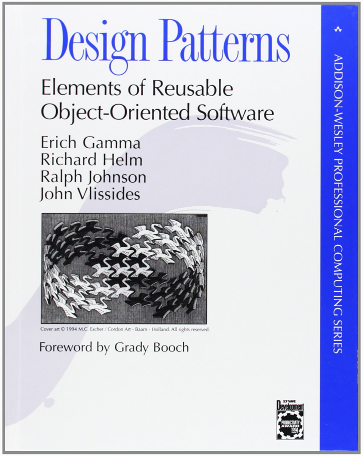
6 May 2021

University of Luxembourg

Chapter X. Design Pattern

Design pattern

- A design pattern is a **reusable** general solution to a software problem.
- A way to organise the code to increase flexibility, reusability, maintainability, ...
- Generally based on inheritance, subtype polymorphism, and interfaces.


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Why are design patterns interesting?

- Introduce a common vocabulary among developers: make it easier to understand the code.
- They are robust solutions, designed over the years by expert developers.
- Extensible and modular: weak coupling between software components.

Classification of design patterns

1. **Creational patterns**: to build an object when it is complicated (e.g., to “help” the constructor).
    - *Factory, AbstractFactory, Builder, ...*
    - `ASCIIBattlefieldBuilder` builds `Battlefield`.
2. **Structural patterns**: to extend a class with functionalities without modifying it.
    - *Adapter, Facade, Decorator, Proxy, Composite, ...*
3. **Behavioral patterns**: to introspect an object and/or customized its behavior.
    - *Iterator, Observer, Strategy, Visitor, ...*
    - `TileVisitor` allows us to visit the tiles of the battlefield.

A selection of design patterns

We discuss five design patterns:

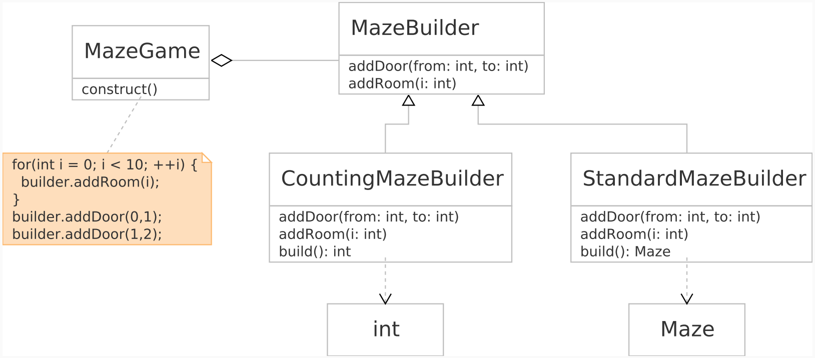
1. *Builder pattern*: used in LOL 2D.
2. *Composite pattern*: used in Calculator and MC (lab 4).
3. *Facade pattern*: used in LOL 2D.
4. *Visitor pattern*: used in LOL 2D.
5. *Observer pattern*: should be used in LOL 2D.

Builder Design Pattern

Builder Pattern: Intent

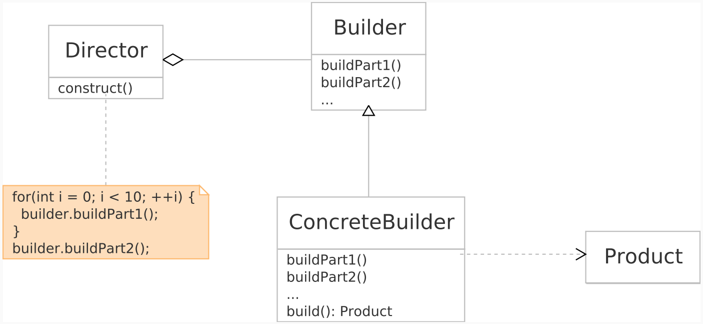
Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Motivation: Maze Builder



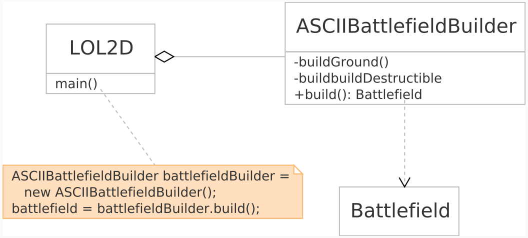
```
for(int i = 0; i < 10; ++i) {
  builder.addRoom(i);
}
builder.addDoor(0,1);
builder.addDoor(1,2);
```

General case: Builder design pattern



```
for(int i = 0; i < 10; ++i) {
  builder.buildPart1();
}
builder.buildPart2();
```

A restricted usage in LOL 2D



```
ASCIIBattlefieldBuilder battlefieldBuilder =
   new ASCIIBattlefieldBuilder();
battlefield = battlefieldBuilder.build();
```

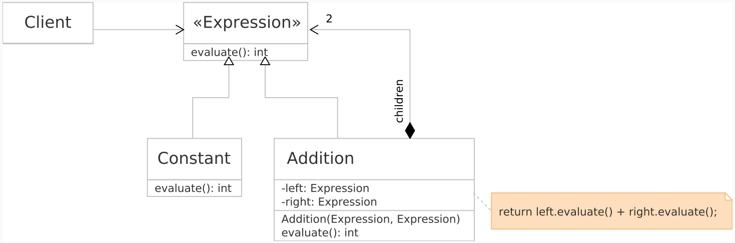
- Constructing the battlefield with an ASCII file is 100 LOC.
- Usage of the builder to separate object construction from the object itself.
- Currently, no need for a `Builder` interface.
- Could be added later when required, *e.g.*, suppose you want to propose a map editor.

Composite Design Pattern

Intent

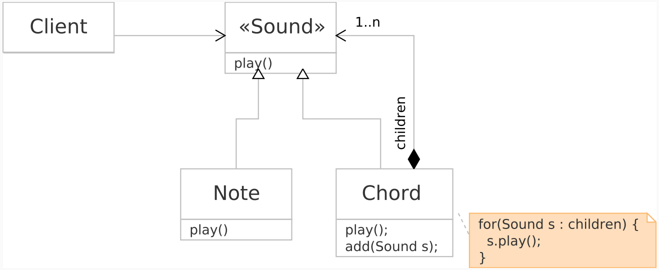
Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Motivation: Calculator



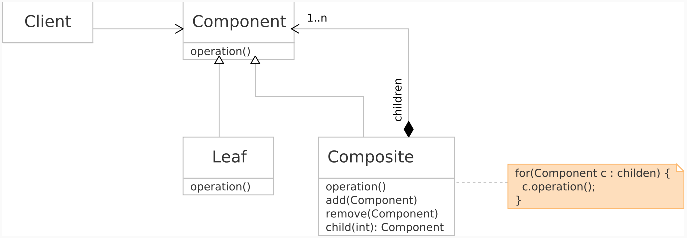
A constant or a *composition* of constants through `Addition` are manipulated uniformly through `Expression`.

Motivation: Musical score



A note or a *composition* of notes through `Chord` are manipulated uniformly through `Sound`.

General case: Composite design pattern

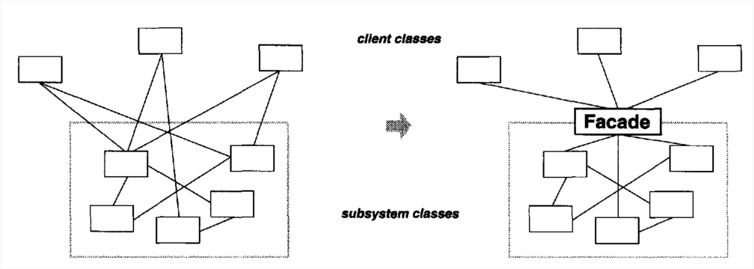


```
for(Component c : childen) {
  c.operation();
}
```

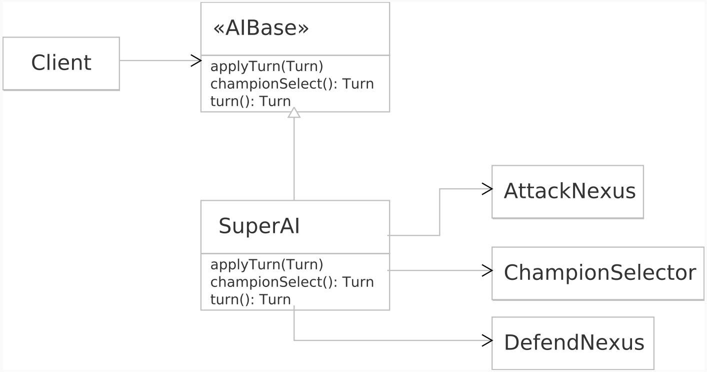
Facade Design Pattern

Intent

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.



Motivation: AI interface



- Each client can implement its own AI, which might be super sophisticated and involves many components.
- All AIs are used in Client the same way, through the facade `AIBase` interface.

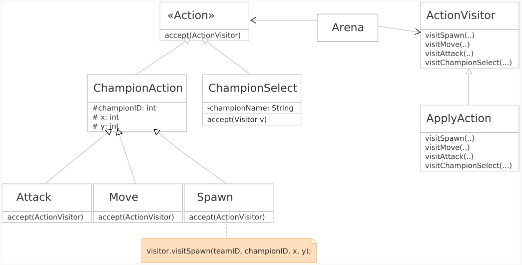
Visitor Design Pattern

Intent

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.

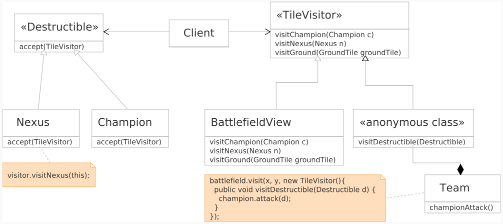
It is a solution to the *Expression problem* mentioned in Live coding 4.

Motivation: Action on battlefield



- An action has an effect on the battlefield (e.g., moving a champion, attacking a destructible, ...).
- The class `Turn` has an `ArrayList<Action>`.
- How to iterate over the list of actions, and know the concrete subtype?
- The visitor pattern allows us to introspect the actions.

Motivation: Tiles of the battlefield

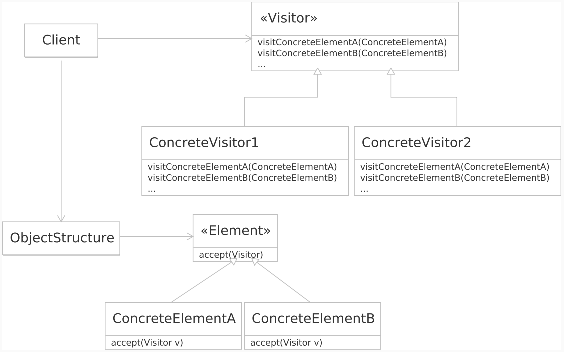


```
visitor.visitNexus(this);
```

```
battlefield.visit(x, y, new TileVisitor(){
  public void visitDestructible(Destructible d) {
    champion.attack(d);
  }
});
```

- The battlefield is constituted of different kind of tiles, either ground or destructible.
- The visitor allows us to introspect a destructible tile.

General case: Visitor design pattern

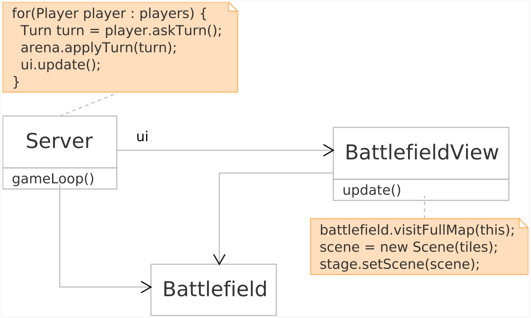

Client
«Visitor»
visitConcreteElementA(ConcreteElementA)
visitConcreteElementB(ConcreteElementB)
...
ConcreteVisitor1
visitConcreteElementA(ConcreteElementA)
visitConcreteElementB(ConcreteElementB)
...
ConcreteVisitor2
visitConcreteElementA(ConcreteElementA)
visitConcreteElementB(ConcreteElementB)
...
ObjectStructure
«Element»
accept(Visitor)
ConcreteElementA
accept(Visitor v)
ConcreteElementB
accept(Visitor v)

Observer Design Pattern

Intent

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Motivation: Server / UI communication

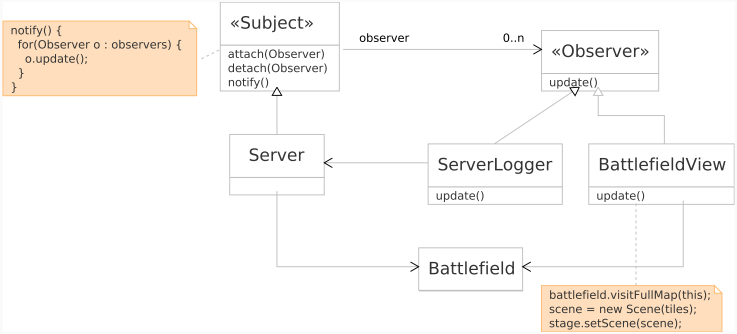


```
for(Player player : players) {
  Turn turn = player.askTurn();
  arena.applyTurn(turn);
  ui.update();
}
```

```
battlefield.visitFullMap(this);
scene = new Scene(tiles);
stage.setScene(scene);
```

Currently, the server directly communicates to the UI.

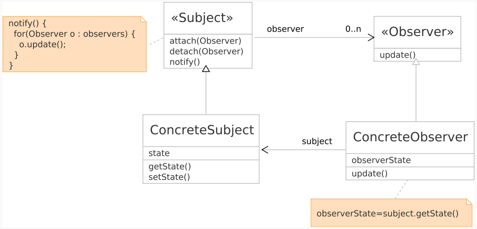
Motivation: Server / UI observer



Observer pattern in Java

In Java, the interface `Observer` and the class `Observable` (`Subject` in the example) are already provided!

General case: Observer design pattern



```
notify() {
  for(Observer o : observers) {
    o.update();
  }
}
```