COMP-8117
Advanced Software Engineering Topics

GoF Design Patterns

Dr. Aznam YACOUB

University of Windsor

Agenda

- Introduction
- History
- Gang of Four’s Design Patterns
  - Creational DP
  - Structural DP
  - Behavioral DP
- Bibliography

Introduction

Introduction

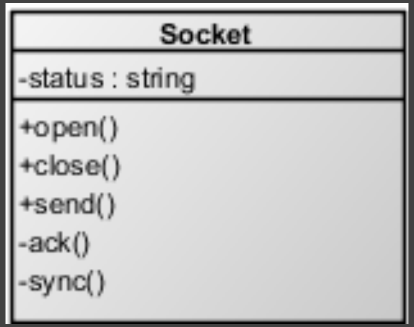
- Some Software Engineer’s Tasks & Responsabilities:
  - Understand functional and technical needs.
  - Formalize and analyze requirements and specifications.
  - **Plan Software Design & Architecture.**
  - Manage the development process.
  - Conduct the Verification and Validation processes.
  - …

Introduction

Software Design & Architecture is difficult activity.

1. Find a suitable algorithm to solve a given problem.

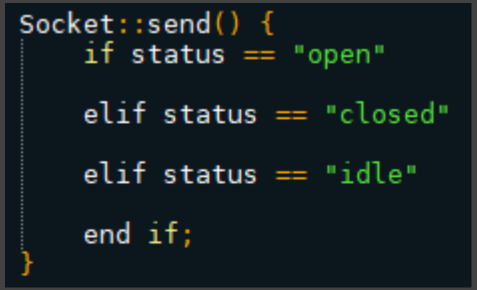
- *Example: Implement a TCP/IP socket with a send function.*
- One solution in OOP paradigm:



Socket

-status : string

+open()
+close()
+send()
-ack()
-sync()



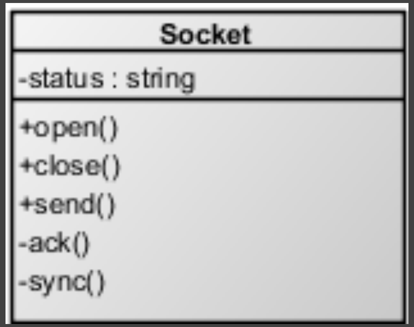
```
Socket::send() {
    if status == "open"

    elif status == "closed"

    elif status == "idle"

    end if;
}
```

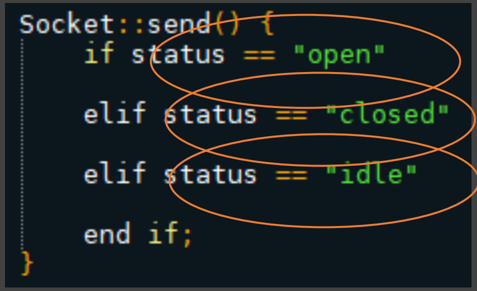
Introduction

Software Design & Architecture is difficult activity.

1. Find a suitable algorithm to solve a given problem.

- *Example: Implement a TCP/IP socket with a send function.*
- **Problem : Each time a socket is sent, a state verification is performed.**





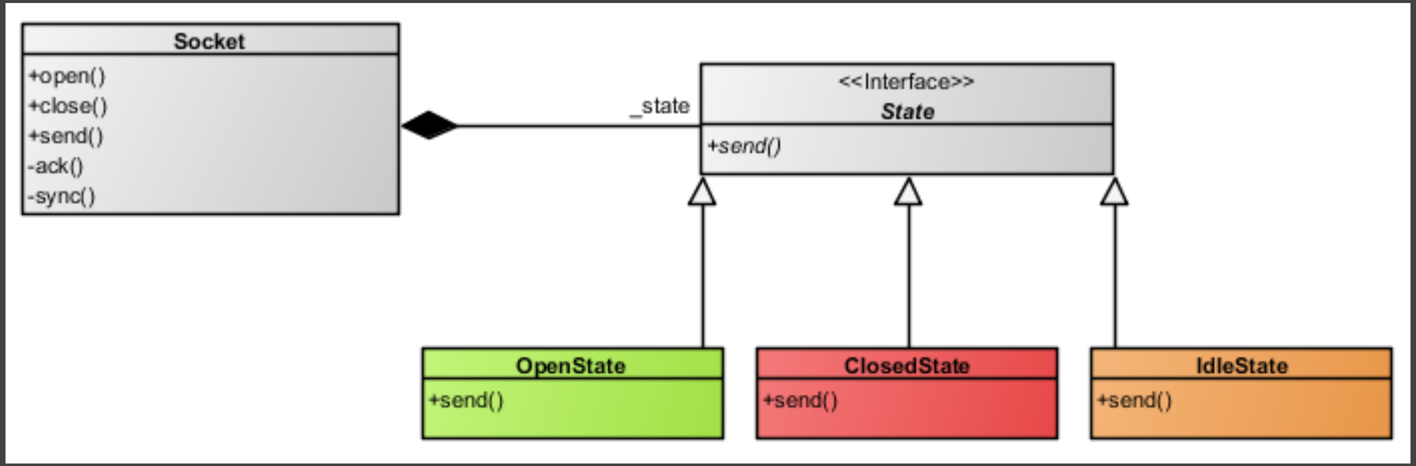
```
Socket::send() {
    if status == "open"

    elif status == "closed"

    elif status == "idle"

    end if;
}
```

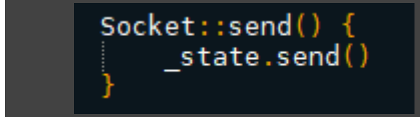
Introduction

Software Design & Architecture is difficult activity.

2. Find a suitable architecture to solve a given problem.

- *Example: Implement a TCP/IP socket with a send function.*
- **Possible Solution: Separate different behaviours into different classes.**





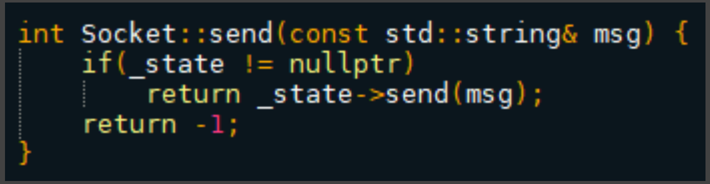
```
Socket::send() {
    _state.send()
}
```

Introduction

Software Design & Architecture is difficult activity.

3. Find a suitable implementation to solve a given problem.

- *Example: Implement a TCP/IP socket with a send function.*
- **Depends on technical properties (language, technical optimization, etc).**



```
int Socket::send(const std::string& msg) {
    if(_state != nullptr)
        return _state->send(msg);
    return -1;
}
```

Introduction

- Developping ***reusable*** software and components is even harder.
  - ➢ Find pertinent objects.
  - ➢ Factor objects into classes at the right granularity.
  - ➢ Define key relationships between classes and objects.
  - ➢ Almost impossible to find a flexible design right the first time (except if you are a genious !)

Introduction

Software Design and Architecture is an Art !

- ➢ New designers can quickly be overwhelmed by the complexity of modern systems and tend to fallback on bad designs (non-OO techniques, etc).
- ➢ Unexperienced architects can quickly produce heavy architectures hard to maintain.
- ➢ Experienced designers tend to produce good designs.

Introduction

- When a solution is found, we use it again, again and again.
  - ➢ If you are familiar with a solution, you can apply it without rediscovering it.
  - ➢ If you recognize different subproblems and know the solutions, you almost only need to think about « how to plug » these solutions.
  - ➢ If you identify a problem and its solution, you can easily teach it and share knowledge.

- These solutions include ***patterns*** and ***fameworks***.

The Road to « Software Design Doctor » Rank

Levine, *« CS 342 : Patterns and Framework »*

1. Learn the rules ! (algorithms, data structures, languages, etc.)
2. Learn the principles ! (structured programming, advanced paradigms, etc.)
3. **Learn from the other Masters ! (This lecture)**
   - Study existing designs and understand them.
   - Memorize them.
   - Apply them repeatedly.
   - *Try to adapt them and improve them if you face a new situation.*

History : Once upon a time…
Design patterns

Make the History

- 1979, C. Alexander, *The Timeless Way of Building*

« 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, without ever doing it the same way twice. »

- Pattern = Well-known solution to a recurrent problem.

Make the History

- 1993, F. Buschmann, *Pattern-Oriented Software Architecture*

“Patterns expose knowledge about software construction that has been gained by many experts over many years. All work on patterns should therefore focus on making this precious resource widely available. Every software developer should be able to use patterns effectively when building software systems. When this is achieved, we will be able to celebrate the human intelligence that patterns reflect, both in each individual pattern and in all patterns in their entirety. ”

Make the History

- 1995, Gang of Four (Gamma, Helm, Johnson, Vlisside), *Design Patterns: Elements of Reusable Object-Oriented Software*

Define **23 classical reference Design Patterns** (DP) which must be known by any software engineers and OO developers.

➢ Today, literature describes hundreds of DPs.

Categories of Patterns

WARNING

In this lecture, Pattern and DP are used as synonyms, but actually, there are different categories of patterns.

- Architectural Patterns: Define structural organization of software (pipes, filters, brokers, deep learning, etc.).
- Design Patterns: Define common structural designs (what we'll study in this lecture).
- Coding Patterns: Specific implemented idioms in a particular language.
- Anti-Patterns: The bad practices or how to avoid a bad practice.
- Organizational Patterns: How to organize people in software engineering.

What’s a Design Pattern ?

- DPs are solutions to **recurring** problems in a **particular context**.

- Patterns capture the static and dynamic structures and collaborations between key participants in software design.

- DPs define reusable high-level structures and behaviours in software design.

What’s a Design Pattern ?

- Shorter, a DP:
  - Defines a recurring idiom or software structure.
  - In an abstraction, from any programming language, of a common solution.
  - Is independent of the algorithms.
  - Is a common language for any developers and designers without ambiguity
  - Makes easier technical design by proposing templates to solve problems in a specific situation.
  - Allows focus on functional design (because they articulate technical and non-functional aspects).
  - (OO) Shows relationships between classes and objects.

What’s a Design Pattern ?

- Shorter, a DP is **NOT**:
  - A code snippet.
  - A finished design which can be directly translated into code.
  - A way to specify final application classes and objects which are involved in the final architecture.

- Patterns => Description of an abstract solution.
- Frameworks => Reusable architecture, detailed design and code.

Benefits of DPs

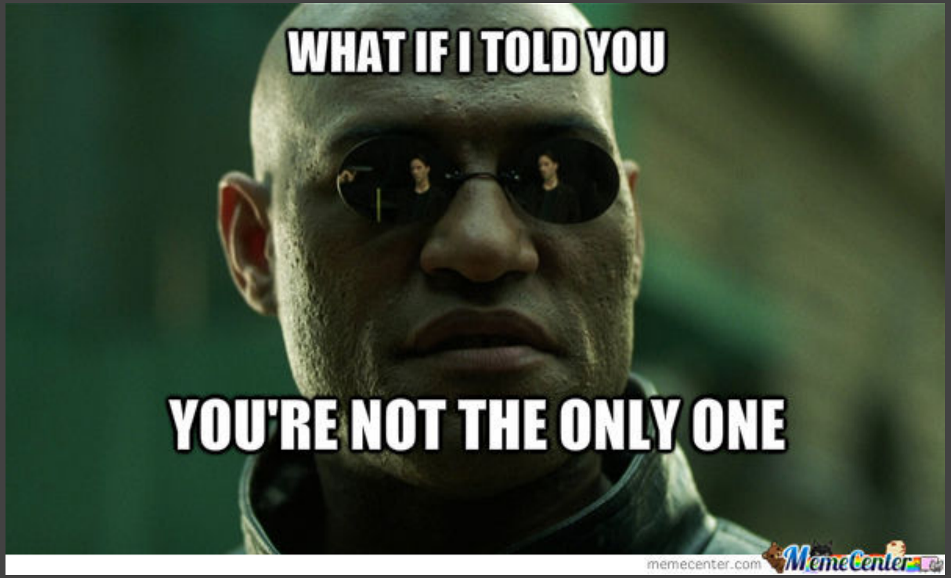
- Improve software quality and reduce development time (modularity, extensibility, performance, reusability).
- Allow engineers to abstract a problem and talk about that abstraction in isolation from its implementation.
- Capture expertise : make easier for other developers to understand what you want to do and how a system is built.
- Improve understandability (patterns are described well, once).

Drawbacks of DPs

- DPs don’t directly lead to code. You can have a good pattern and a poor implementation, or code cannot be reused directly.
- Patterns are useful but careless use leads to **overdesign**.
    - Especially, don’t use a pattern when the direct solution is simple or a linear set of instructions.
- Patterns need a strong expectations management.
- Patterns are validated by experience and not automated testing.
- Patterns integration is a human-intensive activity (for the moment at least..)

Describe a DP

- Name it. *Example: The Singleton Pattern.*



Describe a DP

- Describe the problem. When ? What ? Why ? Scenarios ? Context ?

- *Example : Sometimes we really only ever need (or want) one instance of a particular class. For instance, a keyboard reader, a bank data collection, only one game user interface.*

Describe a DP

- Describe the solution: Forces addressed, abstract description of structure and collaborations, responsabilities and relationships between elements.

- In OO DP, describe classes and objects needed in a language-neutral way (use UML or any other high-level design formalism).

Describe a DP

*Example: A singleton is an **object** which the only one of its type.*

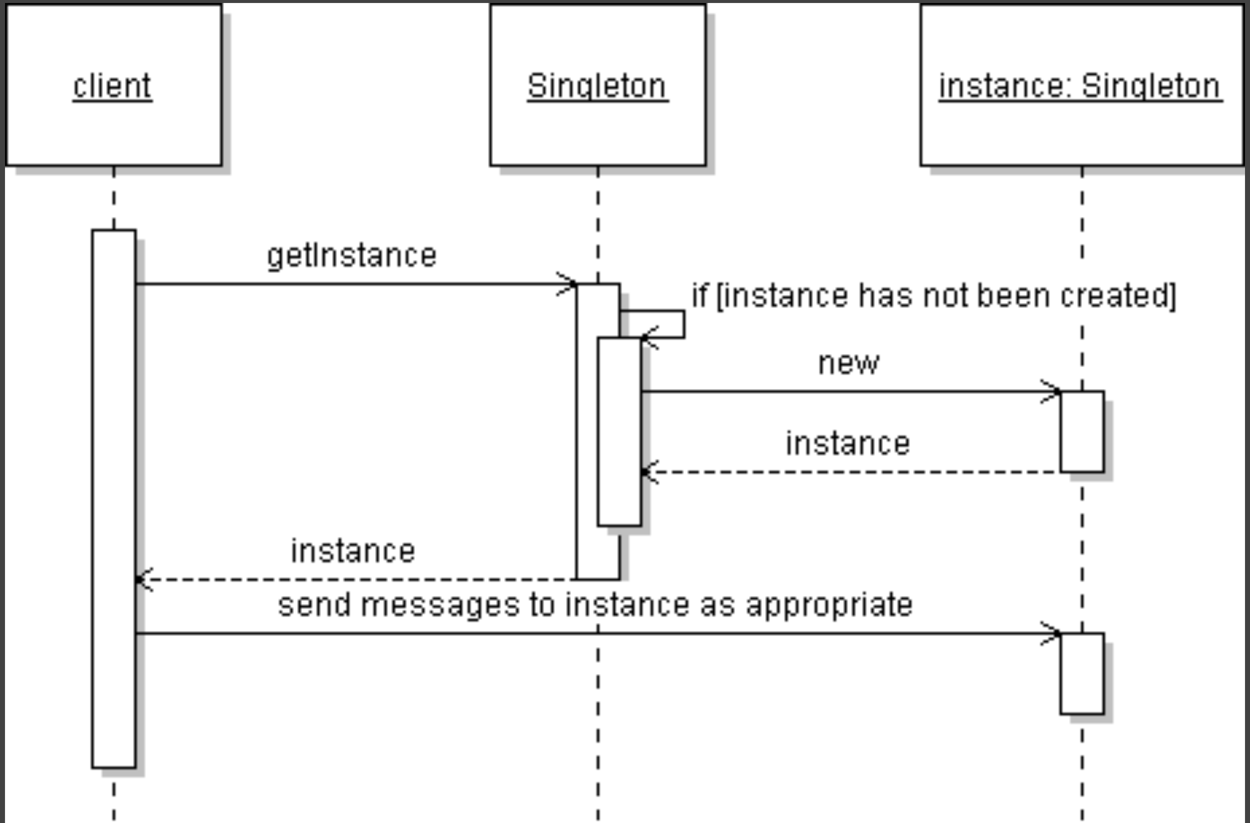
- *Ensuring a class has at most one instance.*
- *Providing a global access point to this instance.*

Make the constructors private so they cannot be called from outside by the client.

Declare a private static instance of the class.

Write a public method that allows access to this static instance.

Describe a DP


client
Singleton
instance: Singleton
getInstance
if [instance has not been created]
new
instance
instance
send messages to instance as appropriate

Describe a DP

- Describe the consequences: Strengths and weaknesses, consequences, results and trade-off (cost vs benefits), implementation guidelines, sample code, known uses and related patterns, existing variations.

Describe a DP

Example:

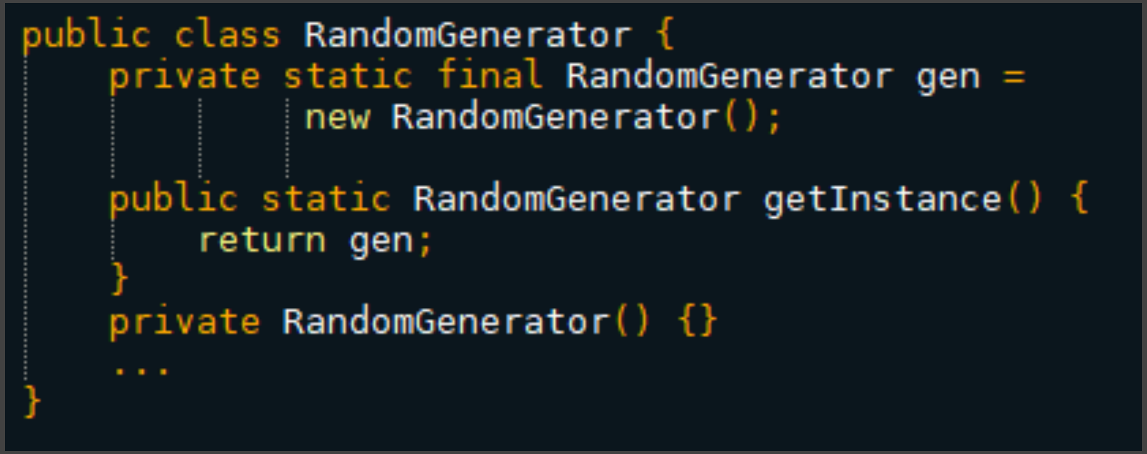
Benefits - Takes responsibility of managing that instance away from the programmer (illegal to construct more instances). Saves memory. Avoids bugs arising from multiple instances.

Disadvantages - Lacks flexibility. Static methods can't be passed as an argument, nor returned.

Variations – Specific implementation for thread-safe context.

Describe a DP

Example: Implementing a Singleton Random Generator



```
public class RandomGenerator {
    private static final RandomGenerator gen =
            new RandomGenerator();

    public static RandomGenerator getInstance() {
        return gen;
    }
    private RandomGenerator() {}
    ...
}
```

The Gang of Four’s Design Patterns

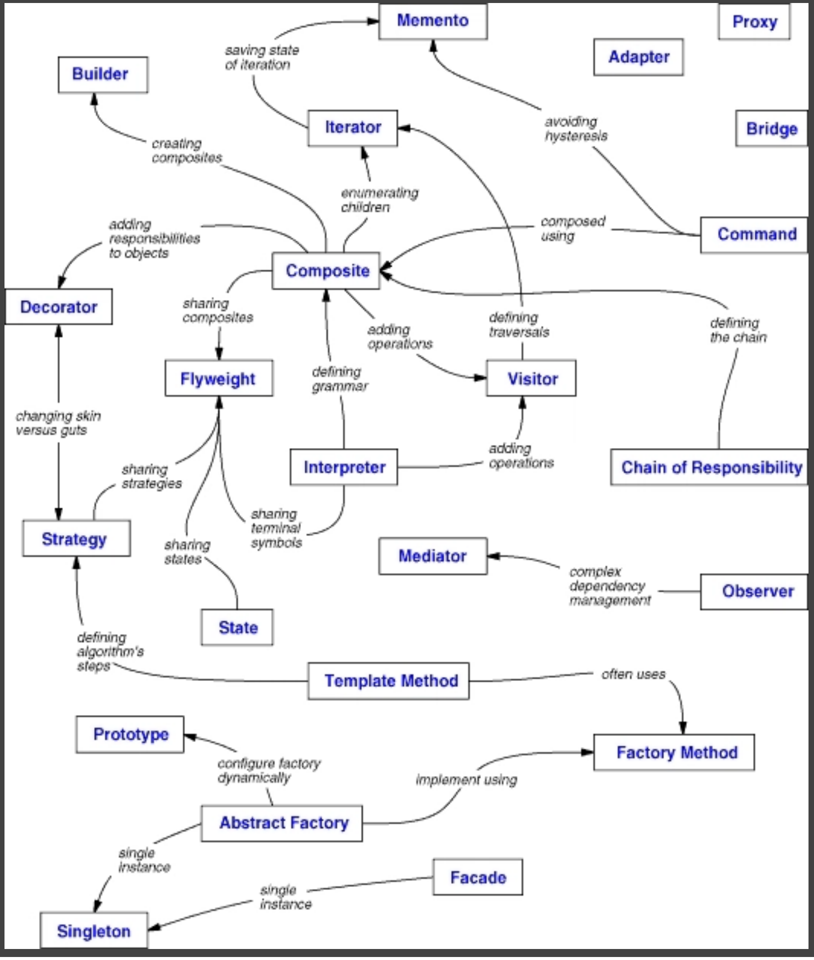
The Gang of Four’s Design Patterns

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

**Class scope: Relationship between classes and subclasses are statically defined when compiling code (inheritance).*

**Object scope: Relationship between classes and subclasses are dynamically instantiated and modified during the execution (composition).*

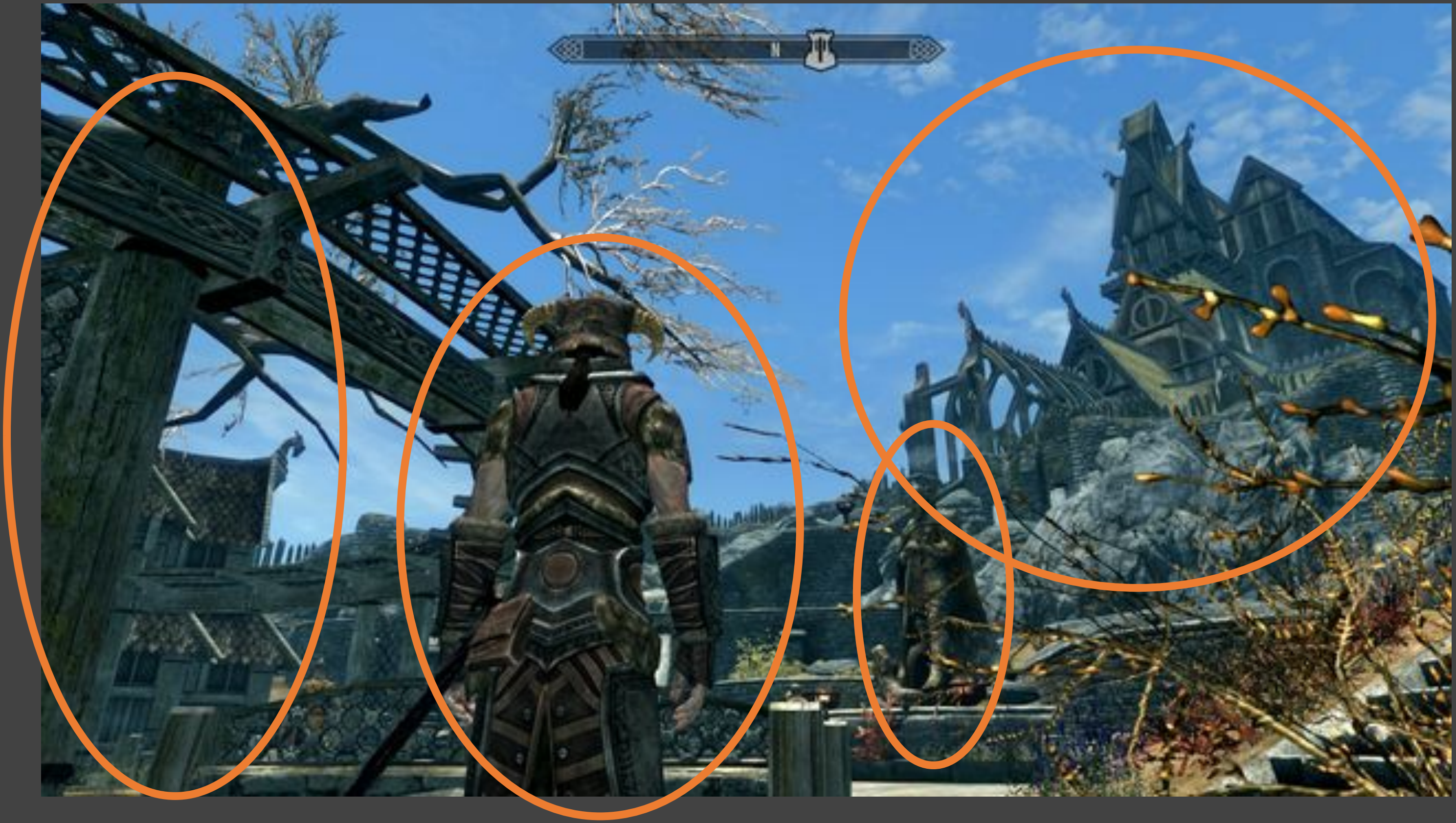
The GoF DPs – Relationship between Patterns



The GoF DPs – 3 categories of Patterns

- Creational Patterns : Abstraction of the object-instantiation process
  - ➢ Make the application independent of the instantation/initialization processes.
  - ➢ Encapsulate concrete classes.
  - ➢ Hide what is created / initialized / configuring, when, where and by who.

GoF DP – Creational Pattern - Builder



The problem : Looking at Skyrim. When we start a new game, we need to create a scene with:

- a custom character in a specific map. The character can have different weapons.

- a map with a specific scenery.

GoF DP – Creational Pattern - Builder

Our needs : A design which builds dynamically the scene

- Independently of the objects composing the map;
- Independently of the character;
- Different combinations should be possible.

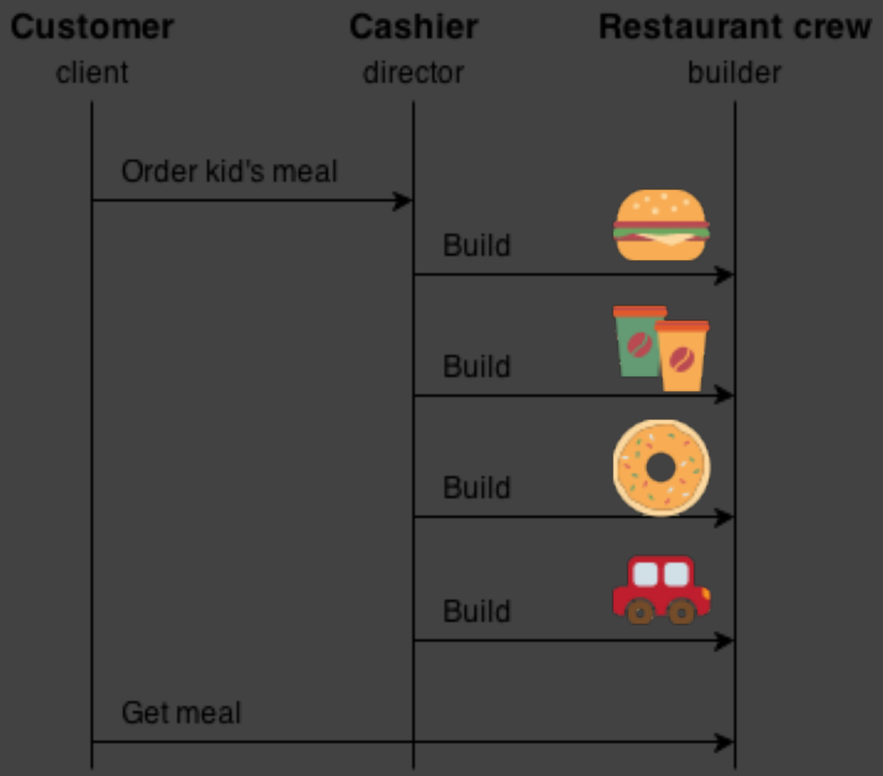
GoF DP – Creational Pattern - Builder

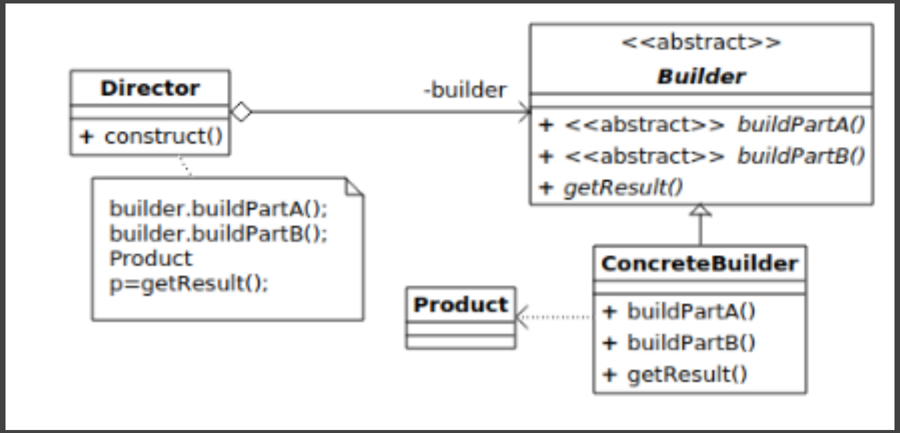
- GoF's DPs describe the **Builder Pattern**.
- Intent:
  - *Create complex objects whose the components must be created following a specific order or specific algorithm.*
  - *Dissociate object building and its representation. In this way, the same build process can create different representations.*

GoF DP – Creational Pattern - Builder

- Problems:
  - *Object creation is performed part by part. Each part can be variable.*
  - *Object creation needs a lot of [optional] arguments. Some of these arguments can have varying types.*
- Solutions:
  - *Encapsulate object creation by letting the reponsibility of the creation to an external class.*

GoF DP – Creational Pattern - Builder

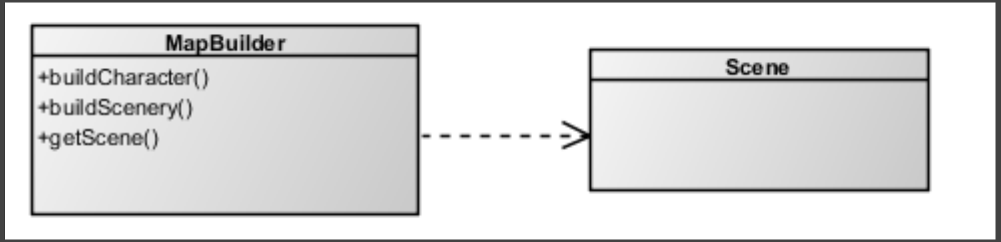


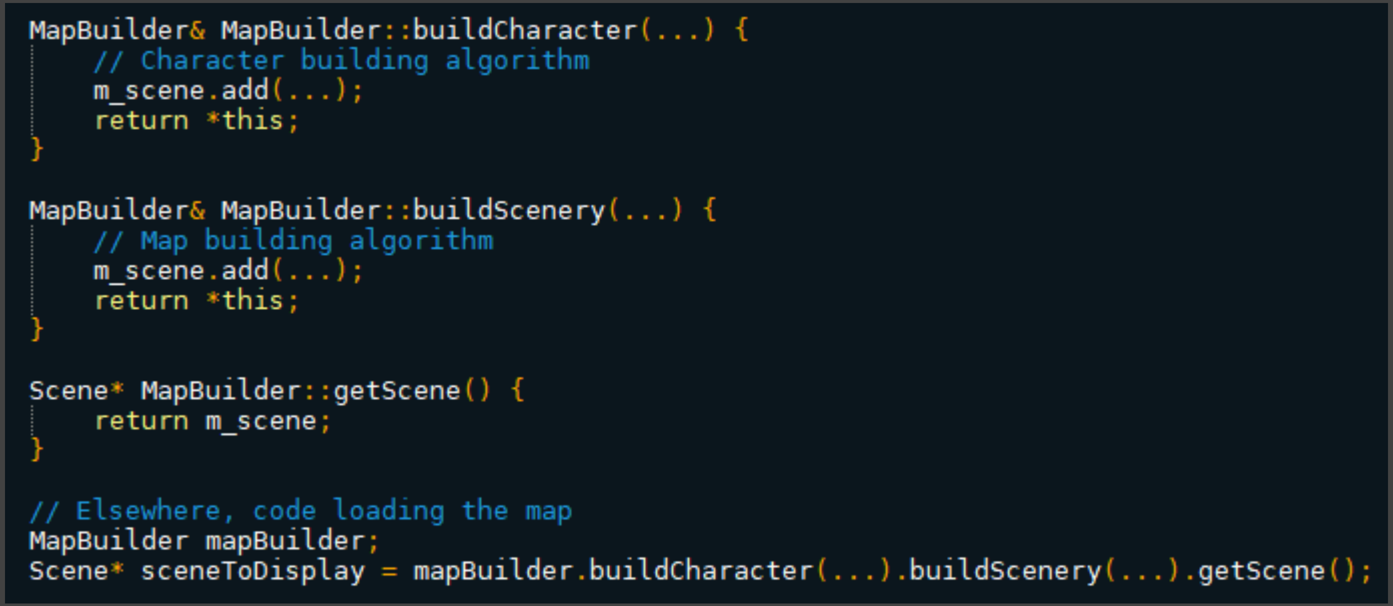


```
builder.buildPartA();
builder.buildPartB();
Product
p=getResult();
```

GoF DP – Creational Pattern - Builder

- The Builder corresponds to our problem. So we apply it.





```cpp
MapBuilder& MapBuilder::buildCharacter(...) {
    // Character building algorithm
    m_scene.add(...);
    return *this;
}

MapBuilder& MapBuilder::buildScenery(...) {
    // Map building algorithm
    m_scene.add(...);
    return *this;
}

Scene* MapBuilder::getScene() {
    return m_scene;
}

// Elsewhere, code loading the map
MapBuilder mapBuilder;
Scene* sceneToDisplay = mapBuilder.buildCharacter(...).buildScenery(...).getScene();
```

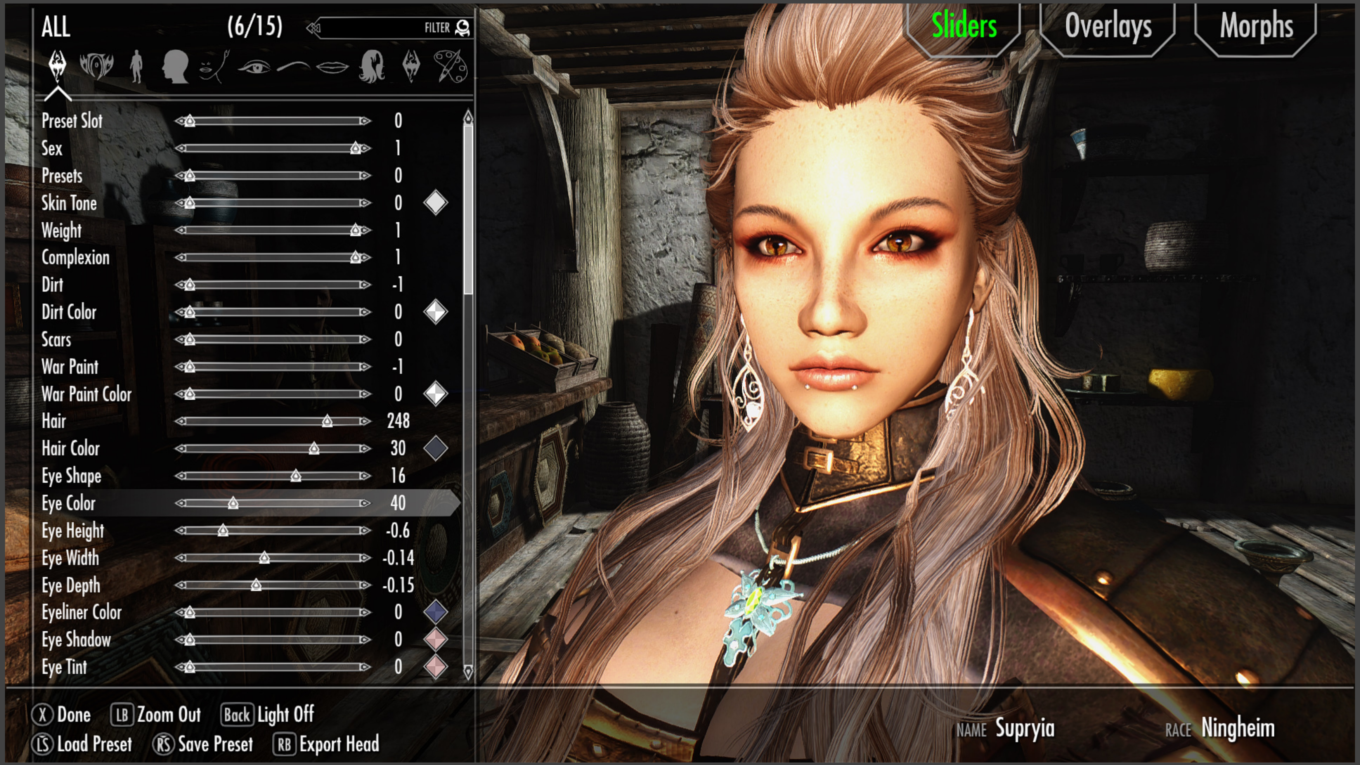
In this case…

- The builder answers a part of our problem.
- Concerning the character with different weapons, we’ll need a specific builder for it.
- If the character is defined in a hierarchical way, we’ll probably combine the builder with a composite pattern.

The GoF DPs – 3 categories of Patterns

- Structural Patterns : How Objects/Class are combined
  - ➢ Describe how classes and instances are organized and independently connected in order to obtain new complex functionalities.
  - ➢ Separate ***interface*** and ***implementation***.
  - ➢ Related to Class and Object Composition.

GoF DP – Structural Pattern - Composite

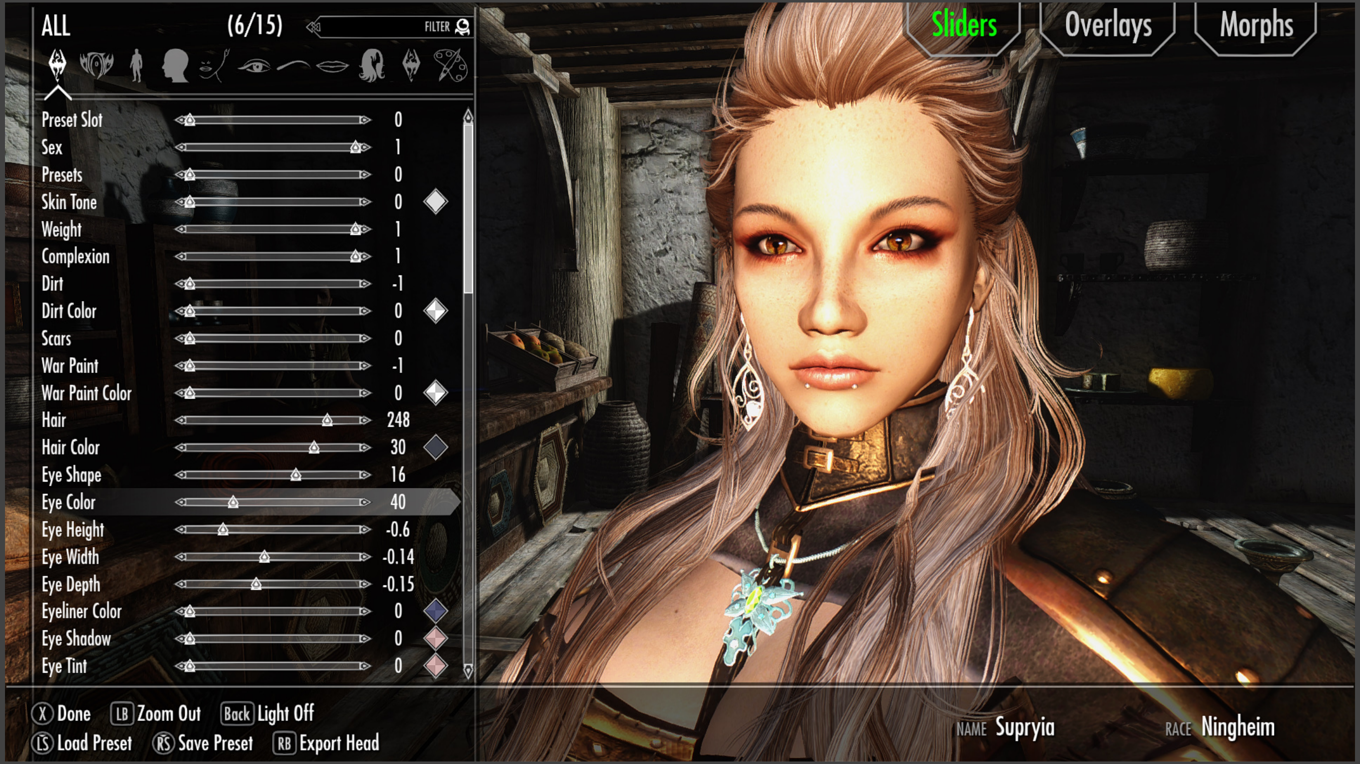


Preset Slot 0
Sex 1
Presets 0
Skin Tone 0
Weight 1
Complexion 1
Dirt -1
Dirt Color 0
Scars 0
War Paint -1
War Paint Color 0
Hair 248
Hair Color 30
Eye Shape 16
Eye Color 40
Eye Height -0.6
Eye Width -0.14
Eye Depth -0.15
Eyeliner Color 0
Eye Shadow 0
Eye Tint 0

NAME Supryia
RACE Ningheim

- Look at Skyrim again. A character has different part: a head, a upper body, a lower body.
- A upper body is defined by chest, arms.
- Arms are defined by shoulder, hand, fingers.
- Etc.

GoF DP – Structural Pattern - Composite



Preset Slot	0
Sex	1
Presets	0
Skin Tone	0
Weight	1
Complexion	1
Dirt	-1
Dirt Color	0
Scars	0
War Paint	-1
War Paint Color	0
Hair	248
Hair Color	30
Eye Shape	16
Eye Color	40
Eye Height	-0.6
Eye Width	-0.14
Eye Depth	-0.15
Eyeliner Color	0
Eye Shadow	0
Eye Tint	0

NAME Supryia

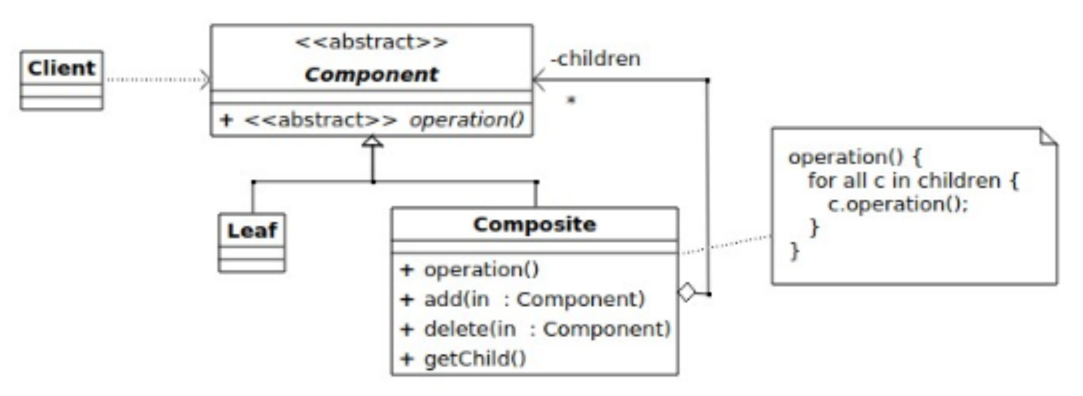
RACE Ningheim

- We want to implement an efficient collision algorithm: we want to adapt the possible damages to the damaged zone (the head, the hand, a finger).
- We'll not test the collision with every part, and effects can differ according to the damaged zone (an headshot is not equal to a hand injury).

GoF DP – Structural Pattern - Composite

- GoF’s DPs define the **Composite Pattern**.
- Intent:
  - *Compose objects into tree structures to represent whole-part hierarchies.*
  - *Composite lets clients treat individual objects and compositions of objects uniformly.*
- Problem:
  - *Application needs to manipulate a hierarchical collection of "primitive" and "composite" objects. Processing of a primitive object is handled one way, and processing of a composite object is handled differently. Having to query the "type" of each object before attempting to process it is not desirable.*

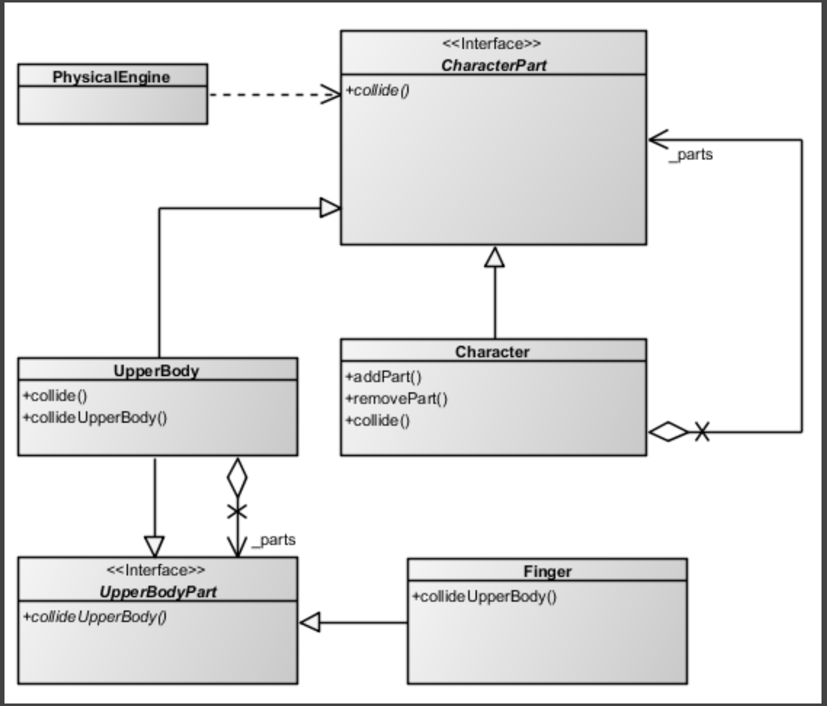
GoF DP – Structural Pattern - Composite

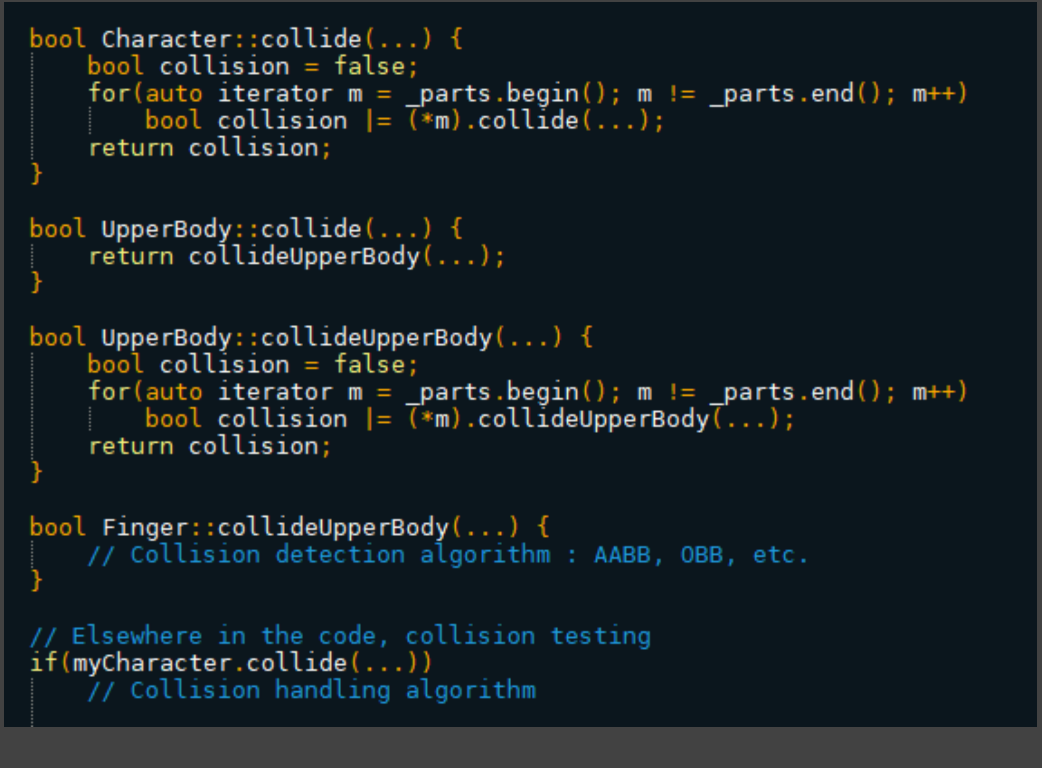


```
operation() {
  for all c in children {
    c.operation();
  }
}
```

GoF DP – Structural Pattern - Composite

- The Composite pattern fits with our needs. We apply it.





```
bool Character::collide(...) {
    bool collision = false;
    for(auto iterator m = _parts.begin(); m != _parts.end(); m++)
        bool collision |= (*m).collide(...);
    return collision;
}

bool UpperBody::collide(...) {
    return collideUpperBody(...);
}

bool UpperBody::collideUpperBody(...) {
    bool collision = false;
    for(auto iterator m = _parts.begin(); m != _parts.end(); m++)
        bool collision |= (*m).collideUpperBody(...);
    return collision;
}

bool Finger::collideUpperBody(...) {
    // Collision detection algorithm : AABB, OBB, etc.
}

// Elsewhere in the code, collision testing
if(myCharacter.collide(...))
    // Collision handling algorithm
```

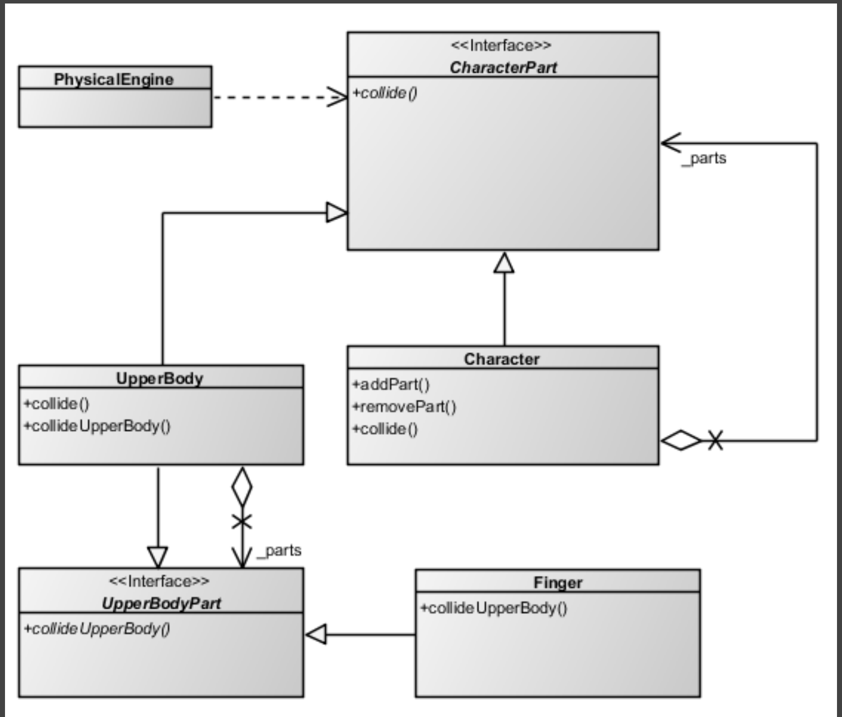
In this case…

- So, the collision detection algorithm is totally transparent and delegated to the each part.
- However, suppose you want to adapt your collision algorithm according to the current performance of the target computer.

The GoF DPs – 3 categories of Patterns

- Behavioural Patterns : How objects communicate
  - ➢ Describe behaviours and interactions between objects and how responsibilities are shared in a service.
  - ➢ Related to dynamic communication between objects.
  - ➢ Describe algorithms.

GoF DP – Behavioural Pattern - Strategy

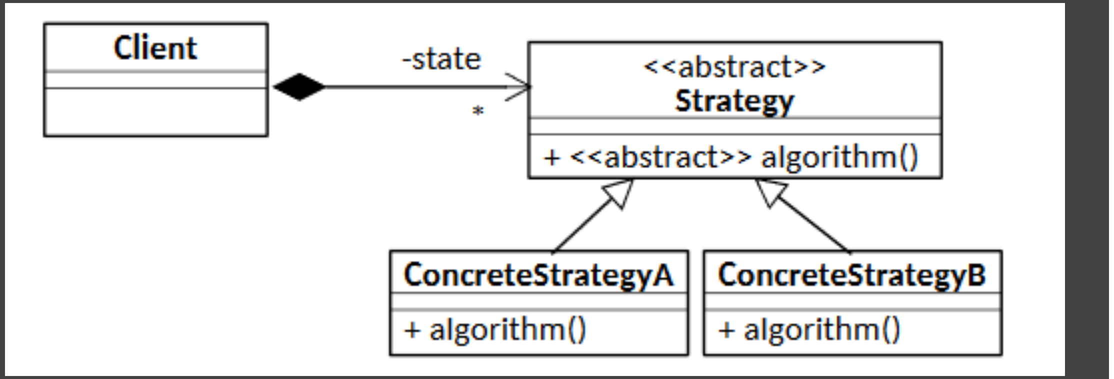


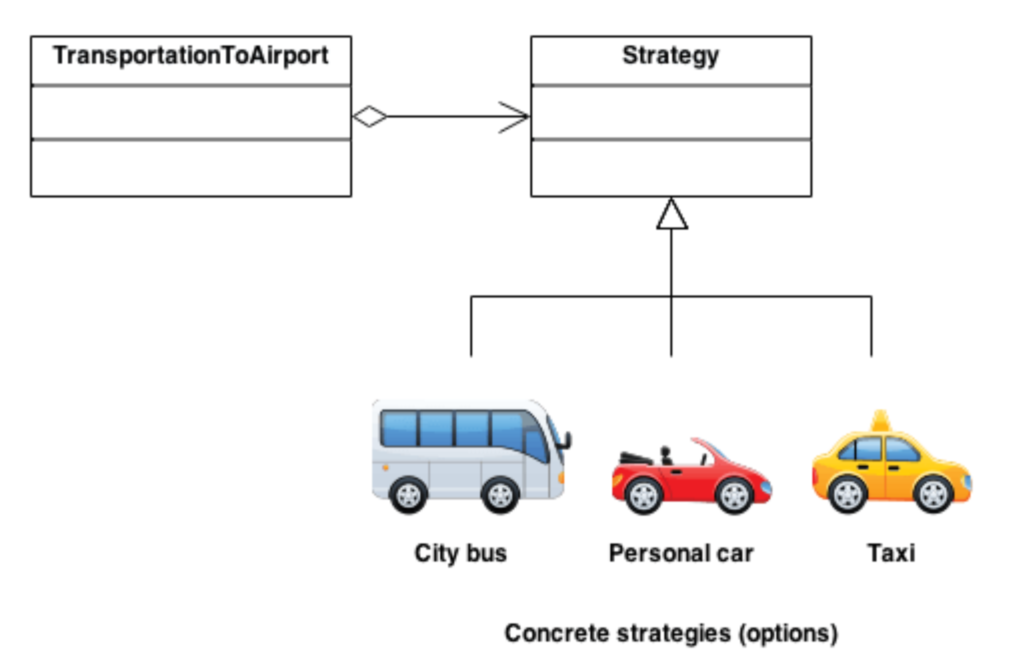
- We want to adapt the collide() method according to external parameters without changing the architecture.
- Especially, the detection algorithm is basically the same for each part, only the parameters change, and the processing of the result (damage computation for instance).

GoF DP – Behavioural Pattern - Strategy

- GoF’s DPs define the **Strategy Pattern**.
- Intent:
  - *Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.*
  - *Capture the abstraction in an interface, bury implementation details in derived classes.*
- Problem: *Selection of the algorithm depends only on the client who performs the request. Only the algorithm changes.*

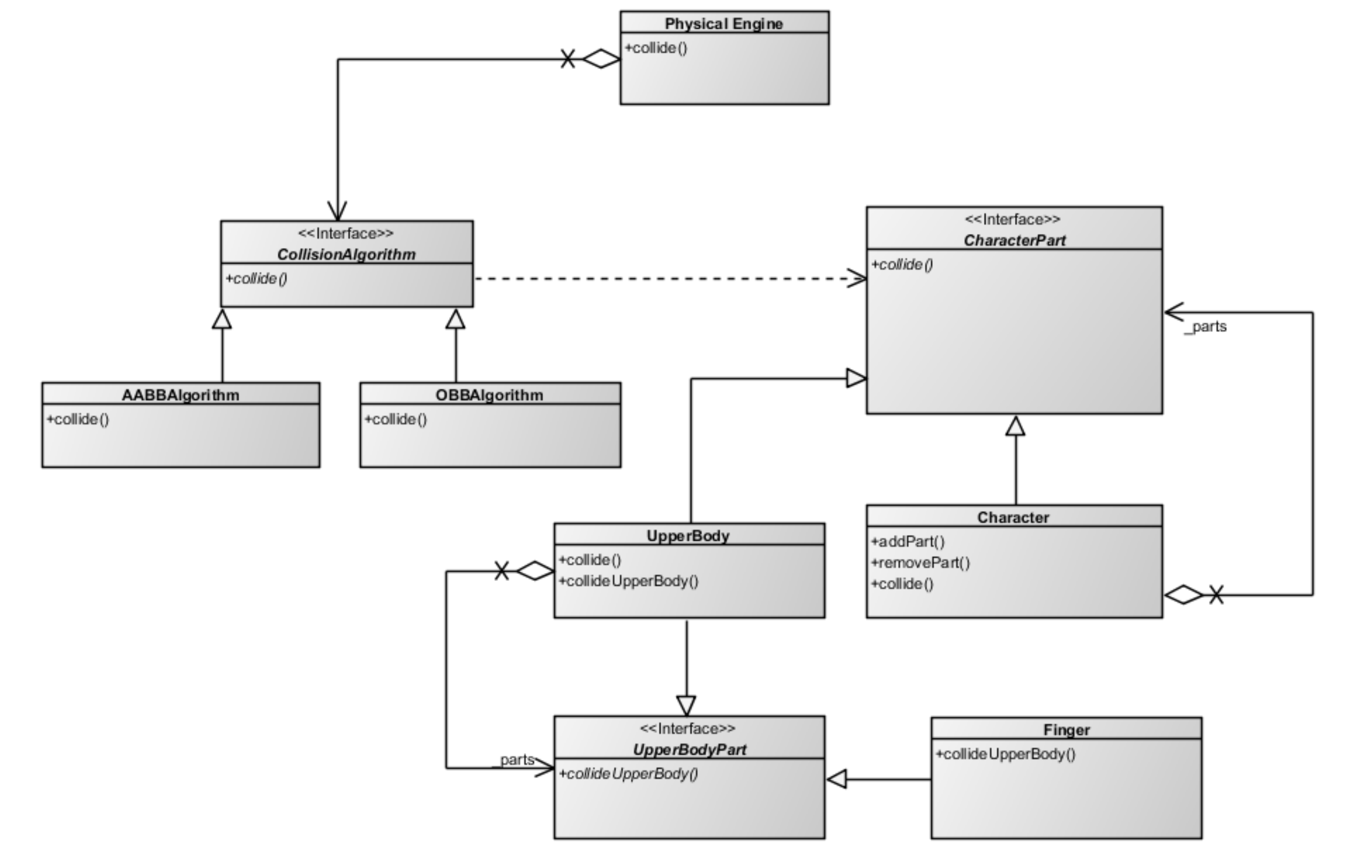
GoF DP – Behavioural Pattern - Strategy

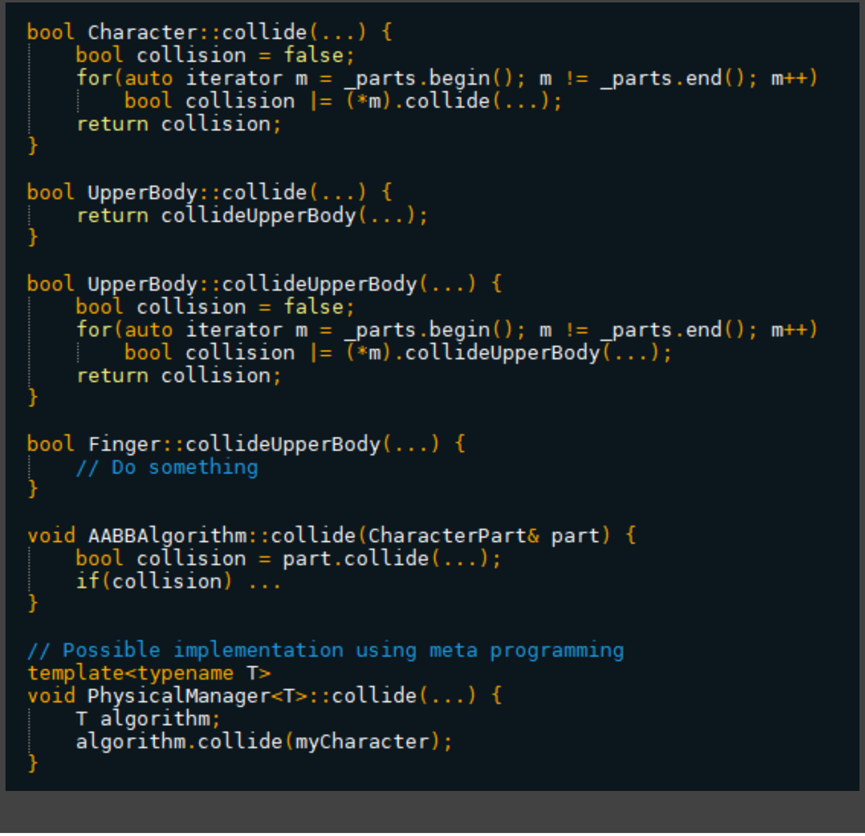




GoF DP – Behavioural Pattern - Strategy

- The Strategy pattern fits with our needs. We apply it.




```
bool Character::collide(...) {
    bool collision = false;
    for(auto iterator m = _parts.begin(); m != _parts.end(); m++)
        bool collision |= (*m).collide(...);
    return collision;
}

bool UpperBody::collide(...) {
    return collideUpperBody(...);
}

bool UpperBody::collideUpperBody(...) {
    bool collision = false;
    for(auto iterator m = _parts.begin(); m != _parts.end(); m++)
        bool collision |= (*m).collideUpperBody(...);
    return collision;
}

bool Finger::collideUpperBody(...) {
    // Do something
}

void AABBAlgorithm::collide(CharacterPart& part) {
    bool collision = part.collide(...);
    if(collision) ...
}

// Possible implementation using meta programming
template<typename T>
void PhysicalManager<T>::collide(...) {
    T algorithm;
    algorithm.collide(myCharacter);
}
```

Conclusion

Summary

- Patterns are the key of modern software engineering.
- A lot of successful applied solution to recurring well-known problems.
- Don’t try to recreate the wheel: Study them, learn them, apply them.

Summary

Design Patterns:

- Enable large-scale reuse of software architectures.
- Capture expert knowledge and design tradeoff.
- Give a concrete solution to non-functional problems when context is well-identified.

Summary

But:

- Be careful : Use design patterns everywhere when you can, don’t overuse them. Overdesign is worse than no design at all.
- DPs are not code. A pattern implementation can differ from one project to another. Elaborated patterns with bad implementation can lead to major flaws.
- Patterns are generic. They need more memory and are more expensive.

Sources

- « *Design Patterns* », Lecture from University Paris-Est - http://igm.univ-mlv.fr/ens/Master/M1/2018-2019/POO-DesignPatterns/cours/2-DP-A-intro.pdf
- « *Introduction to Design Pattern* », Sanae Bekkar - https://fr.slideshare.net/SanaeBEKKAR/introductiontodesignpattern
- « *CS 342 : Patterns and Framework* », David Levine - https://www.cse.msu.edu/~cse870/Materials/Patterns/Docs/patterns-frameworks-schmidt.pdf
- « *Design Pattern Catalog* », Regis Clouard - https://foad.ensicaen.fr/pluginfile.php/1214/course/section/633/DP-03.pdf?time=1566973225228
- « *Design Pattern* », Reda Bendraou - https://pagesperso.lip6.fr/Reda.Bendraou/IMG/pdf/cours3_gererlaqualite_design_patterns.pdf
- « *Design Pattern* », Gauthier Picard - https://www.emse.fr/~picard/cours/2A/DesignPatternsISI.pdf
- « *Object-Oriented Design Pattern* », Benoit Combemale - *https://www.irit.fr/~Benoit.Combemale/course/l3info/pattern-design.pdf*
- « *CSE 331 : Design Patterns* », Marty Stepp - http://www.cs.washington.edu/331/