Structural Design Patterns

Contents

1. Introduction 2
Adapter Design Pattern 2
How to build Adapter Design Pattern 2
Facade Design Pattern 6
Decorator Design Pattern 9
Towards Decorator - Step 1 9
Towards Decorator - Step 2 11
Towards Decorator - Step 3 11
Flyweight Design Pattern 14
Towards Flyweight 15
Another example: Text Editor with Character Objects 18

Code and Notes are @ https://github.com/nishithjain/Structural_Design_Patterns

1. Introduction

- Structural Design Patterns focus on how to organize and structure our code base.
- What all the classes/interfaces should exist in the codebase and how they relate to each other.
- What attributes/behavior should be there in a class/interface.

Adapter Design Pattern

- First let's understand what is an adapter. Taken an example of US-to-India Electrical Adapter.
    - An electrical adapter that converts a US plug to an Indian plug is a perfect real-world example of the Adapter design pattern.

- The Adapter Design Pattern enhances code maintainability when integrating with **third-party libraries or systems** by allowing incompatible interfaces to work together without modifying existing code.
- If our codebase has a tight coupling with 3rd party, then our codebase will not be maintainable (Violates DIP).
- Whenever we are connecting to a 3rd party, then never connect with them directly, always use an interface.

PhonePe, a popular digital payment platform in India, initially relied on Yes Bank for its banking services. However, due to certain reasons, PhonePe decided to migrate its banking operations to ICICI Bank. This transition involved integrating PhonePe's existing functionalities with ICICI Bank's APIs and systems.

- Problem Statement
    - We might need to change the 3rd party service providers.
    - If 3rd party goes down, then we will have to migrate to some other 3rd party.
- We can use Adapter design pattern to achieve this.

How to build Adapter Design Pattern

- Whenever we are connecting to a 3rd party service, create an interface with all the methods that we need from 3rd party.

```
class BankAPI {
public:
    virtual ~BankAPI() = default;
    virtual double CheckBalance(const std::string& account_number) = 0;
    virtual bool SendMoney(const std::string& from, double amount,
                 const std::string& to) = 0;
    virtual bool RegisterAccount(const std::string& account_number) = 0;
};
```

- "Yes Bank" won't implement these interfaces for us. We have to implement this interface and internally call the 3rd party "Yes Bank" APIs.
- Create a wrapper class which implements this BankAPI interface.

```
class YesBankAdapter final : public BankAPI
{
	std::unique_ptr<YesBankClient> yes_bank_client_;
public:
	explicit YesBankAdapter(std::unique_ptr<YesBankClient> client);
	double CheckBalance(const std::string& account_number) override;
	bool SendMoney(const std::string& from, double amount,
		const std::string& to) override;
	bool RegisterAccount(const std::string& account_number) override;
};
```

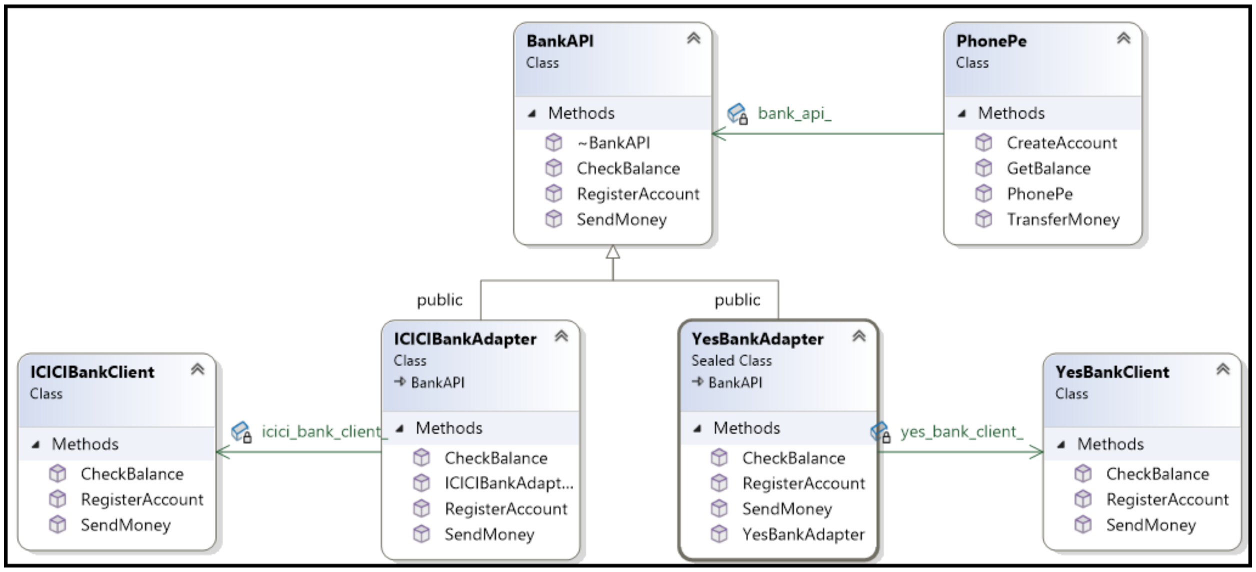
- Inside this method, we call the Yes Bank specific methods to get the balance, to transfer money and to register new account. Hence, we have a pointer pointing to the Yes Bank client.

```
YesBankAdapter::YesBankAdapter(std::unique_ptr<YesBankClient> client)
	: yes_bank_client_(std::move(client)) {
}

double YesBankAdapter::CheckBalance(const std::string& account_number) {
	return yes_bank_client_->CheckBalance(account_number);
}

bool YesBankAdapter::SendMoney(const std::string& from, const double amount,
	const std::string& to) {
	return yes_bank_client_->SendMoney(from, amount, to);
}

bool YesBankAdapter::RegisterAccount(const std::string& account_number) {
	return yes_bank_client_->RegisterAccount(account_number);
}
```

- PhonePe uses the BankAPI interface reference to check the balance, to transfer money and to register. Underneath BankAPI, it can be YesBankAdapter or ICICIBankAdapter.

```
class PhonePe
{
	std::unique_ptr<BankAPI> bank_api_;
public:
	explicit PhonePe(std::unique_ptr<BankAPI> api);
	double GetBalance(const std::string& account_number);
	bool TransferMoney(const std::string& from, double amount, const std::string& to);
	bool CreateAccount(const std::string& account_number);
};
```

```cpp
PhonePe::PhonePe(std::unique_ptr<BankAPI> api)
       : bank_api_(std::move(api)) {}

double PhonePe::GetBalance(const std::string& account_number) {
       return bank_api_->CheckBalance(account_number);
}

bool PhonePe::TransferMoney(const std::string& from, const double amount,
       const std::string& to)  {
       return bank_api_->SendMoney(from, amount, to);
}

bool PhonePe::CreateAccount(const std::string& account_number) {
       return bank_api_->RegisterAccount(account_number);
}
```



```cpp
auto bankClient = std::make_unique<YesBankClient>();
auto bankAdapter = std::make_unique <YesBankAdapter>(std::move(bankClient));
PhonePe phonePe(std::move(bankAdapter));

// Use PhonePe's methods
const double balance = phonePe.GetBalance("1234567890");
const bool transferSuccess = phonePe.TransferMoney("1234567890", 1000,"9876543210");
const bool registrationSuccess = phonePe.CreateAccount("9876543210");

std::cout << "Registration Done: " << registrationSuccess << "\n";
std::cout << "Transfer Done: " << transferSuccess << "\n";
std::cout << "Balance: " << balance << "\n";
```

- With this pattern, if PhonePe wants to move from “Yes Bank” to ICICI Bank, we just need to change 2 lines as highlighted above.

- When to Use the Adapter Design Pattern
    - **Incompatible Interfaces**: When you have existing classes or interfaces that you want to use but their interfaces don’t match the requirements of the system you are developing.
    - **Third-Party or Legacy Code Integration**: When integrating third-party libraries or legacy code with a different interface that you cannot modify directly.
    - **Simplifying Complex Interfaces**: When you need to simplify a complex interface for easier use by other parts of your system.
    - **Converting Data Formats**: When you need to convert data formats or protocols to make one interface compatible with another.

Facade Design Pattern

- The term "facade" refers to the exterior appearance or exterior wall of a building.
- Facade provides a cleaner/simplified view of a building by hiding complexities.

Requirement:

Build a Music Player system that allows users to load, play, pause, and stop music tracks. The system should also control the volume and display lyrics for the currently playing track.

- As per the requirement, we have to load the Music. So, let's create a class that loads the music track.

```
class MusicLoader {
public:
    void LoadMusic(const std::string& track);
};
```

- Let's create a class that can do play, pause and stop the music track.

```
class MusicPlayer {
public:
        void PlayMusic();
        void PauseMusic();
        void StopMusic();
};
```

- Let's create a class that can control the volume.

```
class VolumeControl {
public:
        void SetVolume(int level);
};
```

- Let's create a class the display lyric of the loaded sound track.

```
class LyricsDisplay {
public:
        void DisplayLyrics(const std::string& track);
};
```

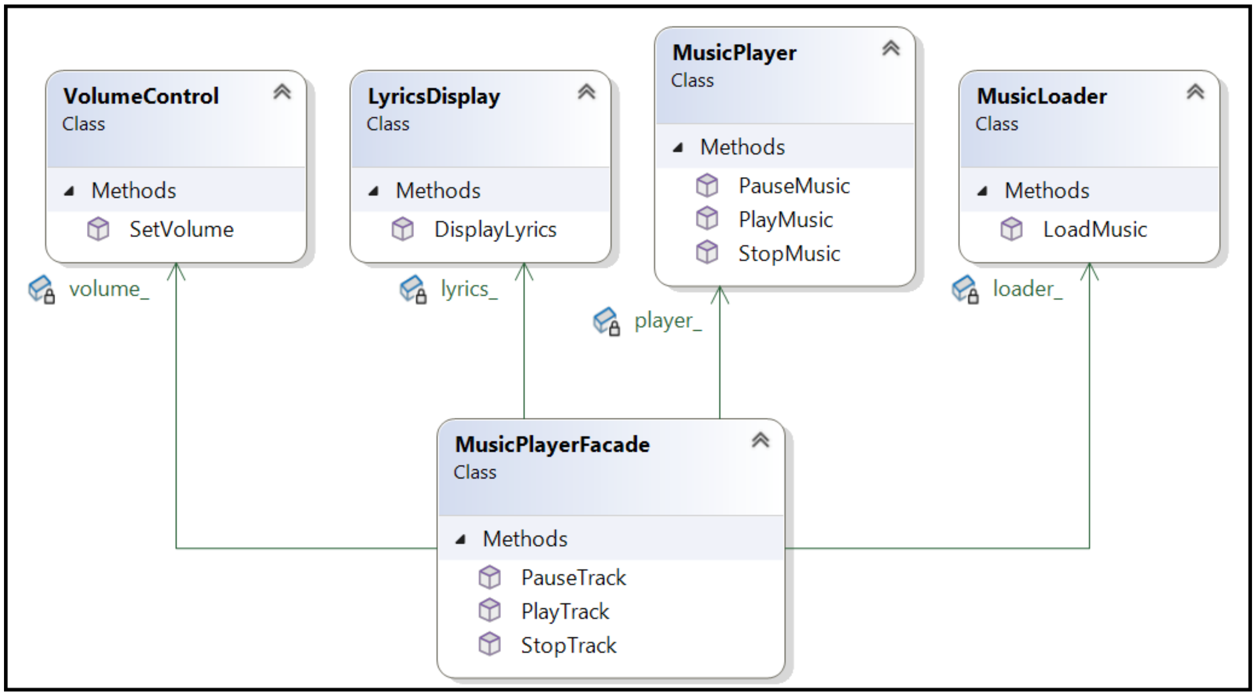
- Now the client code can directly interact with each subsystem as shown…

```
int main() {
    MusicLoader loader;   // Create instances of each subsystem
    MusicPlayer player;
    VolumeControl volume;
    LyricsDisplay lyrics;

    // Client must manage the sequence of calls and interactions
    const std::string track = "Bones by Imagine Dragons ";
    constexpr int volumeLevel = 7;

    loader.LoadMusic(track);         // Load the track
    volume.SetVolume(volumeLevel);   // Set the volume level
    player.PlayMusic();              // Play the music
    lyrics.DisplayLyrics(track);     // Display the lyrics
    player.PauseMusic();             // Pause the music
    std::cout << "Music has been paused.\n";
    player.StopMusic();              // Stop the music
    std::cout << "Music has been stopped.\n";
}
```

- Problem with this approach is…
    - **Tightly Coupled Client Code**: The client needs to understand the sequence of interactions among subsystems, making the code tightly coupled and harder to maintain.
    - **Complexity in Client Code:** The client is responsible for coordinating the subsystems, which adds complexity. The sequence of operations (e.g., load music, set volume, play, display lyrics) must be explicitly handled.
    - **Increased Maintenance**: Any changes in how the subsystems work would require modifications in the client code. If a new subsystem is added or an existing one is modified, the client code must be updated accordingly.

- Let's implement a Facade class that provides a simplified interface to interact with the music player system.

```
class MusicPlayerFacade {
    MusicLoader loader_;
    MusicPlayer player_;
    VolumeControl volume_;
    LyricsDisplay lyrics_;

public:
    void PlayTrack(const std::string& track, int volume_level);
    void StopTrack();
    void PauseTrack();
};
```



- Now, the client Interacts only with the Facade, not with individual subsystem classes, making it straightforward and easy to use.

```
int main() {

	MusicPlayerFacade musicPlayer;
	// Play a track with volume level 7
	musicPlayer.PlayTrack("Imagine by John Lennon", 7);
	musicPlayer.PauseTrack();     // Pause the track
	musicPlayer.StopTrack();      // Stop the track

	return 0;
}
```

- Benefits of the Facade Design Pattern in This Example
    - **Simplifies Complex Interactions**: The Facade consolidates multiple steps (loading, playing, setting volume, displaying lyrics) into simple method calls.
    - **Decouples Client Code**: The client is decoupled from the detailed operations of each subsystem, promoting a cleaner design.
    - **Easier Maintenance**: Changes in the underlying subsystems can be managed within the Facade without affecting the client code.

Decorator Design Pattern

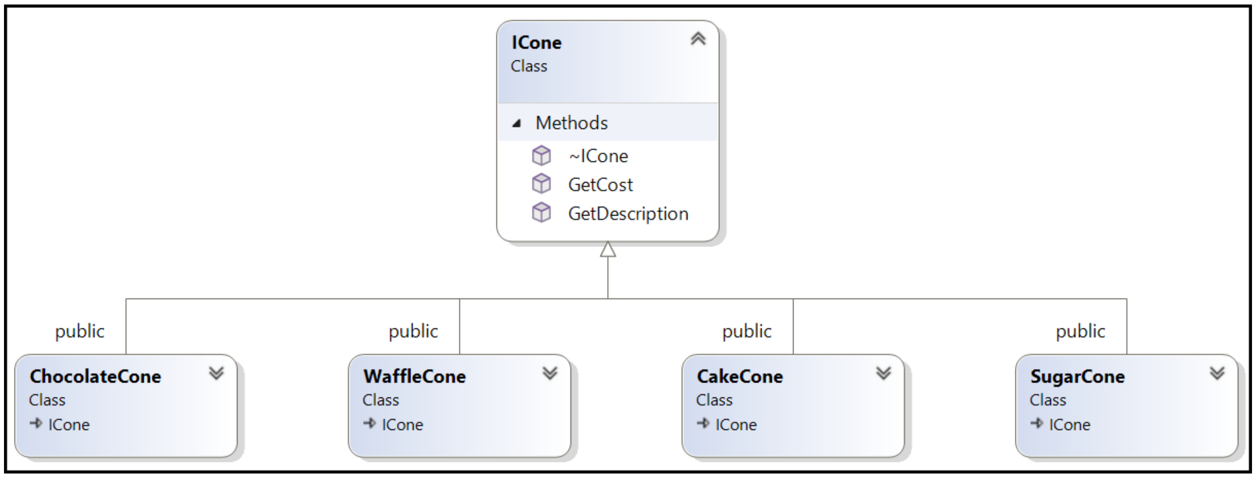
This design pattern is used building systems such as ice-cream management system, pizza management system, coffee management system, etc...

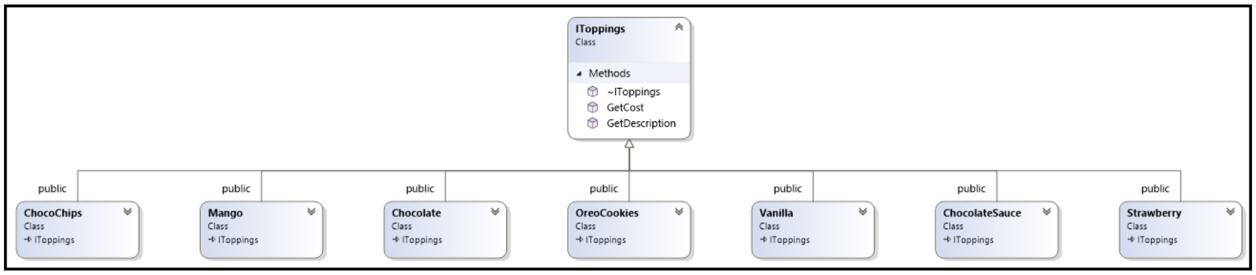
Problem Statement: Build an Ice-cream Order Management system that allows customers to customize their ice creams with various toppings, flavors, and sizes.

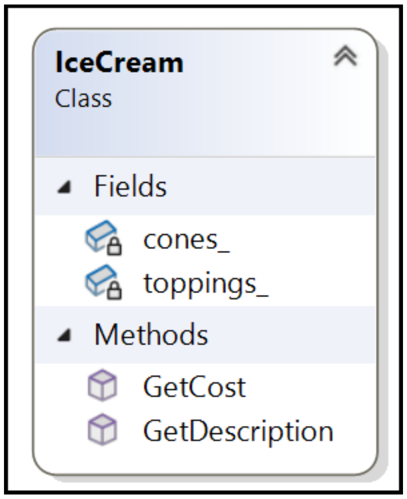
- Application that takes order for ice-cream.
- User can customize ice-cream.
- **Return total cost** of ice-cream.
- **Get the description** of ice-cream.
- Assume that **user cannot take scoop of ice-cream without cone**.

Towards Decorator - Step 1

- Let's have 2 interface `ICone` and `IToppings`.



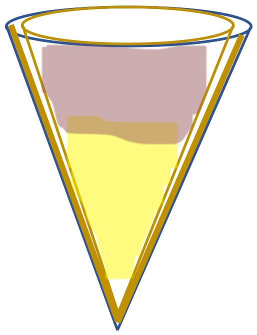




- Can we say total cost of the ice-cream is the summation of cost of every ingredient.
- Can we say description of ice-cream is the concatenation of description of every ingredient.
- Now, we can have an IceCream class with list of ICons and list of IToppings along with GetCost() and GetDescription() methods.
- To get the total cost in GetCost() method, we can iterate over list of cones_ and list of toppings_ and get the cost. We can do the same for description also.

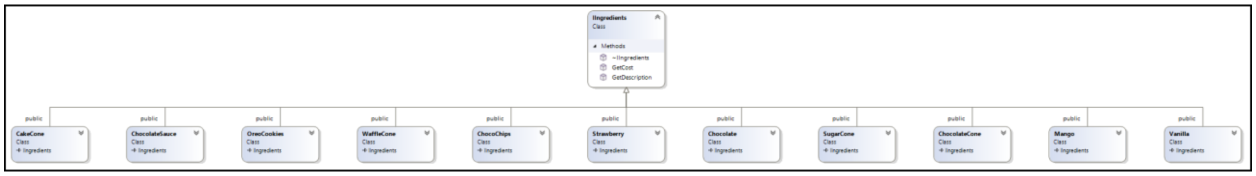
```
double IceCream::GetCost() const
{
	double totalCost = 0.0;
	for (const auto& i : cones_) {
		totalCost += i->GetCost();
	}
	for (const auto& i : toppings_) {
		totalCost += i->GetCost();
	}
	return totalCost;
}

std::string IceCream::GetDescription() const
{
	std::string description;
	for (const auto& i : cones_) {
		description += i->GetDescription();
	}
	for (const auto& i : toppings_) {
		description += i->GetDescription();
	}
	return description;
}
```

- But there is an issue with this approach.
  - If a user orders SugarCone + ChocolateSauce + WaffleCone + Mango + Chocolate.
  - Note that there are 2 Cones in the above order. In between 2 cones, user has asked for Chocolate Sause.
  - To get the description, if we use the above method, where we are first iterating over cones and then toppings, description will not be in the same order. Order will be SugarCone + WaffleCone + ChocolateSauce + Mango + Chocolate.
  - The description goes to the chef and he/she can mess up the order in which he/she can put the toppings.
- This method has a limitation.



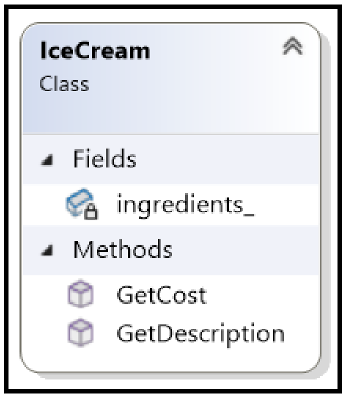
Towards Decorator - Step 2

- In the above solution, we had 2 interfaces which had same methods. If both of them have the same interfaces, we can have a single interface.
- Let's have a single interface and let's call it `IIngredients` and let all other classes implement this interface.



- In the `IceCream` class, instead of having 2 list, now we can have only 1 list namely .
- We can iterate over this list and can find the cost and description. Which this approach, the above issue of order mismatch will be solved.

```
double IceCream::GetCost() const
{
        double totalCost = 0.0;
        for (const auto& i : ingredients_) {
                totalCost += i->GetCost();
        }
        return totalCost;
}

std::string IceCream::GetDescription() const
{
        std::string description;
        for (const auto& i : ingredients_) {
                description += i->GetDescription();
        }
        return description;
}
```



- Ingredients may have dependencies or constraints (e.g., a cone is mandatory before any topping). The current approach, which merely stores them in a list, does not inherently enforce these rules.

Towards Decorator - Step 3

- Lot of children only eat the cone. An empty cone can be considered as `IceCream`. Say we have `CakeCone`.
- So, from the beginning we have an `IceCream`. Let's add `ChocolateSauce` to it and it can be considered as an `IceCream`. (`CakeCone` + `ChocolateSauce`)
- Let's user asked for one more cone, say `ChocolateCone`. Now we have `CakeCone` + `ChocolateSauce` + `ChocolateCone`. This can also be considered as `IceCream`.
- So, *right from the beginning we had an* `IceCream`*, and we can just keep adding the addons on top of that*.
- If we keep on adding the addons, cost and description will change.

If we have a scenario where we add properties or behaviors to an already existing object at runtime, consider using Decorator Desing Pattern.

1. Define an interface/abstract `class` for the entity that we are building. Let's call this interface as `IIceCream`.
2. Add the functionalities that we need for this `IIceCream` interface.
   a. `GetCost()`
   b. `GetDescription()`
3. We have 2 types of ingredients,
   a. Cone.
   b. Addons.
4. Some one can order an empty cone. So, cone can be considered **base entity or addon**. Similar to a pizza base. Pizza base can be considered as base entity.
5. For each addon, create a `class` that implements the `IIceCream` interface.
6. For the base entity which can exist independently,
   a. `GetCost()` -> Returns cost of itself.
   b. `GetDescription()` -> Returns description of itself.

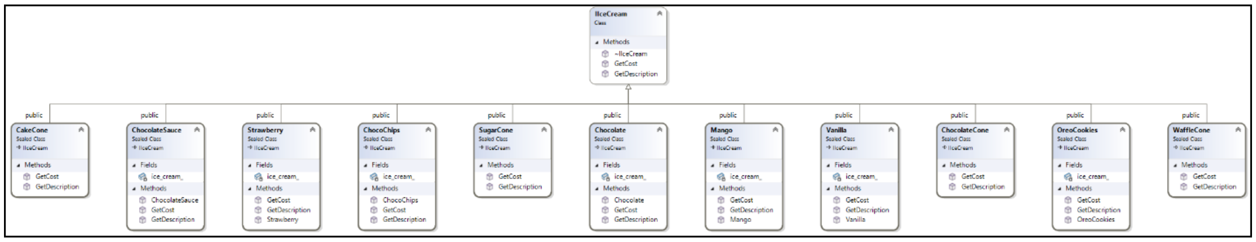
```
class CakeCone final : public IIceCream
{
public:
	double GetCost() override;
	std::string GetDescription() override;
};

double CakeCone::GetCost() {
	return 10.5;
}

std::string CakeCone::GetDescription() {
	return "Cake Cone";
}
```

7. For Addons, we need to have an `IceCream` first (can be an empty cone also). Remember we need to have a cone in order to have an addon.
   a. `GetCost()` -> Cost of existing `IceCream` + Cost of itself
   b. `GetDescription()` -> Description of existing `IceCream` + Description of itself.
      - To calculate the cost, we need existing `IceCream` cost. We need existing `IceCream` object which can be passed via **constructor injection**.
      - To get the description, we need the exiting `IceCream` description.

```cpp
#include "IIceCream.h"
class Mango final : public IIceCream
{
	std::unique_ptr<IIceCream> ice_cream_;
public:
	explicit Mango(std::unique_ptr<IIceCream> ice_cream);
	double GetCost() override;
	std::string GetDescription() override;
};


Mango::Mango(std::unique_ptr<IIceCream> ice_cream) :
	ice_cream_(std::move(ice_cream))
{
}

double Mango::GetCost()
{
	return ice_cream_->GetCost() + 25.0;
}

std::string Mango::GetDescription()
{
	return ice_cream_->GetDescription() + "," + std::string("Mango");
}
```

- Now if we want to server the order `"Chocolate Cone"` + `"Chocolate Sauce"` + `"Vanilla"` + `"Mango"` + `"Choco Chips"`, we can do...

```cpp
int main() {

	// Order "Chocolate Cone" + "Chocolate Sauce" + "Vanilla" + "Mango" + "Choco Chips"
	const std::unique_ptr<IIceCream> ic =
		std::make_unique<ChocoChips>(
		   std::make_unique<Mango>(
		       std::make_unique<Vanilla>(
			       std::make_unique<ChocolateSauce>(
				       std::make_unique<ChocolateCone>())))));

	std::cout << "Total Cost: " <<ic->GetCost() << "\n";
	std::cout << "Description: " << ic->GetDescription() << "\n";
	return 0;
}
// Total Cost: 72
// Description: Chocolate Cone, Chocolate Sauce, Vanilla, Mango, Choco Chips
```



Flyweight Design Pattern

Problem Statement: Imagine that we need to build UI of an online gaming application like BGMI.

- There will be 100 players in game lobby. Initially, entire state of the game will be downloaded on the device.
- Hence, starting of game takes some time.
- After that, multiple types of events will be triggered, for example if a player fires a bullet, bullet firing event will be triggered.
- Breakdown of the main entities in BGMI Game...
    1. Player
    2. NPC(Bots)
    3. Weapons
    4. Vehicles
    5. Buildings
    6. Loot Crates
    7. Airdrop
    8. Bullets
    9. Healings
- Let’s take **Bullets** for analysis.
    1. Number of players: 100
    2. Number of guns per player: 2
    3. Maximum ammo per gun: 180 bullets + 100 Backup
- Now, let's calculate the total number of bullets:
    1. **Total guns in the game**: 100 players × 2 guns per player = 200 guns
    2. **Total bullets per gun**: 180 bullets + 100 Backup
    3. **Total bullets in the game**: 200 guns × 280 bullets per gun = **56,000 bullets**
- We can say that number of Bullet objects will be much higher than all other entities in the game.
- Let’s see Bullet class with important attributes that capture its behavior and properties.

```
class Bullet
{
      // Position coordinates (x, y, z) in the game world
    double position_x_;
    double position_y_;
    double position_z_;

    // Direction vector
    double direction_x_;
    double direction_y_;
    double direction_z_;

    // Bullet attributes
    double velocity_;            // Speed of the bullet
    double damage_;              // Damage dealt on impact
    double range_;               // Maximum range before disappearing
    double gravity_effect_;      // Effect of gravity on the bullet
    double lifespan_;            // Time duration bullet exists before despawning
    std::string caliber_;        // Bullet type, e.g., "5.56mm", "7.62mm"
    std::string hit_effect_;     // Visual effect on impact
    std::string trail_effect_;   // Visual trail effect
    Image image_;
};
```

- Let's calculate the memory used by 1 bullet object.

Member Variable	Data Type	Size (bytes)
`position_x_`	`double`	8
`position_y_`	`double`	8
`position_z_`	`double`	8
`direction_x_`	`double`	8
`direction_y_`	`double`	8
`direction_z_`	`double`	8
`velocity_`	`double`	8
`damage_`	`double`	8
`range_`	`double`	8
`gravity_effect_`	`double`	8
`lifespan_`	`double`	8
`caliber_`	`std::string`	32
`hit_effect_`	`std::string`	32
`trail_effect_`	`std::string`	32
`image_`	`Image`	1000
	Total	1184

- Approximate size of 1 bullet object will be 1.1Kb.
- So, total size of all bullet objects in a game will be ≈ 56000 * 1.1Kb = **61.6 MB**.
- We need **61.6 MB** of RAM just to store bullet objects and apart from bullet objects, there will be other objects also.
- This is not a Good Approach.

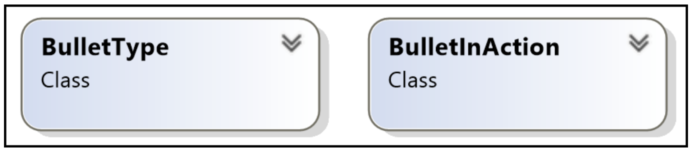
Towards Flyweight

- For some classes, we have 2 types of attributes.
    1. Extrinsic Properties.
        - Extrinsic properties are the values which might change over the time.
    2. Intrinsic Properties.
        - Intrinsic properties are the those whose values won't change over the time.
- Based on the definitions of intrinsic and extrinsic properties, we can classify the bullet class attributes as shown below
- Intrinsic Properties (Immutable, Inherent to the Bullet):
    - `caliber_` - The type of the bullet (e.g., "5.56mm") defines its inherent characteristics.
    - `hit_effect_` - The visual effect shown upon impact.
    - `trail_effect_` - The visual trail left behind by the bullet; specific to the bullet type.
    - `damage_` - The amount of damage a bullet can cause on impact.
    - `range_` - The maximum range the bullet can travel; determined by its type and design.
    - `image_` - The image or texture representing the bullet visually; tied to the specific bullet type.

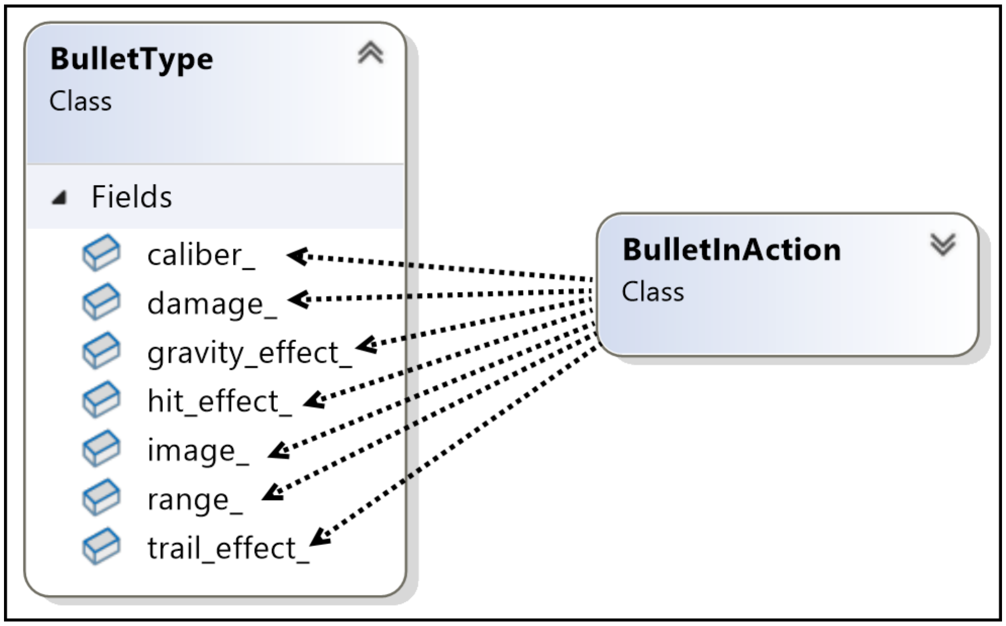
  - gravity_effect_ - The effect of gravity on the bullet, which depends on its design and type.
- Extrinsic Properties (Mutable, Context-Dependent):
  - position_x_, position_y_, position_z_ - The current coordinates of the bullet in the game world; change as the bullet moves.
  - direction_x_, direction_y_, direction_z_ - The current direction in which the bullet is traveling; changes during bullet movement.
  - velocity_ - The current speed of the bullet; can change due to environmental factors like drag or gravity.
  - lifespan_ - The remaining time before the bullet despawns; changes over time as the bullet exists.

If we have a class which demonstrates both intrinsic and extrinsic properties and we notice lot of memory is getting consumed because of lot of these objects, consider using **Flyweight design pattern**.

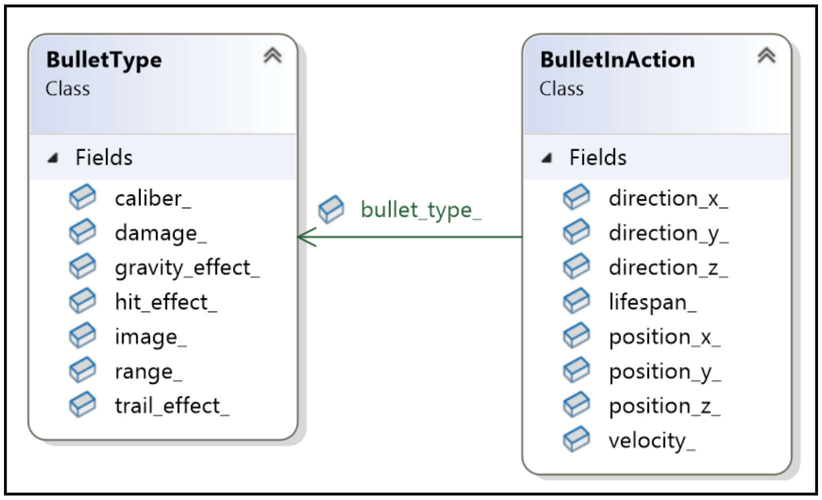
- How Flyweight Works
  - Divide the class into 2 classes.
  - Class with only intrinsic properties.
  - Class with only extrinsic properties.



- In the game, we create objects of BulletInAction class. But every object of BulletInAction class should have intrinsic properties present in BulletType class…



- To refer all the intrinsic properties which are present in the `BulletType`, we can store a reference of `BulletType` object inside `BulletInAction`.



```
class BulletType {
    // Intrinsic attributes
    double damage_;
    double range_;
    double gravity_effect_;
    std::string caliber_;
    std::string hit_effect_;
    std::string trail_effect_;
    Image image_;
public:

};
```

```
class BulletInAction {
    // Extrinsic attributes
    double position_x_;
    double position_y_;
    double position_z_;
    double direction_x_;
    double direction_y_;
    double direction_z_;
    double velocity_;
    double lifespan_;

    std::shared_ptr<BulletType> bullet_type_;
public:

};
```

- Now let's calculate the size of 1 `BulletInAction` object.

Member Variable	Data Type	Size (bytes)
`position_x_`	`double`	8
`position_y_`	`double`	8
`position_z_`	`double`	8
`direction_x_`	`double`	8
`direction_y_`	`double`	8
`direction_z_`	`double`	8
`velocity_`	`double`	8
`lifespan_`	`double`	8
`bullet_type_`	`std::shared_ptr<BulletType>`	16
	Total	80

- So, total size of all bullet objects in a game will be ≈ 56000 * 80 Bytes = **4.3 MB**.
- We have reduced the memory usage from **61.6 MB** to **4.3 MB**.

Another example: Text Editor with Character Objects

Context: In a text editor, every character displayed on the screen is an object.

- Without optimization, each character could have its own attributes like font, size, color, etc.
- If each character object is stored individually with all its attributes, this can lead to excessive memory usage, especially in large documents.
    - For example, the intrinsic state could include the **character shape**, while the extrinsic state could be the character's **position**, **font size**, or **color**.
- Imagine there is a large text document and each character is an object for text editor. If we are storing font, size, text formatting (italic, bold, underlined), color, etc. then it will consume lot of memory.
- Flyweight Pattern can be implemented to manage character objects efficiently by sharing common state.

```
// Flyweight class
class Character {
	char symbol_;
public:
	explicit Character(const char sym) : symbol_(sym) {}
	void Display(const int font_size, const int position_x,
		const int position_y) const
	{
		std::cout << "Character: " << symbol_ << " at (" << position_x
			<< ", " << position_y << "), Font Size: " << font_size << '\n';
	}
};
```

- The `Character class` represents a single character with intrinsic state (the character symbol itself).
- This `class` defines how a character is displayed using `Display()` method which displays the character on the screen with its associated extrinsic state (`font size`, `position`).

- The `CharacterFactory class` is responsible for managing the creation and sharing of `Character` objects.
- It ensures that there is only one instance of each unique character. For example, if character `'w'` appears `10000` times, there will be only 1 entry in the map.
    - **First Request**: The `CharacterFactory` checks if `'w'` is in the `characters_` map. Since it's not, it creates a new `Character` instance for `'w'`, adds it to the map, and returns a `std::shared_ptr<Character>` to this instance.

- **Subsequent Requests**: For each subsequent request for `'w'`, the `CharacterFactory` finds the existing entry in the `characters_` map and returns the same `std::shared_ptr<Character>`.

```cpp
// Character factory
class CharacterFactory {
	std::unordered_map<char, std::shared_ptr<Character>> characters_;
public:
	std::shared_ptr<Character> GetCharacter(char symbol)
	{
		if (characters_.find(symbol) == characters_.end())
		{
			characters_[symbol] = std::make_shared<Character>(symbol);
			std::cout << "Creating new character: " << symbol << '\n';
		}
		return characters_[symbol];
	}
};
```