Low Level Design

Contents

`Notes and code is at` https://github.com/nishithjain/solid_principles_cpp

1. Introduction

Let's understand what HLD first is.

- HLD (High-Level Design) is a design process in software development that provides an overview of the system's architecture and components without delving into detailed specifics. It focuses on the broader structure of the system, including the major modules, their interactions, and the technologies that will be used.
- Key Aspects of HLD:
    - **Architecture Overview**: Describes the system's overall architecture, including major components (e.g., servers, databases, APIs) and their interactions.
    - **Module Descriptions**: High-level descriptions of the main modules or components of the system and their responsibilities.
    - **Data Flow**: Illustrates how data flows between different components or modules, often represented using data flow diagrams.
    - **Technology Stack**: Outlines the technologies, frameworks, libraries, and platforms that will be used.
    - **Interfaces and Protocols**: Specifies how different components will interact, including APIs, communication protocols, and data formats.
    - **Integration Points**: Identifies how the system will integrate with external systems or third-party services.
    - **Security Considerations**: Addresses high-level security requirements and mechanisms, such as authentication, authorization, and data encryption.
    - **Scalability and Performance**: High-level strategies for ensuring the system can scale and perform efficiently under expected loads.
- Purpose of HLD:
    - Provides a clear blueprint of the system architecture that guides developers, architects, and stakeholders.
    - Facilitates communication among team members, allowing them to understand the overall structure of the project.
    - Helps in identifying potential challenges early in the design phase, allowing for adjustments before detailed design and implementation.
- HLD is typically followed by Low-Level Design (LLD), which goes into the specifics of individual components, including algorithms, data structures, and implementation details.
- So HLD is the process of understanding and designing how different infrastructure layers work together to ensure that the entire system functions efficiently.

Why Low-Level Design?

- Research says that 12% of the time is spent writing code, the remaining 88% is typically spent on various other activities that are crucial to the software development process.
    - Design and Planning (15-20%):
        - High-Level Design (HLD) and Low-Level Design (LLD).
        - Requirement gathering, analysis, and documentation.
        - Architectural and technical design discussions.
    - Testing and Debugging (25-30%):
        - Writing and running test cases (unit testing, integration testing, etc.).
        - Debugging issues and fixing bugs.
        - Performance testing and code reviews.
    - Code Review and Refactoring (10-15%):
        - Reviewing code written by team members for quality, efficiency, and adherence to standards.
        - Refactoring existing code to improve readability and performance.
    - Collaboration and Meetings (10-15%):
        - Daily stand-ups, sprint planning, retrospectives, and other team meetings.
        - Discussions with stakeholders, designers, or product owners to clarify requirements.
    - Documentation (5-10%):
        - Writing technical documentation, user manuals, and API documentation.
        - Updating design documents as the project evolves.
    - Research and Learning (5-10%):
        - Researching new technologies, libraries, and tools.
        - Learning best practices or exploring solutions to technical challenges.
    - DevOps and Deployment (5-10%):
        - Setting up and maintaining CI/CD pipelines.
        - Managing deployments, monitoring systems, and handling infrastructure-related tasks.
- LLD (Low-Level Design) helps optimize the development process by providing detailed guidance, which improves efficiency during the **88% of the time spent on activities other than writing code**.
- LLD Helps
    - Readability/Understandability
    - Maintainability
        - Bug identification and fixing
        - Platform Updates
        - Library Updates
    - Extensibility

OOP

Main Principle of OOP:

- Abstraction
    - Definition: Abstraction involves **hiding complex implementation** details and showing only the essential features of an object. It helps simplify complex systems by exposing only the necessary parts of an object to the user.
    - Purpose: Allows developers to focus on high-level operations without getting bogged down by intricate details, making it easier to manage and use software components.
    - Example: In a `Vehicle class`, you may have a `startEngine()` method. The user knows that calling this method will start the engine but doesn't need to know the complex details of how the engine is started internally.

3 Pillars of OOP which supports Abstraction are

- Encapsulation
    - Definition: Encapsulation is the concept of bundling data (attributes) and methods (functions) that operate on the data into a single unit, typically a class. It also restricts direct access to some of the object's components, which is why only necessary parts of an object are exposed.
    - Purpose: Protects the internal state of an object and ensures data integrity by controlling access through public methods (getters and setters).
    - Example: A `class BankAccount` has private attributes like balance and methods like `deposit()` and `withdraw()`. The internal balance cannot be accessed directly; it must be manipulated through these methods.
- Inheritance
    - Definition: Inheritance allows a new class to inherit properties and behaviors (methods) from an existing `class`. This promotes code reuse and establishes a parent-child relationship between `classe`s.
    - Purpose: Enables the creation of a new `class` with minimal modification by reusing the existing code, leading to less redundancy and easier maintenance.
    - Example: A `Vehicle class` can be the parent `class` of `Car` and `Bike classe`s. Both `Car` and `Bike` inherit common properties like `engine` and methods like `start()` from the `Vehicle class` but can also have their own specific features.
- Polymorphism
    - Definition: Polymorphism allows objects of different classes to be treated as objects of a common superclass. It also enables a single function or method to work in different ways based on the object it is acting upon.
    - Purpose: Provides flexibility and extendibility in code, allowing methods to be overridden or overloaded to suit specific object types.
    - Example: A `Shape class` with a method `draw()`. The `Circle` and `Rectangle classe`s, which inherit from `Shape`, can implement their own versions of `draw()`, providing specific behavior when called.

2. SOLID Design Principle

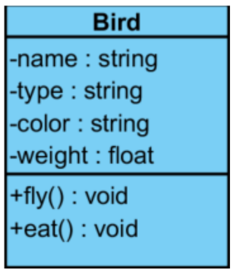
- Single Responsibility Principle
- Open Closed Principle
- Liskov substitution Principle
- Interface Segregation Principle
- Dependency inversion Principle

Single Responsibility Principle

Problem statement: Design a Bird

- We must build a software system that will store the information of all the different type of Birds.

Version 1



```
class Bird {
	string name;
	string type;
	string color;
	float weight;

public:
	Bird(string name, string type,
		string color, float weight);
	void fly();
	void eat();
};
```

```
// If we are not using the OOP principles,
// this is how the fly method looks like
void Bird::fly()
{
	if (type == "Domestic")
		cout << "Flying low\n";
	else if (type == "Wild")
		cout << "Flying high\n";
}
```

- The `Bird::fly()` above is it…
  - Maintainable?
    - The above `Bird::fly()` method is **hard to maintain**. The reason for that is, if are making 1 change in this method, we have to test the entire method.
    - Assume we have 50 birds, we will be having 50 `if-else` cases, this means, we have to test 50 different types of behaviors.
  - Transparent?
    - The above `Bird::fly()` method is easily understandable? No. Reviewing is difficult.

        - If you give a developer to review 10 lines of code, he will find 10 mistakes. If you give the same developer to review 1000 lines, he will say everything is okay. Noone wants to review 1000s of line
    - Extensible?
        - The above `Bird::fly()` method is easily extensible? Can we add a new `Bird`? No. The reason for that is, if we are adding a new `Bird`, we need to change this method, this means, we need to test all the 50 different behaviors.
        - The `Bird::eat()` might also have 50 different `if-else` cases. If we want to add a new `Bird`, then this also needs to be modified.
- The root cause of the method `Bird::fly()` not being maintainable, transparent and extensible?
    - **The single method `Bird::fly()` is responsible for 50 different types of behaviors.**
- Ideally every method, every class, every interface, every package should be responsible for 1 and exactly 1 type of behavior.

  This is principle is called as Single Responsibility Principle
- How to fix the above code?
    - **We can make use of inheritance and create separate `class`es for each of the `Bird`s.**
- Another example of vialoation of SRP.

```java
public void save() {
    try {
        Employee employee = this;
        StringBuilder sb = new StringBuilder();
        sb.append("### EMPLOYEE RECORD ####");
        sb.append(System.lineSeparator());
        sb.append("NAME: ");
        sb.append(employee.firstName + " " + employee.lastName);
        sb.append(System.lineSeparator());
        sb.append("POSITION: ");
        sb.append(employee.getClass().getTypeName());
        sb.append(System.lineSeparator());
        sb.append("EMAIL: ");
        sb.append(employee.getEmail());
        sb.append(System.lineSeparator());
        sb.append("MONTHLY WAGE: ");
        sb.append(employee.monthlyIncome);
        sb.append(System.lineSeparator());
        Path path = Paths.get(employee.getFullName().replace(" ", "_") + ".rec");

        Files.write(path, sb.toString().getBytes());
        System.out.println("Saved employee " + employee.toString());
    } catch (IOException e) {
        System.out.println("ERROR: Could not save employee. " + e);
    }
}
```

- SRP Violation in the save Method:

The save method in the code provided violates the SRP because it handles multiple responsibilities:

1. Generating Employee Data Representation:
   - The method formats the employee's data into a human-readable string, which is a task related to data presentation.
2. File Path Construction:
   - It determines the file path where the data should be saved, which is related to file system management.
3. Writing Data to a File:
   - The method performs the actual file I/O operation by writing the data to a file, which is a responsibility related to data persistence.
4. Logging Output:
   - It logs a message to the console after saving, which is related to user interaction or logging.

- Consequences of the Violation:
  - If you need to change how the employee data is formatted, you must modify this method.
  - If the file writing mechanism changes (e.g., switching to a database), you must also modify this method.
  - Any changes in logging or file path generation would also affect this method.
- Refactoring for SRP Compliance:
  - To comply with SRP, you can refactor the code by splitting these responsibilities into separate methods or classes:
    - `EmployeeFormatter`: A class or method responsible for formatting employee data into a string.
    - `FileHandler`: A class or method responsible for saving data to a file.
    - `EmployeeSaver`: A class that uses the above components to save employee data, thereby orchestrating the process.
- Litmus test for SRP
  - Every code unit should ideally have 1 responsibility. **If there are multiple reasons to change the same method, this means, that methos is violating SRP**.
  - **Multiple if-else with Different Behavior**: Multiple if-else statements with different behaviors often indicate that a class or method has multiple responsibilities, violating SRP. The presence of multiple if-else or switch statements within a Factory is not a violation of SRP as long as their purpose is to control the instantiation process based on specific criteria.
  - **Large Method/Monster Method**: Large methods often violate SRP because they tend to handle multiple tasks or responsibilities, making the code harder to maintain and test.
  - **Unspecified Util/Helper Class**: Unspecified Util/Helper classes that contain unrelated functions violate SRP because they serve multiple purposes, making the class a "catch-all" for different responsibilities.

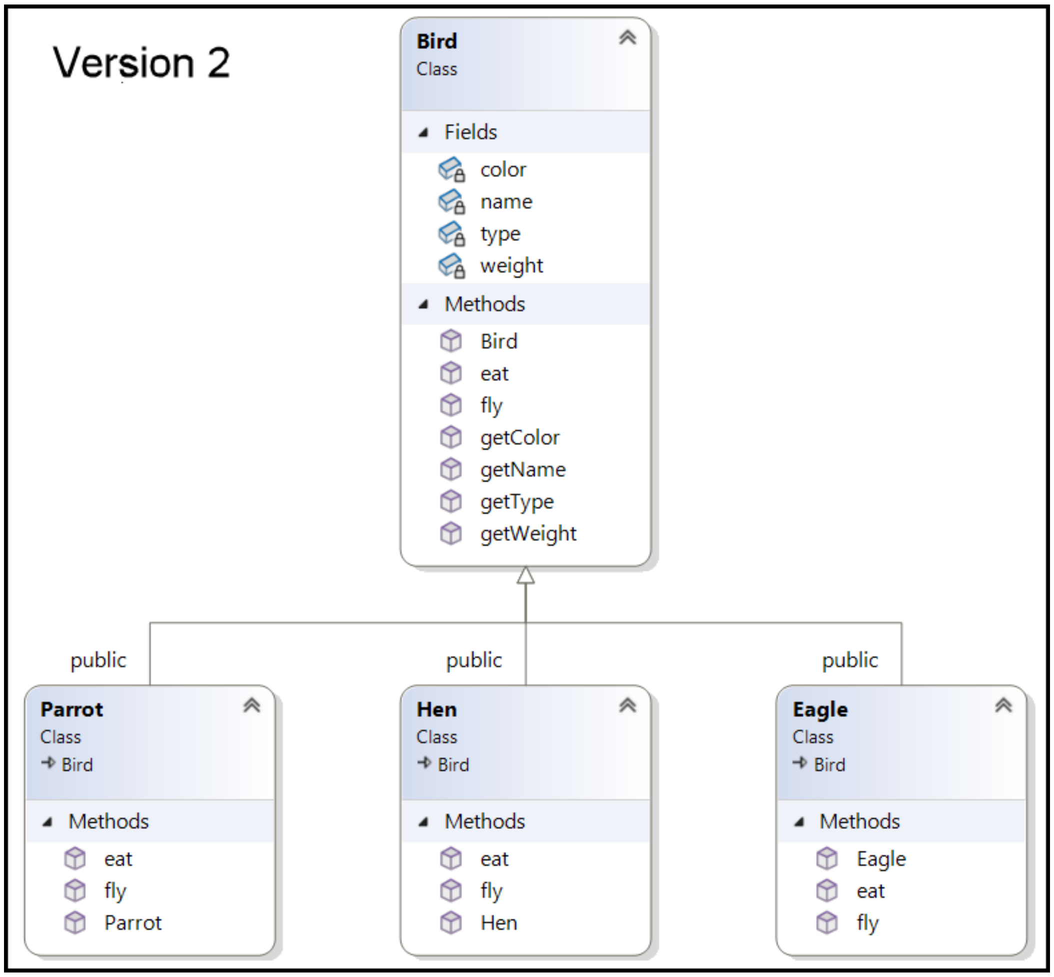
Open Closed Principle

"Software entities (such as classes, modules, functions) should be open for extension but closed for modification."

- Explanation:
    - **Open for Extension**: The behavior of a module, class, or function can be extended to meet new requirements or changes.
    - **Closed for Modification**: The existing code should not be modified when extending its functionality.
- The version 1 of the `Bird` `class` is a bad design. If we want to add a new `Bird`, say `Parrot`, we need to modify the existing `if-else` code.
- So, in the next version 2 of `Bird` `class`, let the `Bird` `class` have only the general attributes and behavior and for **every type of** `Bird`**, we should have separate** `class`.
- Since, the `Bird` itself cannot `fly`, we can declare abstract methods and the derived `class`es can implement these abstract methods.

```
class Bird {
	string name;
	string type;
	string color;
	float weight;

public:
	Bird(string name, string type,
		string color, float weight);

	string getName() const { return name; }
	string getType() const { return type; }
	string getColor() const { return color; }
	float getWeight() const { return weight; }

	// Abstract method: must be implemented by any derived class.
	// This pure virtual function makes the Bird class abstract.
	virtual void fly() = 0;

	// Abstract method: must be implemented by any derived class.
	// This pure virtual function makes the Bird class abstract.
	virtual void eat() = 0;
};
```

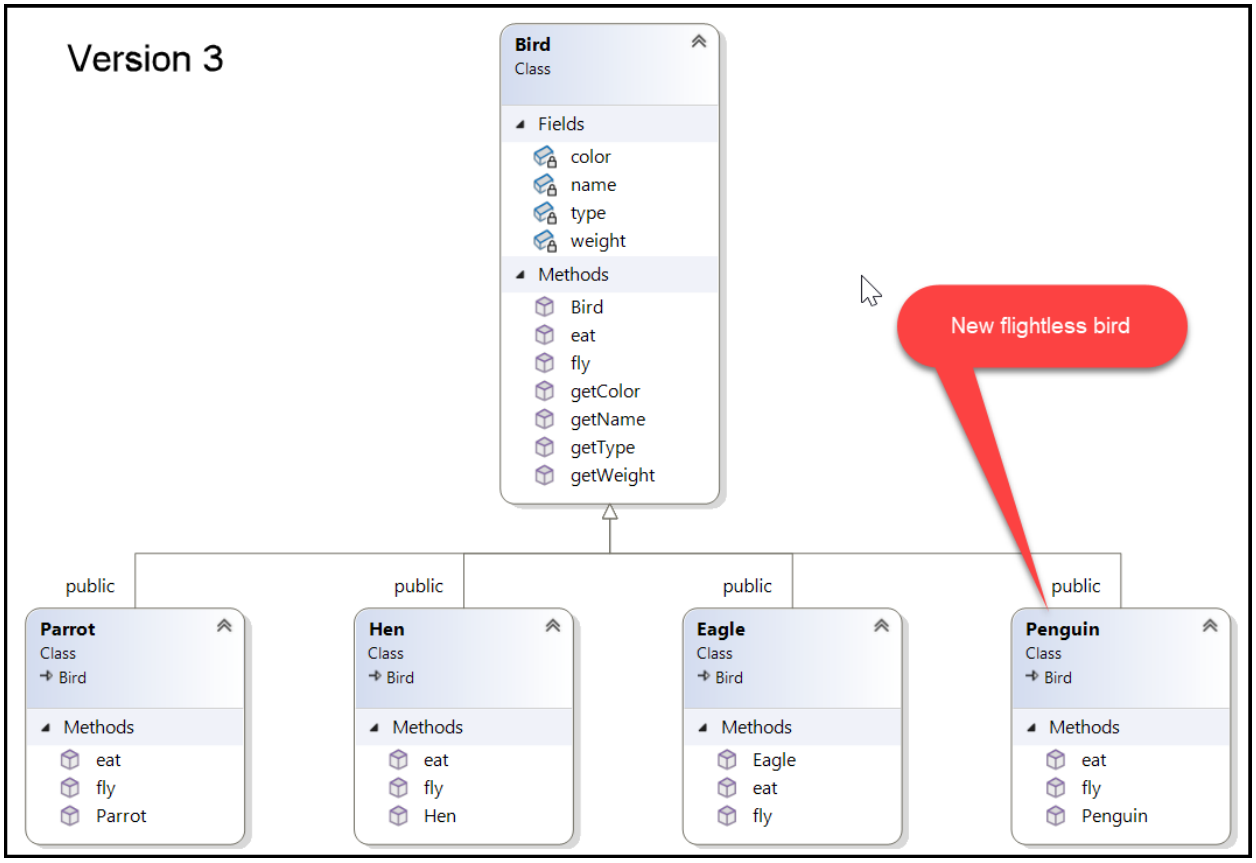
- To add a new type of bird `Parrot`, we need to create new `class Parrot` and implement `fly()` and `eat()` methods.
- Now, every bird class has single reason to change! SRP also followed!
- If we want to store data of all the Birds in a list, we can use `vector<unique_ptr<Bird>> birds`.
- Even though we will be having 50 different `class`es for 50 different `Bird`s, it enables parallel development.



```cpp
int main() {
	vector<unique_ptr<Bird>> birds;
	birds.push_back(make_unique<Hen>("Hen", "Domestic", "Brown", 2.5));
	birds.push_back(make_unique<Eagle>("Eagle", "Wild", "Golden", 6.0));
	birds.push_back(make_unique<Parrot>("Parrot", "Tropical", "Green", 1.0));

	for (const auto& bird : birds) {
		bird->fly();
		bird->eat();
	}
	return 0;
}
/***********************************
Hen is Flying low.
Hen is Eating insects.
Eagle is Flying high.
Eagle is Eating fish.
Parrot is flying with colorful wings.
Parrot is eating fruits and seeds.
***********************************/
```

Liskov Substitution Principle

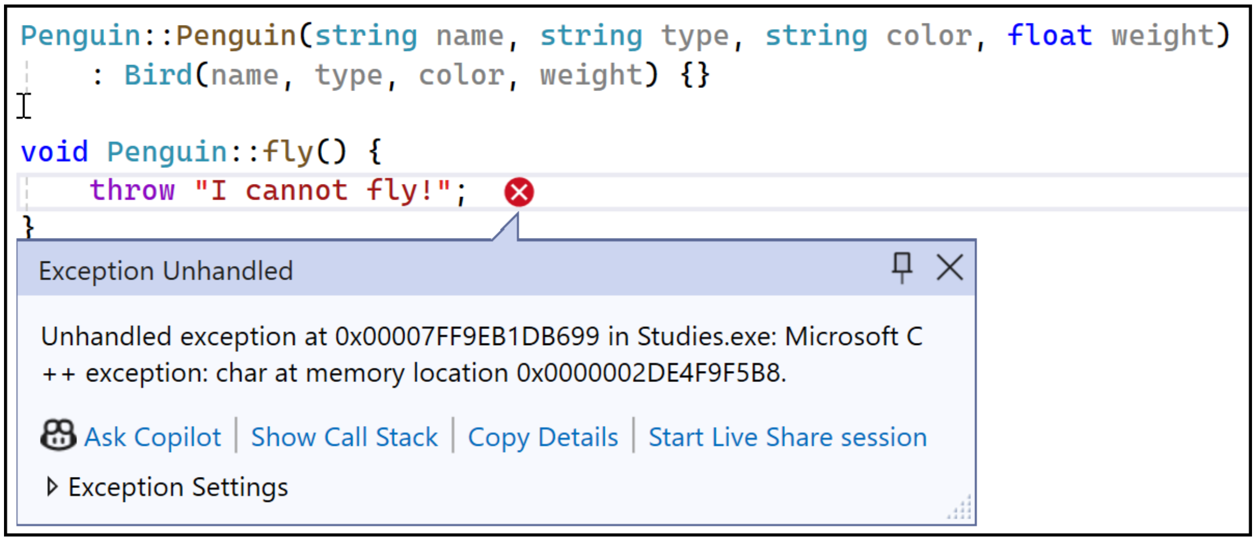
- Now a new requirement came where we need to add a new bird `Ostrich`, `Emu`, `Kiwi` and `Penguin`.
- Right now we have an abstract base `class Bird` which has a pure virtual function `fly()` and `eat()`, we must implement the `fly()` and `eat()` method in the new `Ostrich`, `Emu`, `Kiwi` and `Penguin` `class`es. **But these are flightless birds**!!!



- If we implement these methods, what do we write in that method?
  - Throw exception.
  - Return null.
  - Leave it blank.
- If client who is using our Bird class, does the following…

```
int main() {
	vector<unique_ptr<Bird>> birds;
	birds.push_back(make_unique<Hen>("Hen", "Domestic", "Brown", 2.5));
	birds.push_back(make_unique<Eagle>("Eagle", "Wild", "Golden", 6.0));
	birds.push_back(make_unique<Parrot>("Parrot", "Tropical", "Green", 1.0));
	birds.push_back(make_unique<Penguin>("Penguin", "Antarctic",
		"Black and White", 20.0));
	for (const auto& bird : birds) {
		bird->fly();
		bird->eat();
	}
	return 0;
}
```

- An exception is thrown…



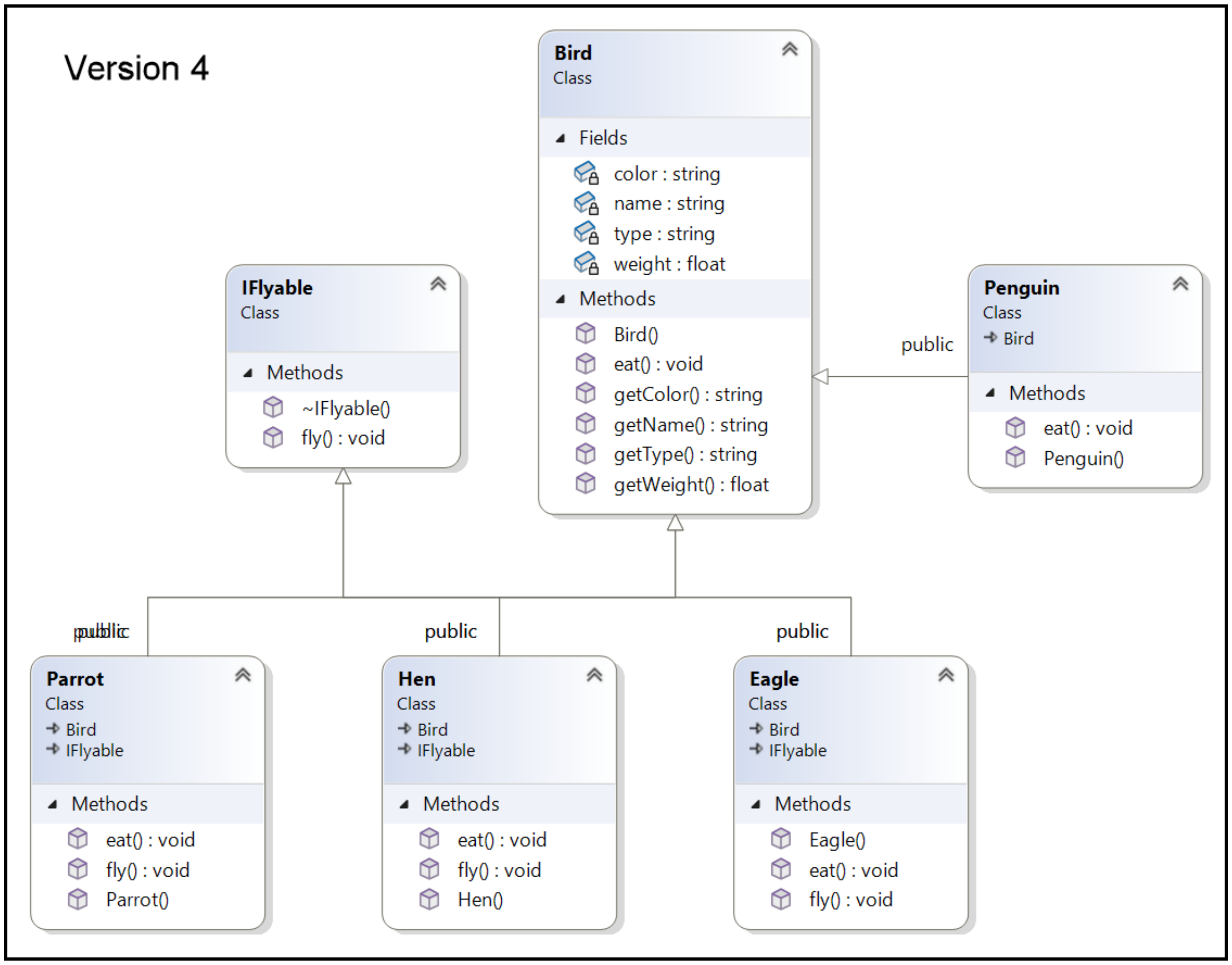
```
Penguin::Penguin(string name, string type, string color, float weight)
    : Bird(name, type, color, weight) {}

void Penguin::fly() {
    throw "I cannot fly!";
}
```

Exception Unhandled

Unhandled exception at 0x00007FF9EB1DB699 in Studies.exe: Microsoft C ++ exception: char at memory location 0x0000002DE4F9F5B8.

Ask Copilot | Show Call Stack | Copy Details | Start Live Share session
Exception Settings

- If there is any functionality that is present in parent `class`, that functionality must work for all the child `class`es. This is "Liskov substitution Principle". It states "**Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program.**"
- How to correct the above issue?
  - Some Birds demonstrate a **behavior** and some Birds doesn't demonstrate a **behavior**.
    - Only the Birds that demonstrate a **behavior** should have that method.
    - We should be able to create a list of Birds with a particular **behavior**.
    - `Class` => Blue print of Entity.
    - `Interface` => Blue print of Behaviors.
  - We must remove the `fly()` method from the base `class Bird`.
  - We can have an interface (`IFlyable`) and all the `Bird`s that can `fly()` must implement this interface.
  - `Bird`s that cannot fly need not implement this interface.



```
class Bird {
        string name;
        string type;
        string color;
        float weight;
public:
        Bird(string name, string type,
                string color, float weight);
        string getName() const { return name; }
        string getType() const { return type; }
        string getColor() const { return color; }
        float getWeight() const { return weight; }

        virtual void eat() = 0; // Removed Fly()
};

class IFlyable {
public:
    virtual void fly() = 0;
    virtual ~IFlyable() = default;
};
```

```cpp
#include "Bird.h"
#include "IFlyable.h"

class Eagle : public Bird, public IFlyable {
public:
	Eagle(string name, string type, string color, float weight);
	void fly() override; // Interface method
	void eat() override;
};


#include "Bird.h"
class Penguin : public Bird {
public:
    Penguin(string name, string type, string color, float weight);
    void eat() override;
};


int main() {
    vector<unique_ptr<Bird>> birds;
    vector<IFlyable*> flyableBirds; // Vector to store flyable birds

	birds.push_back(make_unique<Hen>("Hen", "Domestic", "Brown", 2.5f));
	birds.push_back(make_unique<Eagle>("Eagle", "Wild", "Golden", 6.0f));
	birds.push_back(make_unique<Parrot>("Parrot", "Tropical", "Green", 1.0f));
	birds.push_back(make_unique<Penguin>("Penguin", "Antarctic",
	     "Black and White", 20.0f));

    for (const auto& bird : birds) {
        bird->eat();
        auto flyableBird = dynamic_cast<IFlyable*>(bird.get());
        if (flyableBird) {
            flyableBird->fly();
            flyableBirds.push_back(flyableBird); // List of flyable birds
        }
    }

    for (const auto& flyableBird : flyableBirds) {
        auto bird = dynamic_cast<Bird*>(flyableBird);
        cout << bird->getName() << " is a flyable bird!\n";
    }
	return 0;
}
/***********************************
Hen is Eating insects.
Hen is Flying low.
Eagle is Eating fish.
Eagle is Flying high.
Parrot is eating fruits and seeds.
Parrot is flying with colorful wings.
Penguin is eating fish.
Hen is a flyable bird!
Eagle is a flyable bird!
Parrot is a flyable bird!

***********************************/
```

Interface Segregation Principle

- A new requirement came where we need to add 2 methods `flapWings()` and `takeOff()` to all the `Bird`s `class`es.
- Since these to methods `flapWings()` and `takeOff()` are related to `fly()`. Is it good idea to put these 2 methods in the `IFlyable` interface?
  - No, it is not good idea to keep these 2 methods in `IFlyable` interface even though they are related.
  - There can be other `class`es which are not `Bird` but can fly. Example, `Aeroplan`. Now this `Aeroplan` has to implement these 2 methods `flapWings()` and `takeOff()` even though `Aeroplan` cannot flap wings.
- `Aeroplan` `class` is forced to implement these 2 `flapWings()` and `takeOff()` methods if we keep these 2 methods in the `IFlyable` interface. This is **Interface Segregation Principle (ISP)**.
- The ISP states:
  - Clients should not be forced to depend on interfaces they do not use.
- Why ISP is Important:
  - **Decoupling**: By splitting interfaces, clients are only exposed to the methods they actually use, reducing unnecessary dependencies.
  - **Flexibility**: Smaller, focused interfaces make it easier to change or extend the system without affecting unrelated parts.
  - **Maintainability**: Changes in one interface won't impact other clients that are not using that part of the interface.
  - **Testability**: It becomes easier to create mock implementations for testing since each interface is concise and specific.

Another example of ISP

- We are asked to implement a printer that can do multiple functions such as printing, scanning, and fax. Since these are behaviors, we can implement an interface.

```
class IPrinter {
public:
    virtual void print() = 0;
    virtual void scan() = 0;
    virtual void fax() = 0;
};
```

- Problems:
  - A printer that can only print would still need to implement `scan()`, and `fax()`, even if those methods are not relevant.

- Applying ISP:
    - To follow ISP, break down the `IPrinter` interface into smaller, more specific interfaces:

```
// Specific interface for printing
class IPrintable {
public:
    virtual void print() = 0;
    virtual ~IPrintable() = default;
};

// Specific interface for scanning
class IScannable {
public:
    virtual void scan() = 0;
    virtual ~IScannable() = default;
};

// Specific interface for faxing
class IFaxable {
public:
    virtual void fax() = 0;
    virtual ~IFaxable() = default;
};
```

- Now the printer which can only print can be implemented as follows…

```
class BasicPrinter : public IPrintable {
public:
    void print() override {
        // Print implementation
    }
};
```

- The multi-function printer can be implemented as follows…

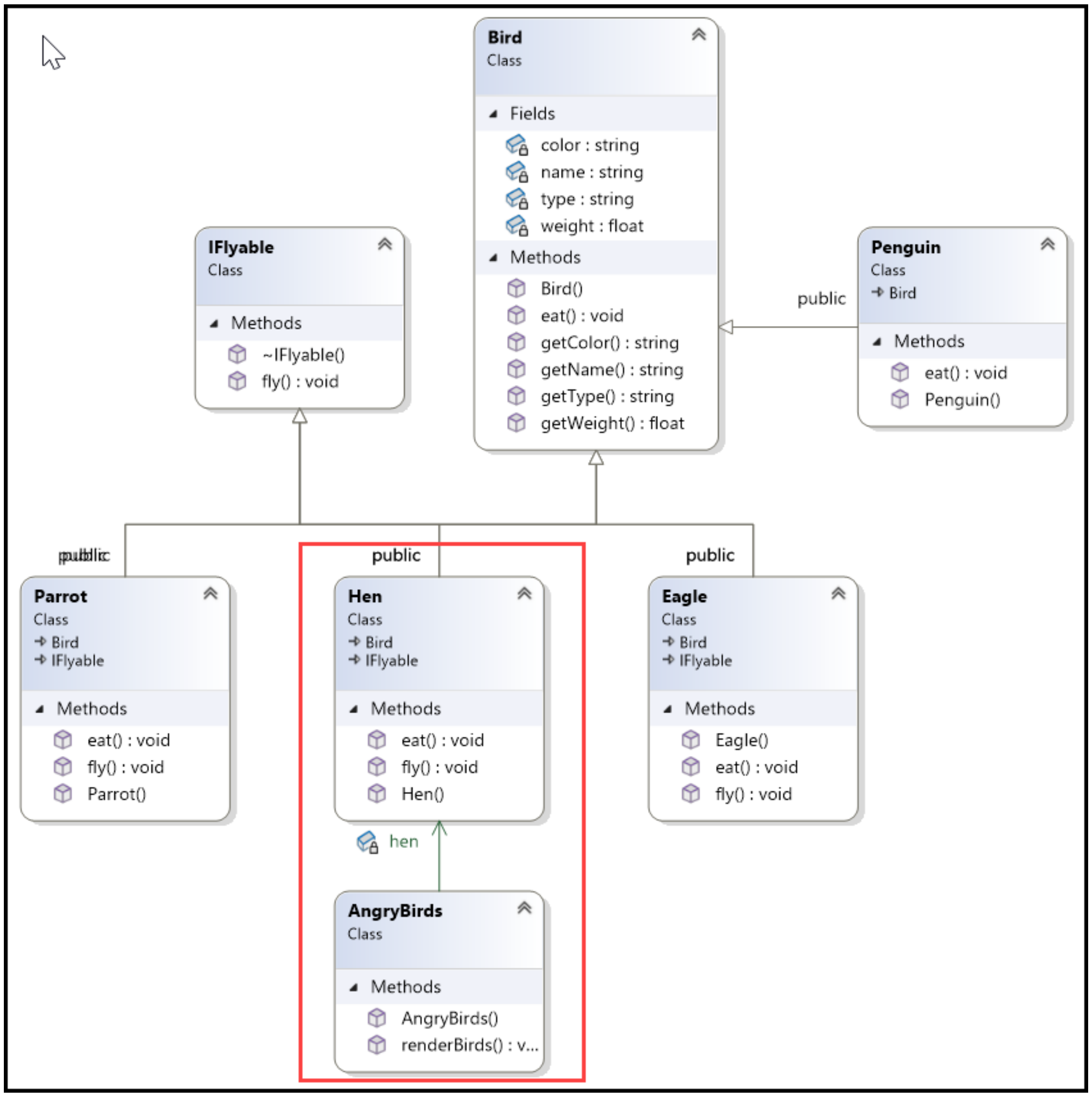
```
class MultifunctionPrinter : public IPrintable,
                             public IScannable,
                             public IFaxable {
public:
    void print() override {
        // Print implementation
    }
    void scan() override {
        // Scan implementation
    }
    void fax() override {
        // Fax implementation
    }
};
```

Dependency Inversion Principle

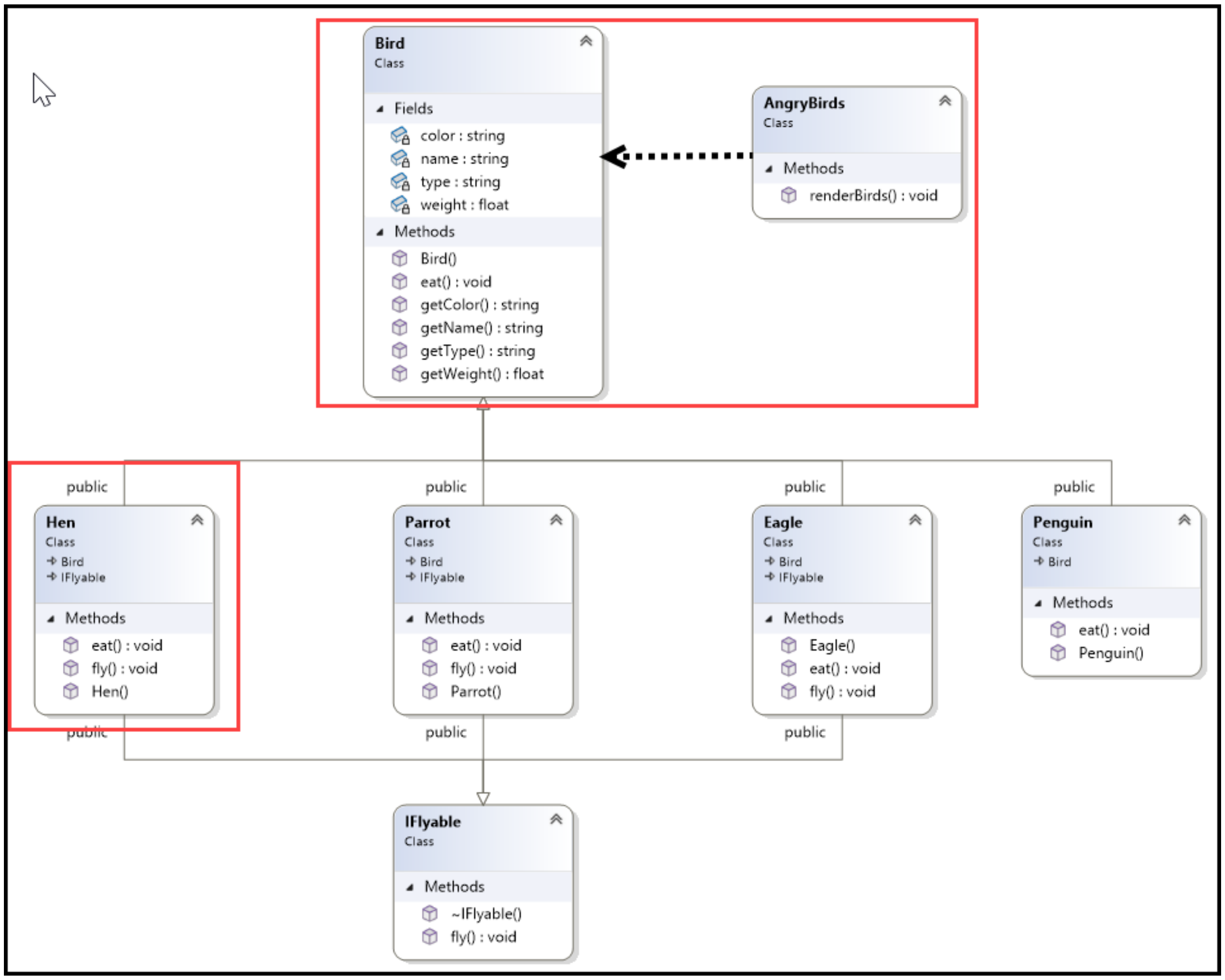
- Using the above Bird library, we are asked to create a Game, say AngryBirds…
- In AngryBirds game, we asked to render the Birds..
- We might develop as shown below…

```
class AngryBirds {
private:
	Hen hen;

public:
	AngryBirds(const string& name, const string& type,
		const string& color, float weight) :
		hen(name, type, color, weight)
	{
	}
	void renderBirds()
	{
		cout << "Our bird name is " << hen.getName() << " which is of "
			<< hen.getType() << " type and is " << hen.getColor()
			<< " in color.\n";
		hen.fly();
	}
};

int main() {
    AngryBirds angryHen("Ginger", "Domestic", "Brown", 2.5f);

    angryHen.renderBirds();

    return 0;
}

/***********************************
Our bird name is Ginger which is of Domestic type and is Brown in color.
Ginger is Flying low.
***********************************/
```

- The problem with the above code is tight coupling.
- If we want to render `Parrot` instead of `Hen`, we have to make changes in the `renderBirds()` function.



- To fix the issue, we need to make sure that, we never use 'new' in our code. This means that we are creating an object and hence dependency.
- To solve this issue, we need to inverse the dependency. Right now, the `AngryBirds class` depends on the `Hen class`. Instead of this, `AngryBirds class` and the `Hen class` should have dependency on some kind of abstraction.
- So `AngryBirds class` instead of depending on the `Hen class`, it should be depending on the abstract `Bird class`. The `Hen class` should also depend on the `Bird class`.



```
class AngryBirds {
public:
	void renderBirds(const std::unique_ptr<Bird>& bird)
	{
		cout << "Our bird name is " << bird->getName() << " which is of "
			<< bird->getType() << " type and is " << bird->getColor()
			<< " in color.\n";

		auto flyableBird = dynamic_cast<IFlyable*>(bird.get());
		if (flyableBird)
		{
			flyableBird->fly();
		}
	}
};
```

```
int main() {
    vector<unique_ptr<Bird>> birds;

    birds.push_back(make_unique<Hen>("Ginger", "Domestic", "Brown", 2.5f));
    birds.push_back(make_unique<Parrot>("Snowball", "Tropical", "White", 2.8f));

    AngryBirds angryBirds;

    for (const auto& bird : birds) {
        angryBirds.renderBirds(bird);
    }

      return 0;
}

/************************************
Our bird name is Ginger which is of Domestic type and is Brown in color.
Ginger is Flying low.
Our bird name is Snowball which is of Tropical type and is White in color.
Snowball is flying with colorful wings.
************************************/
```

- Instead of creating new object in the `renderBirds()` method, the `renderBirds()` method now accepts a object of abstract `Bird class`.
- Now we can pass any `Bird`s to `renederBird()` method.
- We can pass object to constructor or method. We are injecting the dependency to the `renederBird()` method in the above case. If we implement the same via constructor, we would have injected the dependency via constructor. This is called as Dependency injection.
- We are achieving dependency inversion by injecting dependency. The dependency inversion principle says,
    - No 2 concrete classes should depend on each other directly, they should depend on abstraction.
- Key Concepts of Dependency Inversion Principle (DIP):
    - High-level modules should not depend on low-level modules. Both should depend on abstractions.
        - Before applying DIP,
            - `AngryBirds` is a high-level module because it handles the rendering of birds.
            - `Hen` is a low-level module with specific details about a type of bird.
            - `AngryBirds` directly depends on the `Hen class`, a specific implementation.
        - After applying DIP
            - High-Level Module (`AngryBirds`):
                - No longer depends on the concrete `Hen class`. Instead, it operates on the `Bird` abstraction.
                - This makes `AngryBirds` more flexible and easier to maintain.

- Low-Level Module (`Hen`, `Parrot`):
    - Implements the `Bird` interface but does not need to be aware of how `AngryBirds` will use it.
- Abstractions:
    - Both the high-level module (`AngryBirds`) and the low-level modules (`Hen`, `Parrot`) depend on the abstraction `Bird`.

- Abstractions should not depend on details. Details should depend on abstractions.
    - The `Bird` `class` is the abstraction that defines the common interface.
    - Specific bird types like `Hen` and `Parrot` are details that depend on this abstraction.
    - The `AngryBirds` `class` depends on the `Bird` abstraction rather than on specific details, allowing it to work with any bird type that adheres to the `Bird` interface.