Design Patterns

Contents

[1. Introduction 2](#_Toc177052500)

[Types of Object-Oriented Desing Patterns 2](#_Toc177052501)

[Creational Design Pattern 2](#_Toc177052502)

[Structural Desing Pattern 2](#_Toc177052503)

[Behavioral Design Pattern 2](#_Toc177052504)

[2. Creational Design Patterns 3](#_Toc177052505)

[Singleton Desing Pattern 3](#_Toc177052506)

[Configuration Manager – Singleton 3](#_Toc177052507)

[Builder Design Pattern 7](#_Toc177052508)

[Towards Builder - Step 1 8](#_Toc177052509)

[Towards Builder - Step 2 8](#_Toc177052510)

[Towards Builder - Step 3 10](#_Toc177052511)

[Towards Builder - Step 4 12](#_Toc177052512)

[Towards Builder - Step 5 13](#_Toc177052513)

[Prototype Design Pattern 15](#_Toc177052514)

[Towards Prototype - Step 1 15](#_Toc177052515)

[Towards Prototype - Step 2 15](#_Toc177052516)

[Towards Prototype - Step 3 16](#_Toc177052517)

[Towards Prototype - Step 4 17](#_Toc177052518)

[Registry Design Pattern 18](#_Toc177052519)

[Practical/Static/Simple Factory 20](#_Toc177052520)

[Factory Method Desing Pattern 23](#_Toc177052521)

[Abstract Factory Desing Pattern 27](#_Toc177052522)

[Visual Studio 2022 Solution File 31](#_Toc177052523)

# Introduction

* Design patterns are general, reusable solutions to common problems that occur in software design.
* They are not finished designs that can be directly converted into code but are templates that guide you in solving particular design issues.
* Design patterns capture best practices and proven strategies, making code more maintainable, flexible, and easier to understand.
* GOF – Gang of Four – 23 Design Patterns
* Why we need to learn?
  + Standard solution to common problems.
  + Common vocabulary across software industry.

## Types of Object-Oriented Desing Patterns

### Creational Design Pattern

* Deals with “How an object can be created”?
* Different ways of creating an object.
* How many objects can be created?

### Structural Desing Pattern

* How a class will be structured?
* How should different classes relate to each other?
* What all the attributes and behavior will be there in a class (Interface or Abstract class)

### Behavioral Design Pattern

* Deals with the methods.

# Creational Design Patterns

* Here we will study
  + Singleton
  + Builder
  + Factory
    - Factory Method
    - Abstract Factor
    - Practical Factor
  + Prototype

## Singleton Desing Pattern

* Singleton allows us to create a class for which a single object can be created.
* Some of the use cases of Singleton Desing Patterns are…
  + Configuration or Settings Manager:
    - When an application needs a single, centralized access point for configuration settings that are used across the system.
  + Logging Service:
    - Ensures that there is a single logging instance for the entire application to avoid multiple log files and to manage logging in a consistent way.
  + Database Connection Pool:
    - A singleton can manage a pool of database connections, ensuring controlled access and minimizing resource usage by reusing connections.
  + Thread Pool Manager:
    - Managing thread pools where you need a centralized object that manages the creation, reuse, and destruction of threads.
  + Resource Manager (e.g., Printer Spooler):
    - Managing access to a shared resource like a printer or file handler, ensuring that only one instance is coordinating resource usage.

### Configuration Manager – Singleton

* A Configuration Manager (often referred to as a Configuration or Settings Manager) is a component in software systems that centralizes the management and access to configuration settings used throughout an application. It plays a crucial role in ensuring that various parts of an application have consistent and easy access to configuration data, which can include settings like:
  + Application settings (e.g., UI preferences, language, theme).
  + Database connection strings.
  + API keys and secrets.
  + Environment-specific settings (e.g., development, testing, production configurations).
  + Feature toggles (to enable/disable specific features dynamically).
* Let’s build singleton ConfigManager class from basic.

|  |
| --- |
| class ConfigManager  {  // Storage for settings  std::unordered\_map<std::string, std::string> settings\_;  public:  void setSetting(const std::string& key, const std::string& value);  std::string getSetting(const std::string& key) const;  void loadSettings();  }; |

* Is the ConfigManager class a Singleton?
  + No, because we can any number of objects of the ConfigManager class.
* Can we say, till the time the class has a public constructor, that class can never be a Singleton.
  + So, we can make the constructor private.

|  |
| --- |
| class ConfigManager  {  // Storage for settings  std::unordered\_map<std::string, std::string> settings\_;  ConfigManager(); // Private constructor  public:  void setSetting(const std::string& key, const std::string& value);  std::string getSetting(const std::string& key) const;  void loadSettings();  }; |

* If we make the constructor private, we **cannot create any objects**. But we can access the private things of a class inside the methods of the class. So, we can define a method inside the ConfigManager class which creates a static instance of itself.

|  |
| --- |
| class ConfigManager  {  // Storage for settings  std::unordered\_map<std::string, std::string> settings\_;  ConfigManager(); // Private constructor  public:  void setSetting(const std::string& key, const std::string& value);  std::string getSetting(const std::string& key) const;  void loadSettings();    ConfigManager& getInstance()  {  static ConfigManager instance;  return instance;  }  }; |

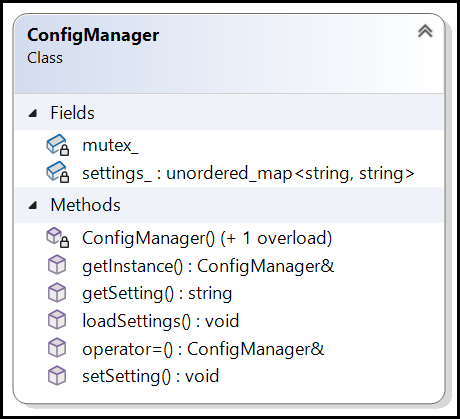
* But to call getInstance() method, we need to create an object of the class ConfigManager. Since we cannot create an object because the constructor is private, we can make the method getInstance() static.

|  |
| --- |
| class ConfigManager  {  // Storage for settings  std::unordered\_map<std::string, std::string> settings\_;  ConfigManager(); // Private constructor  public:  void setSetting(const std::string& key, const std::string& value);  std::string getSetting(const std::string& key) const;  void loadSettings();    static ConfigManager& getInstance()  {  static ConfigManager instance;  return instance;  }  }; |

* The static keyword ensures that the ConfigManager instance is initialized only once, even if the containing function getInstance() is called multiple times.
* The initialization happens the first time the control flow reaches the line containing the definition of static ConfigManager instance during the program's execution.
* Since C++11, the initialization of local static variables is guaranteed to be thread-safe, meaning that the initialization will be correctly synchronized if multiple threads try to access the instance simultaneously.
* static ConfigManager instance is initialized during program load because static local variables within functions are specifically designed to be initialized when the function is called, not before the program's execution begins.
* The static local variables are initialized at their first use during runtime.
* The static member variables and static global/namespace variables are initialized at program load time, before main() starts.
* We also need to delete copy constructor and assignment operator. If we don’t delete them, we can do the following…

|  |
| --- |
| ConfigManager config2 = config; // This creates a new instance  config2.show();  // Assigning one instance to another  ConfigManager config3 = ConfigManager::getInstance();  config3 = config; // This copies s1 to s3  config3.show(); |

* The line ConfigManager config2 = config; uses the copy constructor to create a new instance config2. This instance is a separate object with the same state but a different memory address.
* The line config3 = config; uses the assignment operator to copy config into config3. Although config3 already exists, it now contains the state of config1, making it another unwanted copy.
* Final ConfigManager Singleton is given below…



|  |
| --- |
| class ConfigManager  {  // Storage for settings  std::unordered\_map<std::string, std::string> settings\_;  ConfigManager(); // Private constructor  mutable std::mutex mutex\_;  public:  // Delete copy constructor.  ConfigManager(const ConfigManager&) = delete;  // Delete assignment operator.  ConfigManager& operator=(const ConfigManager&) = delete;  static ConfigManager& getInstance(); // Static method.    void setSetting(const std::string& key, const std::string& value);  std::string getSetting(const std::string& key) const;  void loadSettings();  };  ConfigManager& ConfigManager::getInstance()  {  static ConfigManager instance;  return instance;  }  ConfigManager::ConfigManager()  {  loadSettings();  }  void ConfigManager::setSetting(const std::string& key, const std::string& value)  {  std::lock\_guard<std::mutex> lock(mutex\_);  settings\_[key] = value;  std::cout << "Setting updated: " << key << " = " << value << std::endl;  } |

## Builder Design Pattern

* Why do we need Builder Design Pattern?
  + Requirement 1: Imagine we have to design a class Employee with lot of attributes.
  + If we want to create an object of Employee class and set these attributes, we can do…

|  |
| --- |
| Employee st;  st.SetFirstName("Nishith");  st.SetLastName("Jain");  st.SetAge(42);  st.SetEmployeeId("ISICE035");  // We have to use all the setters. |

* + Requirement 2: We want to validate the Employee object before creation.
    - We only want to create an Employee object if the first and last names are given.
    - We only want to create an Employee object if the age of the employee is > 21.
    - We only want to create an Employee object if a phone number is given.
    - We only want to create an Employee object if the nationality is Indian.
    - No object of the Employee class should be created until all the validation passes.
* Validation can be done inside the constructor. If validations are coming inside the constructor, then setters are needed.
* Constructor should take all the attribute values as parameter in order to validate the attributes as shown…

|  |
| --- |
| Employee(std::string employee\_id, std::string first\_name, std::string last\_name,  std::string date\_of\_birth, std::string gender, int age, std::string nationality,  std::string address, std::string phone\_number, std::string email)  : employee\_id\_(std::move(employee\_id)), first\_name\_(std::move(first\_name)),  last\_name\_(std::move(last\_name)), date\_of\_birth\_(std::move(date\_of\_birth)),  gender\_(std::move(gender)), age\_(age),  nationality\_(std::move(nationality)), address\_(std::move(address)),  phone\_number\_(std::move(phone\_number)), email\_(std::move(email))  {  } |

* This is not at all readable, difficult to understand, prone to error.
* During the creation, we might not have all the parameter values. There can be lot of overloaded constructors. Sometimes it is not possible to have few constructors.

|  |
| --- |
| Employee(std::string first\_name, std::string last\_name, std::string date\_of\_birth);  Employee(std::string first\_name, std::string last\_name, std::string employee\_id);  // Error: Constructor already defined |

* As per coding guidelines and best practices, a method or function should take fewer parameters to maintain readability, maintainability, and simplicity. The ideal range is 3 to 5 parameters, with a maximum of 7 parameters.

### Towards Builder - Step 1

* So we can think of passing a std::map<std::string, std::string> as an argument.
* Reasons Not to Use std::map for Passing Multiple Parameters
  + Loss of Type Safety: When using a std::map<std::string, std::string> (or similar), all values are stored as strings, which leads to a loss of type safety.
  + Error-Prone: Key names in a map are usually hard-coded strings, leading to potential runtime errors if there are typos in the key names or if a key is missing.
  + Difficulty in Validation: Validating the input parameters (like checking if the first\_name is given or the age is above 21) becomes more complex and less efficient when values are stored in a map. Each validation check involves extracting values from the map and handling potential errors, increasing code complexity.
  + Lack of Self-Documentation: Parameter names provide useful information about the data being passed to a function. Using a std::map obfuscates this intent, as it relies on arbitrary string keys that don't provide any compile-time checks.

### Towards Builder - Step 2

* Instead of a std::map, we can create an Helper class, which has the same attributes as that of Employee class.

|  |
| --- |
| class Helper  {  public:  std::string employee\_id\_;  std::string first\_name\_;  std::string last\_name\_;  std::string date\_of\_birth\_;  std::string gender\_;  int age\_ = 0;  std::string nationality\_;  std::string address\_;  std::string phone\_number\_;  std::string email\_;  }; |

* We can set the values for all the parameters for this Helper class object.

|  |
| --- |
| int main() {  Helper h;  h.first\_name\_ = "Nishith";  h.last\_name\_ = "Jain";  h.age\_ = 42;  ...  ...  return 0;  } |

* After setting the Helper class object, we can pass this object as parameter to the constructor of the Employee class.

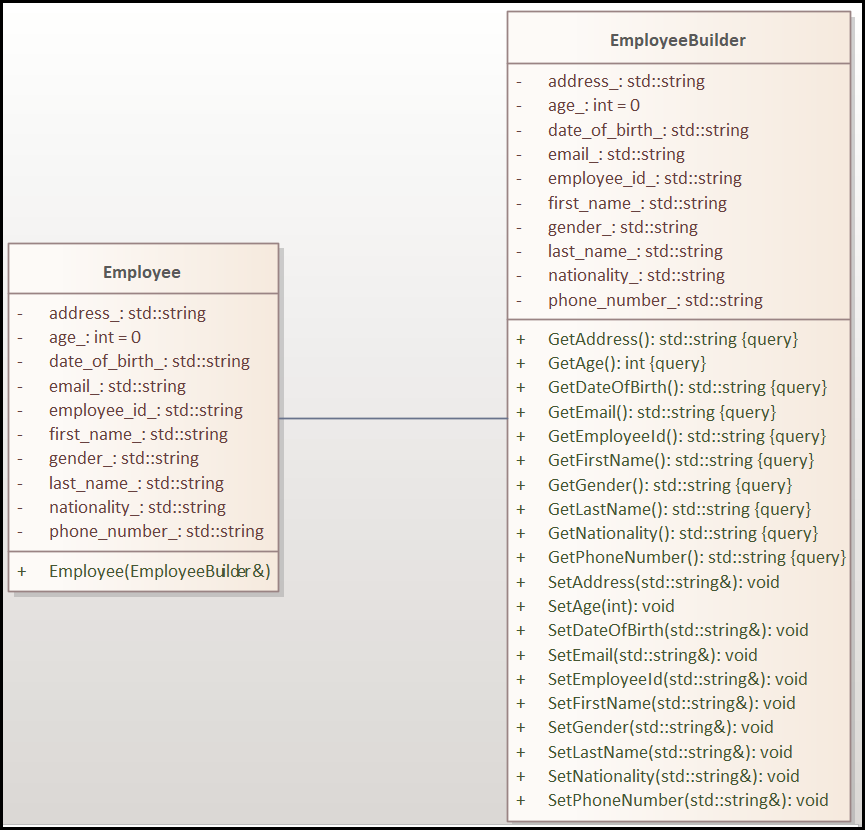
|  |
| --- |
| class Employee  {  // General Information  std::string employee\_id\_;  std::string first\_name\_;  std::string last\_name\_;  std::string date\_of\_birth\_;  std::string gender\_;  int age\_ = 0;  std::string nationality\_;  std::string address\_;  std::string phone\_number\_;  std::string email\_;  public:  explicit Employee(const Helper& h);  }; |
| int main() {  Helper h;  h.first\_name\_ = "Nishith";  h.last\_name\_ = "Jain";  h.age\_ = 42;  ...  ...  Employee anEmployeeObj(h);  return 0;  } |

* In the constructor Employee class, we can do the validation of Helper class object, if the validation fails, we throw an exception. If validation passes, then we can create the Employee object.

|  |
| --- |
| class Employee {  // General Information  std::string employee\_id\_;  std::string first\_name\_;  std::string last\_name\_;  std::string date\_of\_birth\_;  std::string gender\_;  int age\_ = 0;  std::string nationality\_;  std::string address\_;  std::string phone\_number\_;  std::string email\_;  public:  explicit Employee(const Helper& h) {  if (h.first\_name\_.empty() || h.last\_name\_.empty()) {  throw std::invalid\_argument("First and last names are required.");  }  if (age\_ <= 21) {  throw std::invalid\_argument("Age must be greater than 21.");  }  if (phone\_number\_.empty()) {  throw std::invalid\_argument("Phone number is required.");  }  if (nationality\_ != "Indian") {  throw std::invalid\_argument("Only Indian nationality is allowed.");  }  // After validation, we can create Employee object...  this->first\_name\_ = h.first\_name\_;  this->last\_name\_ = h.last\_name\_;  ...  ...  }  }; |

### Towards Builder - Step 3

* This Helper class, is called as Builder.
* Let’s create a EmployeeBulider class from scratch. This class contains all the attributes of the Employee class and all the attributes are private.
* To set and get the attributes, we will have getters and setters.
* Now in the Employee class, we create a constructor that take EmployeeBulider object as parameter.



|  |
| --- |
| class Employee  {  std::string employee\_id\_;  std::string first\_name\_;  std::string last\_name\_;  std::string date\_of\_birth\_;  std::string gender\_;  int age\_ = 0;  std::string nationality\_;  std::string address\_;  std::string phone\_number\_;  std::string email\_;  public:  Employee(const  EmployeeBuilder& emp\_builder);  }; |

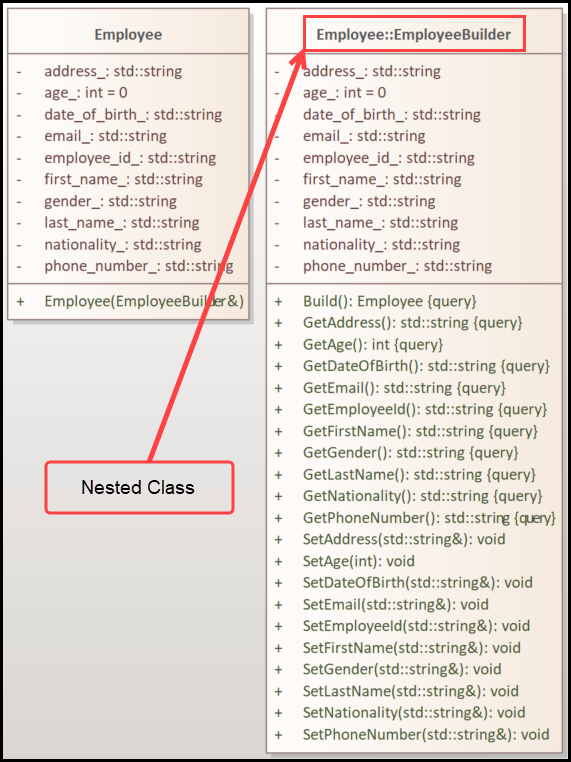
* We can create an object of EmployeeBulider and set all the values using setters.
* Then we can pass this EmployeeBulider object as the argument to the constructor while creating Employee object.

|  |
| --- |
| int main() {  EmployeeBuilder employee\_builder;  employee\_builder.SetFirstName("Nishith");  employee\_builder.SetLastName("Jain");  employee\_builder.SetAge(42);  employee\_builder.SetPhoneNumber("1234567890");  employee\_builder.SetNationality("Indian");  Employee anEmployeeObj(employee\_builder);  return 0;  } |

* We do all the validations in the Employee constructor. If the validation fails, we throw an exception or else we create Employee object!

|  |
| --- |
| Employee::Employee(const EmployeeBuilder& emp\_builder)  {  if (emp\_builder.GetFirstName().empty() || emp\_builder.GetLastName().empty())  {  throw std::invalid\_argument("First and last names are required.");  }  if (emp\_builder.GetAge() <= 21) {  throw std::invalid\_argument("Age must be greater than 21.");  }  if (emp\_builder.GetPhoneNumber().empty()) {  throw std::invalid\_argument("Phone number is required.");  }  if (emp\_builder.GetNationality() != "Indian") {  throw std::invalid\_argument("Only Indian nationality is allowed.");  }  // After validation, we can create Employee object...  this->first\_name\_ = emp\_builder.GetFirstName();  this->last\_name\_ = emp\_builder.GetLastName();  } |

### Towards Builder - Step 4

* What other enhancements we can make to this code so that the code is more readable, maintainable and production ready?
* We can make the EmployeeBuilder class is nested within the Employee class to encapsulate the building logic tightly within the Employee class itself. This encapsulation provides the following benefits:
  +  **Scoping**: The EmployeeBuilder class is directly tied to the Employee class, which makes it clear that its sole purpose is to construct Employee objects.
  + **Access**: The nested class can access private members of the Employee class, which can simplify the building process.
  + **Readability and Organization**: Nesting the EmployeeBuilder class keeps related code together, which can improve readability and make the code easier to maintain.
* First, we create the EmployeeBuilder object and then we set the values using the setters.
* Once all the values are set, then we can pass this EmployeeBuilder object to Employee constructor to create Employee object.

|  |
| --- |
| auto builder = Employee::EmployeeBuilder();  builder.SetFirstName("Nishith");  builder.SetLastName("Jain");  ...  ...  Employee emp(builder); |

### Towards Builder - Step 5

* First, we accumulate the necessary data in a flexible, readable manner by **method chaining**. This can be achieved by returning \*this. For example…

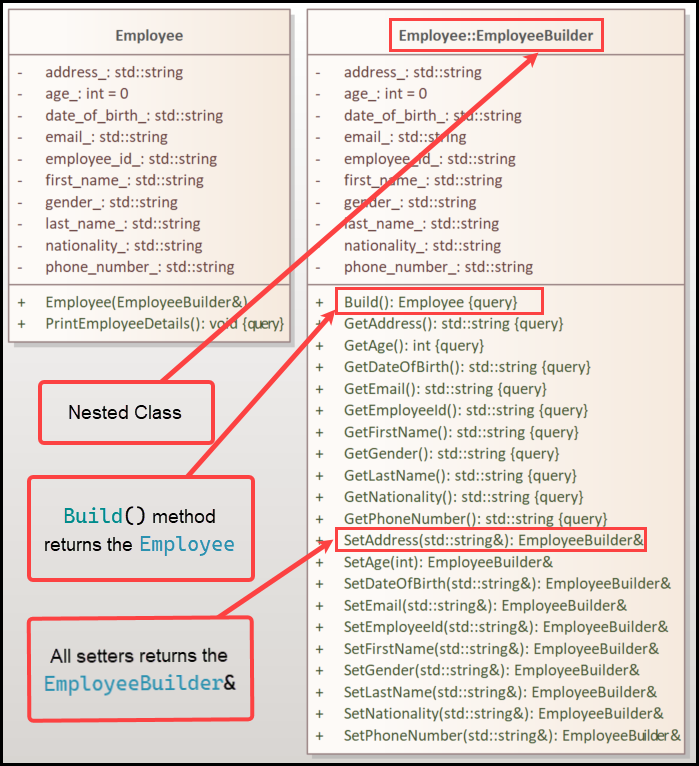
|  |
| --- |
| **Before:**  void Employee::EmployeeBuilder::SetEmployeeId(const std::string& employee\_id)  {  employee\_id\_ = employee\_id;  } |
| **After:**  Employee::EmployeeBuilder& Employee::EmployeeBuilder::SetEmployeeId(const std::string& employee\_id)  {  employee\_id\_ = employee\_id;  return \*this;  } |

* **Setter Methods Return** EmployeeBuilder&:
  + Each **setter** method in the EmployeeBuilder class returns a reference to the EmployeeBuilder object itself (EmployeeBuilder&).
* Chaining in Action:
  + When we call a setter, like SetAddress, it modifies the internal state of the EmployeeBuilder and then returns the current EmployeeBuilder instance, allowing another setter to be called immediately. For example…

|  |
| --- |
| const auto builder = Employee::EmployeeBuilder().  SetFirstName("Nishith").  SetLastName("Jain").  SetAge(42).  SetDateOfBirth("09-May-1982"); |

* Now we have the EmployeeBuilder object with all the data which will be used in the construction of the Employee object.
* We can expose a Build() method in the EmployeeBuilder that directly creates and returns an Employee object. This approach eliminates the need to create an EmployeeBuilder object separately and avoids passing it to the Employee constructor manually.

|  |
| --- |
| Employee Employee::EmployeeBuilder::Build() const  {  return Employee(\*this);  } |

* From the client code perspective, the Build() method provides a clean, understandable way to finalize the object creation without directly calling a constructor.

|  |
| --- |
| int main() {  Employee employee = Employee::EmployeeBuilder()  .SetEmployeeId("E12345")  .SetFirstName("Nishith")  .SetLastName("Jain")  .SetDateOfBirth("09-May-1982")  .SetGender("Male")  .SetAge(42)  .SetNationality("Indian")  .SetAddress("G236, Brigade Northridge")  .SetPhoneNumber("+919876543210")  .SetEmail("nishith.jain@example.com")  .Build();  employee.PrintEmployeeDetails();  return 0;  } |

Click below to open the file.

## Prototype Design Pattern

**Problem statement**: Given an object of a class, we need to create a copy of that object with same attributes.

Example: We have an object of Player class, we need to create a copy of that object.

### Towards Prototype - Step 1

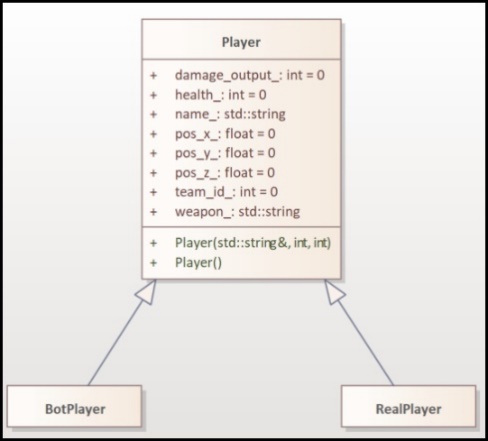
* We can do a member wise copy.

|  |
| --- |
| Player player("DefaultPlayer", 101, 1000);  Player playerCopy;  playerCopy.name\_ = "SoleMortal";  playerCopy.health\_ = player.health\_;  playerCopy.damage\_output\_ = player.damage\_output\_; |

* What is the disadvantage of doing this way…
  + Some fields in the Player class can be private, hence cannot be copied.
  + Client needs to know the implementation details of the Player class to copy so that client won’t miss any attributes while copying.

### Towards Prototype - Step 2

* There can be 2 types of players, BotPlayer and RealPlayer.



* If we are creating a copy, from pointer to base class Player and object from which we are copying can be a Player or BotPlayer or RealPlayer, then the copy should also be of the respective type.
* Not a good approach.

|  |
| --- |
| const auto player = make\_unique<Player>();  std::unique\_ptr<Player> playerCopy;  if (dynamic\_cast<BotPlayer\*>(player.get())) {  playerCopy = std::make\_unique<BotPlayer>(\*dynamic\_cast<BotPlayer\*>(player.get()));  }  else if (dynamic\_cast<RealPlayer\*>(player.get())) {  playerCopy = std::make\_unique<RealPlayer>(\*dynamic\_cast<RealPlayer\*>(player.get()));  }  else if (player) {  playerCopy = std::make\_unique<Player>();  }  else {  std::cout << "Unknown Player Type!\n";  } |

### Towards Prototype - Step 3

* Let’s try to use copy constructor.

|  |
| --- |
| class BotPlayer : public Player  {  int difficulty\_level\_ = 0;  public:  // Copy constructor  BotPlayer(const BotPlayer& other)  : Player(other), difficulty\_level\_(other.difficulty\_level\_)  {  }  };  class RealPlayer : public Player  {  int experience\_points\_ = 0;  public:  // Copy constructor  RealPlayer(const RealPlayer& other)  : Player(other), experience\_points\_(other.experience\_points\_)  {  }  }; |

* Copy constructors are not polymorphic and cannot be used on base class pointers to create copies of derived class objects. Attempting to use a base class copy constructor would result in slicing, where derived class-specific data is lost.
* So, we have to use if-else to check the type and create a copy…

|  |
| --- |
| std::unique\_ptr<Player> player = std::make\_unique<BotPlayer>("Bot1", 1, 2);  std::unique\_ptr<Player> playerCopy;  // Determine the type of player and copy accordingly using copy constructors  if (const auto botPlayer = dynamic\_cast<BotPlayer\*>(player.get())) {  // If player is of type BotPlayer, copy as BotPlayer  playerCopy = std::make\_unique<BotPlayer>(\*botPlayer);  }  else if (const auto realPlayer = dynamic\_cast<RealPlayer\*>(player.get())) {  // If player is of type RealPlayer, copy as RealPlayer  playerCopy = std::make\_unique<RealPlayer>(\*realPlayer);  }  else if (auto basePlayer = dynamic\_cast<Player\*>(player.get())) {  // If player is of type Player, copy as Player  playerCopy = std::make\_unique<Player>(\*basePlayer);  }  else {  std::cout << "Unknown Player Type!" << std::endl;  } |

* This approach is also not good.

### Towards Prototype - Step 4

* Let’s outsource the work of creating copy/clone so that client need not worry about all these issues.
* For example, if client wants to create a copy/clone of Player, it will simply call a method on the object.
* Irrespective of the type of Player (Player, BotPlayer, RealPlayer)copy/clone of the object is created.

|  |
| --- |
| // Create different types of players  std::unique\_ptr<Player> player1 = std::make\_unique<BotPlayer>("Bot1", 2);  std::unique\_ptr<Player> player2 = std::make\_unique<RealPlayer>("Player1", 1500);  // Copying the players  std::unique\_ptr<Player> playerCopy1 = player1->clone(); // Copy of BotPlayer  std::unique\_ptr<Player> playerCopy2 = player2->clone(); // Copy of RealPlayer |

* To achieve this, we need to define virtual clone() method in base Player class.

|  |
| --- |
| class Player  {  protected:  std::string name\_;  int health\_ = 0;  int team\_id\_ = 0;  float pos\_x\_ = 0;  float pos\_y\_ = 0;  float pos\_z\_ = 0;  std::string weapon\_;  int damage\_output\_ = 0;  public:  Player() = default;  virtual ~Player() = default;  virtual std::unique\_ptr<Player> clone() const = 0;  }; |

* Each derived class (BotPlayer, RealPlayer) implements the clone() method, which returns a std::make\_unique of the correct type by using std::make\_unique<T>(\*this), ensuring a deep copy is made.

|  |
| --- |
| std::unique\_ptr<Player> BotPlayer::clone() const  {  return std::make\_unique<BotPlayer>(\*this);  } |

* The above code calls the copy constructor…
  + The expression \*this is passed as an argument to std::make\_unique<BotPlayer>.
  + Internally, std::make\_unique<BotPlayer> forwards the arguments (\*this in this case) to the constructor of BotPlayer.
  + Since \*this is an instance of BotPlayer, the compiler looks for the BotPlayer constructor that matches the parameter type.
  + In this case, BotPlayer(const BotPlayer& other) (the copy constructor) is a match, so it gets called.
* If we are taking this approach, we have to **make sure all the child** classes override the clone() **method**.

## Registry Design Pattern

**Problem statement**: Given an application that manages **multiple intern batches** (one inter batch can have 100s of intern objects) each with distinct batch-specific(there are many such batches) attributes, we need a mechanism to create intern objects dynamically based on different batch.

* Prototype Design Pattern:
  + Creating Intern Objects: The Prototype Pattern is used to create new intern objects by cloning existing prototypes. Each batch of interns can be represented as a prototype, allowing you to efficiently create new instances of interns based on these prototypes.
* Registry Design Pattern:
  + Based on Batches: The Registry Pattern is employed to manage these prototypes. It provides a centralized mechanism (the registry) for storing and retrieving prototypes based on batch identifiers or keys. This allows you to dynamically access and clone prototypes for different intern batches without having to hardcode the creation logic.
* As per the problem statement, there are many batches. Each batch has a prototype.
* Example, for the batch “January 2024”, we can create a prototype…

|  |
| --- |
| auto prototype\_jan\_2024 = std::make\_unique<Intern>("January 2024", "2024-01-01",  "2024-06-30", "6 months", "Location A"); |

* For the batch “February 2024”, we create a prototype…

|  |
| --- |
| auto prototype\_feb\_2024 = std::make\_unique<Intern>("February 2024", "2024-02-01",  "2024-07-31", "6 months", "Location B"); |

* Using these prototypes we create hundreds of objects.

|  |
| --- |
| // This will create an inter object for the batch Jan 2024  auto intern\_jan\_2024 = prototype\_jan\_2024->clone();  // This will create an inter object for the batch Feb 2024  auto intern\_feb\_2024 = prototype\_feb\_2024->clone(); |

* To create hundreds of objects during runtime based on the batch, we need to store prototypes in a map which acts as a registry and retrieve it based on batch. Batch is the key.

|  |
| --- |
| std::unordered\_map<std::string, std::unique\_ptr<Intern>> registry\_; |

* We can have a class that serves as a centralized registry to store and manage different Intern prototypes. We can call it InternRegistry.
* To store the prototype, we can expose a method RegisterIntern() which takes batch and prototype as parameters.

|  |
| --- |
| void RegisterIntern(const std::string& key, std::unique\_ptr<Intern> prototype); |

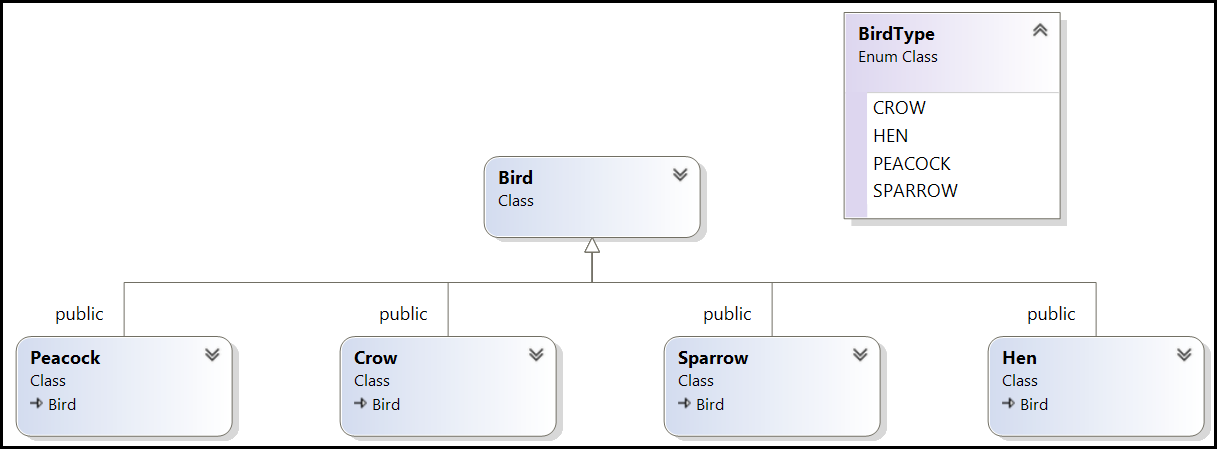
* To store the prototype, we can expose a method RegisterIntern() which takes batch and prototype as parameters.
* We can expose a methos CreateIntern() which creates new Intern object by cloning a registered prototype based on the provided key.

|  |
| --- |
| std::unique\_ptr<Intern> CreateIntern(const std::string& key) const; |

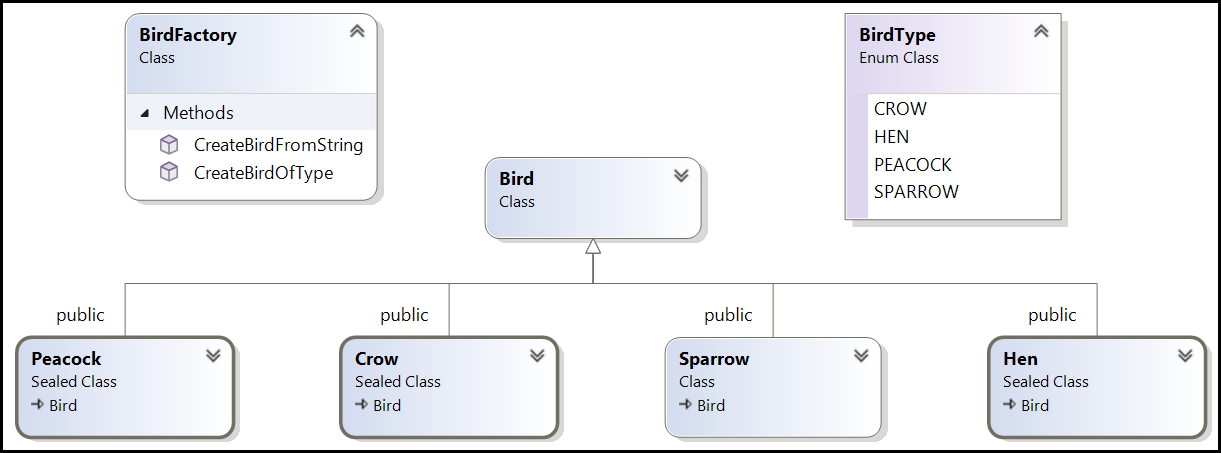
* This is the Registry Design Pattern. In this example, the Registry Design Pattern is used to manage the creation of Intern objects based on different batches, such as "Jan2024" and "Feb2024".

|  |
| --- |
| int main()  {  InternRegistry registry;  // Registering January 2024 batch prototype  registry.RegisterIntern("Jan2024", std::make\_unique<Intern>(  "January 2024", "2024-01-01", "2024-06-30", "6 months", "Location A"));  // Registering February 2024 batch prototype  registry.RegisterIntern("Feb2024", std::make\_unique<Intern>(  "February 2024", "2024-02-01", "2024-07-31", "6 months", "Location B"));  if (const auto intern1 = registry.CreateIntern("Jan2024")) {  intern1->SetName("Alice");  intern1->SetAge(22);  intern1->SetUniversity("XYZ University");  intern1->SetGender("Female");  intern1->SetPhoneNumber("1234567890");  intern1->SetEmail("alice@example.com");  intern1->Display();  }  if (const auto intern2 = registry.CreateIntern("Feb2024")) {  intern2->SetName("Bob");  intern2->SetAge(23);  intern2->SetUniversity("ABC University");  intern2->SetGender("Male");  intern2->SetPhoneNumber("0987654321");  intern2->SetEmail("bob@example.com");  intern2->Display();  }    return 0;  } |

## Practical/Static/Simple Factory

* Not considered a formal pattern by some, but commonly used.
* This pattern moves the responsibility of creating object of corresponding class from the client code to the factory class.
* Imagine, we have a Bird class and Crow, Hen, Peacock and Sparrow inherit from this Bird class as shown below.
* In the client, we have to create concrete birds based on the BirdType. So, we can write…

|  |
| --- |
| int main()  {  unique\_ptr<Bird> bird;  if(const auto birdType = GetIndianBirdType(); birdType == BirdType::CROW)  {  bird = make\_unique<Crow>();  }  else if(birdType == BirdType::HEN)  {  bird = make\_unique<Hen>();  }  else if(birdType == BirdType::PEACOCK)  {  bird = make\_unique<Peacock>();  }  else if (birdType == BirdType::SPARROW)  {  bird = make\_unique<Sparrow>();  }  bird->MakeSound();  return 0;  } |

* The above code even though works, it is bad code…
  + **Violation of the Open/Closed Principle**: This code is not easily extendable. If a new BirdType is introduced, you must modify the main() function to add another else if condition.
  + **High Coupling**: The main() function is tightly coupled with the specific bird classes (Crow, Hen, Peacock, Sparrow). If the class names, constructors, or other details change, you'll need to update this section of the code.
  + **Hard to Unit Test:** Testing the object creation logic requires setting up the entire main() function environment.
* To address these issues, we can encapsulate the object creation logic into a separate function or class, making our code cleaner, more maintainable, and easier to extend. Let’s call this class as BirdFactory which implements a static function.

|  |
| --- |
| class BirdFactory  {  public:  static std::unique\_ptr<Bird> CreateBirdOfType(BirdType bird\_type);  static std::unique\_ptr<Bird> CreateBirdFromString(const std::string& bird\_type);  };  std::unique\_ptr<Bird> BirdFactory::CreateBirdOfType(const BirdType bird\_type)  {  switch (bird\_type)  {  case BirdType::CROW:  return std::make\_unique<Crow>();  case BirdType::HEN:  return std::make\_unique<Hen>();  case BirdType::PEACOCK:  return std::make\_unique<Peacock>();  case BirdType::SPARROW:  return std::make\_unique<Sparrow>();  default:  return nullptr;  }  }  std::unique\_ptr<Bird> BirdFactory::CreateBirdFromString(const std::string& bird\_type)  {  const std::string lowerBirdType = ToLower(bird\_type);  if (lowerBirdType == "crow") {  return std::make\_unique<Crow>();  }  if (lowerBirdType == "hen") {  return std::make\_unique<Hen>();  }  if (lowerBirdType == "peacock") {  return std::make\_unique<Peacock>();  }  if (lowerBirdType == "sparrow") {  return std::make\_unique<Sparrow>();  }  std::cerr << "Unknown bird type: " << bird\_type << '\n';  return nullptr;  }  BirdType GetIndianBirdType()  {  // Some logic for getting Indian Bird from BirdType  return BirdType::PEACOCK;  }  int main()  {  const unique\_ptr<Bird> bird = BirdFactory::CreateBirdOfType(GetIndianBirdType());  bird->MakeSound();  return 0;  } |

* Advantages of Using the Factory Function
  + **Simplifies the main function**: The main logic is now simple and focused only on invoking the factory and using the created object.
  + **Centralized Object Creation**: All instantiation logic is centralized in the factory function, making it easier to manage.
  + **Scalability**: To add a new bird type, you only need to add a new case in the factory function without modifying any client code.

## Factory Method Desing Pattern

* The Factory Method pattern defines an interface (Behavior) for creating objects but allows subclasses to alter the type of objects that will be created.

**Requirement**: Imagine a scenario where we are developing a game that needs to handle different types of vehicles, such as Car, Buggy, Monster Truck, Dacia, Motorcycle etc...

* First let’s define common interface or base class for all Vehicle types.

|  |
| --- |
| class Vehicle {  public:  virtual ~Vehicle() = default;  virtual void Drive() const = 0;  }; |

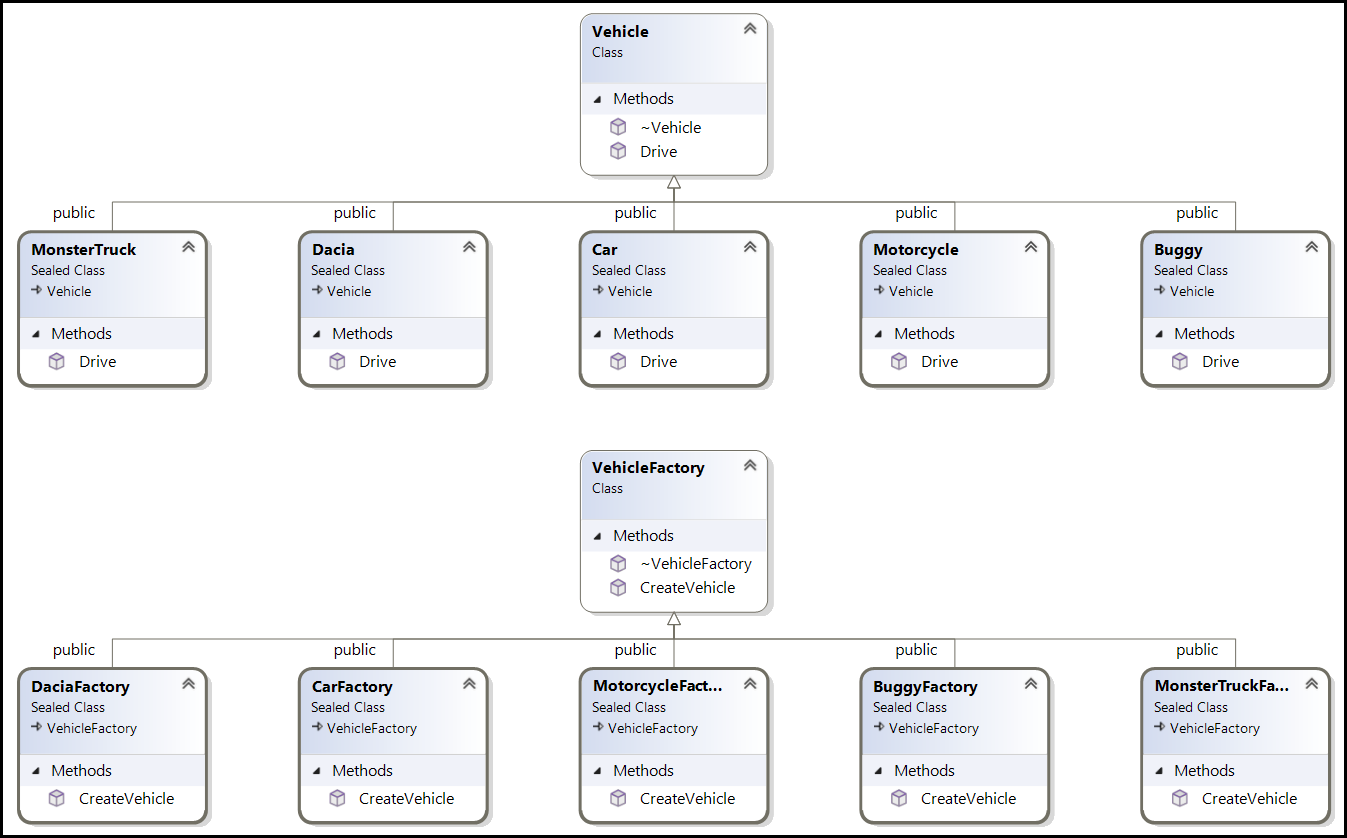
* Each concrete vehicle (Car, Buggy, MonsterTruck, Dacia, Motorcycle) class implements the common interface.

|  |
| --- |
| class Car final : public Vehicle {  public:  void Drive() const override;  };  class Buggy final : public Vehicle {  public:  void Drive() const override;  };  class MonsterTruck final : public Vehicle {  public:  void Drive() const override;  };  class Dacia final : public Vehicle {  public:  void Drive() const override;  };  class Motorcycle final : public Vehicle {  public:  void Drive() const override;  }; |

* We declare the factory method CreateVehicle() in the abstract VehicleFactory class.

|  |
| --- |
| class VehicleFactory {  public:  virtual ~VehicleFactory() = default;  // The Factory Method that subclasses will override  // to create specific vehicles.  virtual std::unique\_ptr<Vehicle> CreateVehicle() const = 0;  }; |

* The subclasses of VehicleFactory (such as CarFactory, BuggyFactory, DaciaFactory, MonsterTruckFactory, and MotorcycleFactory) override this CreateVehicle() method. These subclasses are responsible for the creation of specific objects (Car, Buggy, MonsterTruck, Dacia, Motorcycle) by implementing the CreateVehicle() method according to their respective types.



|  |
| --- |
| class CarFactory : public VehicleFactory {  public:  std::unique\_ptr<Vehicle> CreateVehicle() const override {  return std::make\_unique<Car>();  }  };  class BuggyFactory : public VehicleFactory {  public:  std::unique\_ptr<Vehicle> CreateVehicle() const override {  return std::make\_unique<Buggy>();  }  };  class MonsterTruckFactory : public VehicleFactory {  public:  std::unique\_ptr<Vehicle> CreateVehicle() const override {  return std::make\_unique<MonsterTruck>();  }  };  class DaciaFactory : public VehicleFactory {  public:  std::unique\_ptr<Vehicle> CreateVehicle() const override {  return std::make\_unique<Dacia>();  }  };  class MotorcycleFactory : public VehicleFactory {  public:  std::unique\_ptr<Vehicle> CreateVehicle() const override {  return std::make\_unique<Motorcycle>();  }  }; |

* The client who wants to create an object (Car, Buggy, etc.) and use that object uses VehicleFactory.
* In our example shown below, the CreateAndDriveVehicle() function receives a reference to a VehicleFactory object, which could be any subclass of VehicleFactory (e.g., CarFactory, BuggyFactory).
* It calls CreateVehicle() on this factory object, which returns a std::unique\_ptr<Vehicle>. The specific type of vehicle (Car, Buggy, etc.) is determined by the factory, not by the client.
* The client code then calls Drive() on the vehicle object, demonstrating that it doesn't need to know the specifics of the vehicle’s instantiation—only how to use it via the common Vehicle interface.

|  |
| --- |
| void CreateAndDriveVehicle(const VehicleFactory& factory) {  // Create the vehicle using the factory method  const auto vehicle = factory.CreateVehicle();  vehicle->Drive();  }  int main() {  const CarFactory carFactory;  const BuggyFactory buggyFactory;  const MonsterTruckFactory monsterTruckFactory;  const DaciaFactory daciaFactory;  const MotorcycleFactory motorcycleFactory;  CreateAndDriveVehicle(carFactory);  CreateAndDriveVehicle(buggyFactory);  CreateAndDriveVehicle(monsterTruckFactory);  CreateAndDriveVehicle(daciaFactory);  CreateAndDriveVehicle(motorcycleFactory);  return 0;  } |

* Benefits of Using the Factory Method Here
  + **Scalability**: New vehicle types can be added easily by creating a new vehicle and a corresponding factory class without modifying existing code. If a new vehicle type (e.g., Truck) is introduced, you only need to add a new subclass (TruckFactory) implementing the CreateVehicle() method. The client code remains the same, making the system more maintainable.
  + **Encapsulation of Creation Logic**: The creation logic for each vehicle type is encapsulated within its respective factory class, making the codebase more organized and maintainable.
  + **Loose Coupling**: The client code remains decoupled from the specific vehicle implementations, making it more flexible and easier to maintain. Each class has a single responsibility: the factories handle object creation, while the client code handles object usage.

## Abstract Factory Desing Pattern

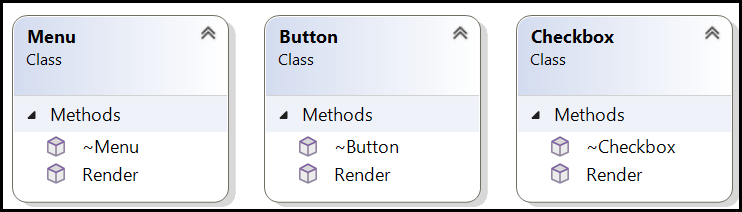
* A super-factory that creates other factories.
* It typically involves multiple factory methods for creating a variety of related objects, which are all part of a family.

**Problem Statement:**

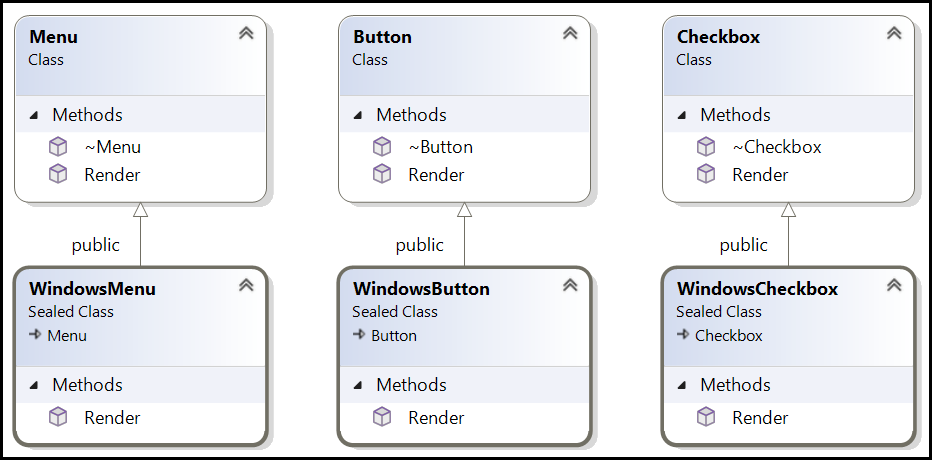
* You are developing a cross-platform Graphical User Interface (GUI) library that needs to support different operating systems such as **Windows**, **Mac** and **Linux**.
* The library includes common UI components like **buttons**, **checkboxes**, and **menus**, but each platform has a unique way of rendering these components to match the native look and feel of the operating system.
* Your goal is to create a system that can dynamically create and render UI components suitable for the target platform without changing the client code that uses these components.

Solution:

* We have 3 products at this time (Products can be added later also) which are Buttons, Checkboxes, and Menus. These can be treated as abstract products.



|  |
| --- |
| class Button {  public:  virtual ~Button() = default;  virtual void Render() const = 0;  };  class Checkbox {  public:  virtual ~Checkbox() = default;  virtual void Render() const = 0;  };  class Menu {  public:  virtual ~Menu() = default;  virtual void Render() const = 0;  }; |

* Windows has Buttons, Checkboxes, and Menus. So, it inherits all these abstract classes and implements WindowsButton, WindowsCheckbox, and WindowsMenu.

|  |
| --- |
| void WindowsButton::Render() const  {  std::cout << "Rendering Windows Button\n";  }  void WindowsCheckbox::Render() const  {  std::cout << "Rendering Windows Checkbox\n";  }  void WindowsMenu::Render() const  {  std::cout << "Rendering Windows Menu\n";  } |

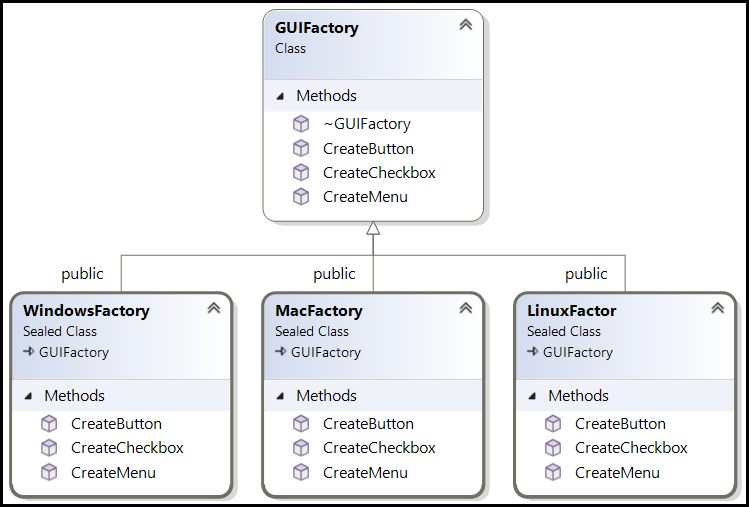
* Similarly, Max also inherits for the Buttons, Checkboxes, and Menus classes and implements MacButton, MaxCheckbox, and MacMenu.
* Similarly, Linux also inherits for the Buttons, Checkboxes, and Menus classes and implements LinuxButton, LinuxCheckbox, and LinuxMenu.



* Now we create an abstract factory. An interface that declares methods to create **families** of **components**.

|  |
| --- |
| class GUIFactory {  public:  virtual ~GUIFactory() = default;  virtual std::unique\_ptr<Button> CreateButton() const = 0;  virtual std::unique\_ptr<Checkbox> CreateCheckbox() const = 0;  virtual std::unique\_ptr<Menu> CreateMenu() const = 0;  }; |

* + **Components** refer to the individual products or objects that the factory creates. Buttons, Checkboxes, and Menus. Each component typically has an abstract base class (or interface) that defines its common behavior (e.g., Render()).
  + **Families** refer to groups of related components that are designed to work together within the same context, such as a specific operating system or platform.
  + Windows family include, WindowsButton, WindowsCheckbox and WindowsMenu.
  + CreateButton(): A method that creates a Button component specific to the family (e.g., WindowsButton or LinuxButton).
  + CreateCheckbox(): A method that creates a Checkbox component specific to the family (e.g., WindowsCheckbox or LinuxCheckbox).
  + CreateMenu(): A method that creates a Menu component specific to the family (e.g., WindowsMenu or LinuxMenu).
* We implement 3 concrete factories from this abstract factory namely WindowsFactory, MacFactory and LinuxFactor. These factories implement the GUIFactory interface and ensure that all created components (Button, Checkbox, Menu) belong to the same family (Windows or Mac or Linux).



* Imagine an application needs to render a user interface. The client code requests the components from a specific factory without knowing which concrete implementation is being used.

|  |
| --- |
| void RenderUI(const GUIFactory& factory) {  const auto button = factory.CreateButton();  const auto checkbox = factory.CreateCheckbox();  const auto menu = factory.CreateMenu();  button->Render();  checkbox->Render();  menu->Render();  } |

* If we pass, WindowsFactory object to the RenderUI() method, it would render Windows specific button, checkbox and menu.

|  |
| --- |
| int main() {  // Create Windows UI  const std::unique\_ptr<GUIFactory> factory =  std::make\_unique<WindowsFactory>();  RenderUI(\*factory);  return 0;  }  /\*  Rendering Windows Button  Rendering Windows Checkbox  Rendering Windows Menu  \*/ |

## Visual Studio 2022 Solution File

