ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

OBJECT-ORIENTED PROGRAMMING

REPORT OF PROJECT

Prototype Pattern: A Way to Clone an Object

Nhóm trưởng: Nguyễn Phúc Khang - 24120068
Thành viên: Hoàng Trọng Nghĩa - 24120103
Thành viên: Phan Chí Cao - 24120026
Thành viên: Nguyễn Hoàng Nhật - 24120110

Thành phố Hồ Chí Minh, 11/2025

Mục lục

1 Introduction

Trong lĩnh vực thiết kế phần mềm riêng và lập trình hướng đối tượng nói chung thì việc áp dụng design pattern vào dự án thực tế là rất cần thiết, vì nó cho phép ta có thể tái sử dụng và có các giải pháp được chứng minh đối với các bài toán thường gặp trong quá trình thiết kế và triển khai các hệ thống phần mềm. Design pattern là các giải pháp tổng quát đối với các vấn đề thường gặp xuất phát trong quá tình kiểm thử và phát triển hệ thống phần mềm. Nó cung cấp một tập các giải pháp đã được chứng minh cho các thử thách trong thiết kế và tối ưu các phương pháp tốt nhất cho việc phát triển phần mềm.

Design pattern được chia thành 3 nhóm khác nhau: **Creational, Structural, Behavioral**. Mỗi loại đều có những vai trò và nhiệm vụ khác nhau chẳng hạn như **Creational** sẽ trừu tượng hóa các quá trình khởi tạo, **Structural** sẽ liên quan tới các class và objects được kết hợp như thế nào để tạo thành các cấu trúc lớn hơn, hay đối với **Behavioral** thì sẽ liên quan tới các thuật toán và phân công trách nhiệm giữa các đối tượng. Và trong các loại trên, bài báo cáo này sẽ tập trung vào **Prototype** thuộc loại **Creational**. Đây là một trong những pattern phổ biến và mạnh mẽ ở trong các quy trình phát triển phần mềm. Nó cung một giải pháp tiện lợi, nhanh chóng và hiệu quả để có thể khởi tạo một object phức tạp, có thể bỏ qua được loại của lớp, cũng như là không cần biết quá chi tiết về logic bên trong của class đó [1].

2 GitHub Repository Link

Nhằm phục vụ cho việc quản lý dự án, cũng như là công khai các mã nguồn có trong dự án thì nhóm cũng đã đẩy toàn bộ dự án lên trên GitHub:

Link: See more

3 GitHub Commit List

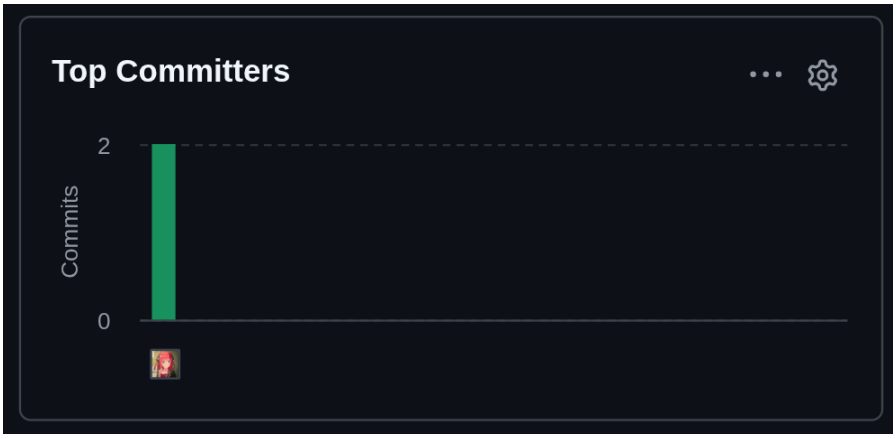
Để cung cấp một cái nhìn minh bạch và toàn diện về quá trình phát triển của dự án, dưới đây là toàn bộ lịch sử commit được trích xuất từ repository. Danh sách này được trình bày theo định dạng rút gọn, thể hiện các thay đổi, gộp nhánh và các cột mốc quan trọng từ khi bắt đầu cho đến khi hoàn thiện dự án.

```
commit a6e0fa047fe45823b7a9c6bdf619f8a54add5b44 (HEAD -> main, origin/main)
Merge: 46ccd09 70bc072
Author: Nguyen Phuc Khang <nguyenphuc.khang110806@gmail.com>
Date:   Mon Dec 1 23:50:45 2025 +0700

    Merge branch 'main' of https://github.com/khang1108/prototype-design-pattern

commit 46ccd097e39a24cde396dc24b6ff031c96ea81dc
Author: Nguyen Phuc Khang <nguyenphuc.khang110806@gmail.com>
Date:   Mon Dec 1 23:46:17 2025 +0700

    Initial commit

commit 70bc072e9e8158615007d6e5bb19ef252ece207b
Author: Nguyen Phuc Khang <94360818+khang1108@users.noreply.github.com>
Date:   Mon Dec 1 23:42:18 2025 +0700

    Initial commit
```

Listing 1: Lịch sử commit đầy đủ của dự án NaTruKi

4 GitHub Contribution

Bằng việc sử dụng GitHub để quản lý dự án, cũng như là tổ chức và xây dựng thì nhóm cũng đã có thể ghi lại những hoạt động và đóng góp của từng thành viên như sau:

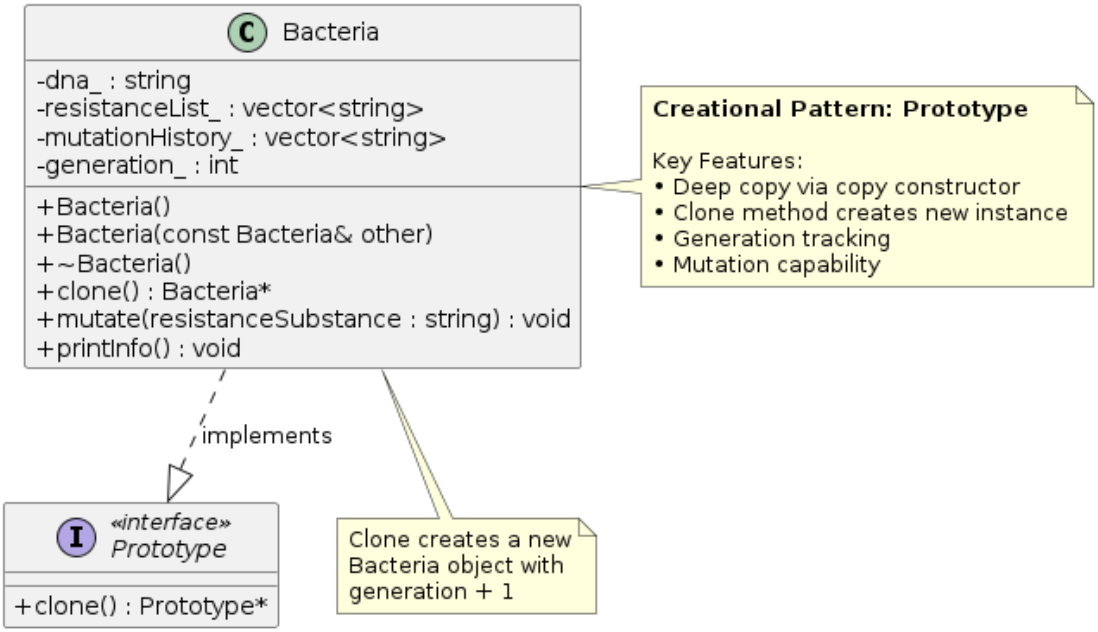


Hình 1: GitHub Contributions stats

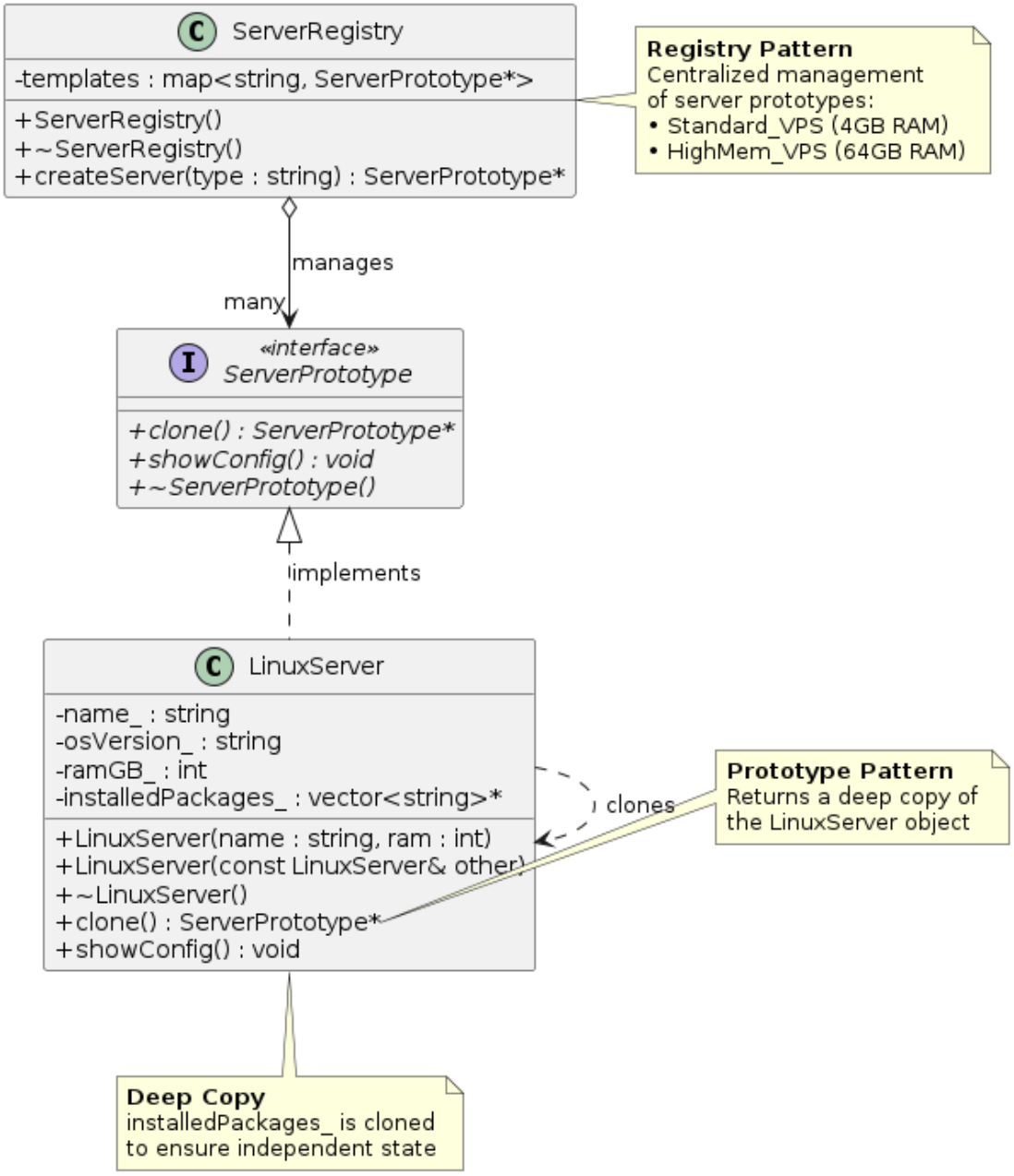
Qua hình ảnh này, ta có thể thấy được mức độ hoạt động của các thành viên trong nhóm. Vì dự án này không quá nặng về code nên mà thiên về tìm hiểu nội dung, các thành viên chủ yếu hoạt động bên Google Docs nên trên Github sẽ không quá nổi bật.

5 Class Diagram

Để trực quan hóa được ý tưởng về các tổ chức và xây dựng dự án này của nhóm thì ta có thể xem qua *Class Diagram - Unified Model Language (UML)*, đây chính là sơ đồ tổng thể toàn bộ các mối quan hệ của các class có trong dự án, cũng như là các thuộc tính, các phương thức được xây dựng cho mỗi class.



Hình 2: Bacteria UML Class Diagram



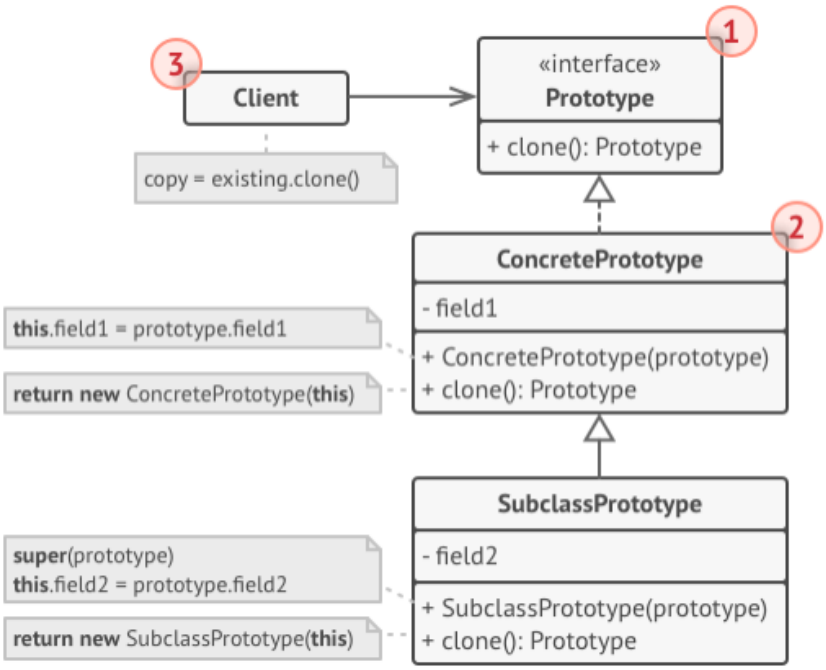
Hình 3: Registry Prototype UML Class Diagram

5.1 Basic Prototype Implementation

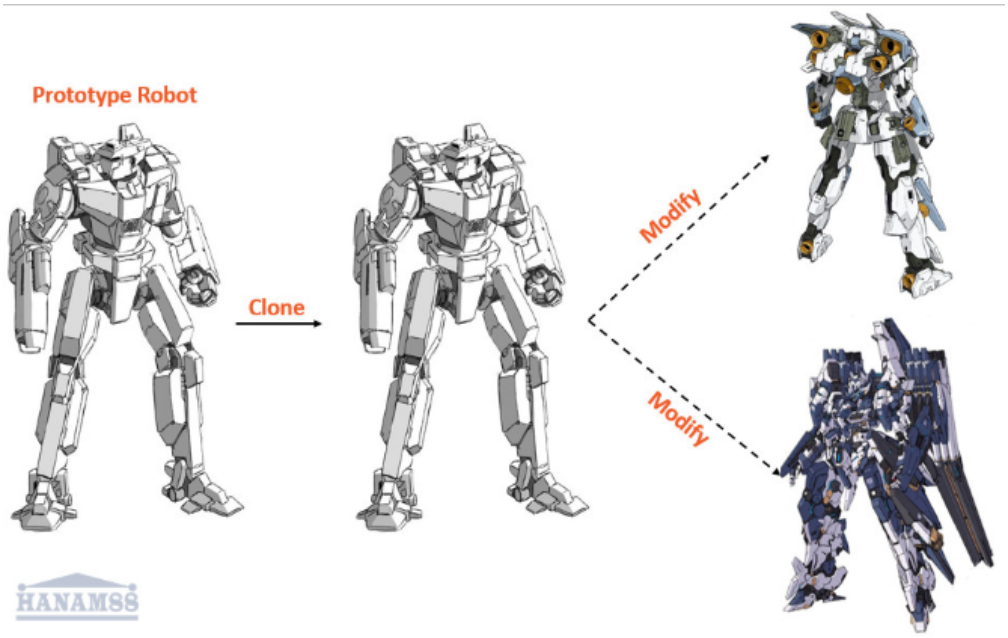
5.1.1 Định nghĩa

Đây là dạng thuần túy của mẫu thiết kế Prototype, nó tập trung vào kỹ thuật **Tự nhân bản**. Cấu trúc của nó bao gồm 3 phần chính đó là **Client**, **Prototype Interface** và **Concrete Prototype** với các nhiệm vụ khác nhau như sau [2]:

- **Prototype Interface:** Cấu trúc này có nhiệm vụ là định nghĩa các phương thức cho việc nhân bản *cloning* các đối tượng và đặt các tiêu chuẩn sao cho tất cả các **concrete prototypes** phải tuân theo. Nó bao gồm một phương thức `clone()` là nơi mà các **concrete prototypes** sẽ triển khai để tạo một bản sao của nó.
- **Concrete Prototypes:** Lớp này sẽ triển khai các `prototype interface` hoặc là mở rộng lớp trừu tượng. Nó biểu diễn một đối tượng có kiểu đặc biệt có thể được nhân bản.
- **Client:** Người dùng là đoạn mã nguồn hoặc là một module nào đó sẽ yêu cầu việc khởi tạo một đối tượng bới bằng cách giao tiếp với **prototype**.
- **Clone method:** Đây sẽ là phương thức được định nghĩa bên trong `prototype interface` hoặc là `abstract class` và chỉ ra rằng một đối tượng phải được nhân bản như nào. Các `concrete prototypes` sẽ triển khai phương thức này để định nghĩa thuộc tính nhân bản riêng biệt cho bản thân nó.



Hình 4: Cấu trúc của Basic Prototype Implementation



Hình 5: Hình ảnh ẩn dụ cho Prototype

Chẳng hạn như hình ảnh ẩn dụ thực tế về Protype ở trên, ban đầu ta có một mẫu được gọi là `Prototype` có các thuộc tính và thông tin cơ bản về class này. Sau đó `Client` sẽ tiến hành gọi `clone()` để có thể nhân bản ra các bản sao khác để tạo ra thêm các mẫu rời rạc khác từ đó mà có thể tùy chỉnh các thông tin cho các mẫu đó.

Việc áp dụng Prototype không chỉ giúp cho `Client` có thể dễ dàng khởi tạo một đối tượng mới dựa trên đối tượng mẫu mà không cần biết chi tiết về bên trong lớp đó có những gì. Từ đó mà giúp cho code của ta có thể gọn hơn, dễ chỉnh sửa cũng như là sửa lỗi. Tóm gọn lại Protype sẽ có các lợi ích như sau [3]:

- **Efficient Object Creation:** Prototype pattern cho phép ta một cách khởi tạo các object một cách hiểu quả hơn thông qua việc cho phép các đối tượng có thể `clone()` thay vì là khởi tạo từ đầu. Nó có thể cải thiện hiệu suất một cách mạnh mẽ, đặc biệt là nếu các đối tượng phức tạp.
- **Reduced Subclassing:** Thay vì là ta phải tạo nhiều subclass khác nhau để tạo các đối tượng khác nhau, thì với `Prototype` thì ta chỉ cần xây dựng phương thức `clone()`. Nó giúp ta giải được sự bùng nổ subclass, cũng như là đơn giản hóa các lớp kế thừa và làm cho `codebase` có thể dễ bảo trì hơn.

- **Flexibility:** Nhân bản các đối tượng cho phép ta có thể tùy chỉnh các thông tin đối tượng một cách độc lập, cũng như là thay đổi `dynamic runtime changes` không ảnh hưởng tới đối tượng nguyên bản.
- **Promotes Reusability:** Bằng cách cung cấp các kỹ thuật `clone` thì `Prototype` pattern có thể tái sử dụng. Các đối tượng được nhân bản có thể bảo toàn như là các mẫu ban đầu với sự tương đồng về thuộc tính, hành vi, từ đó mà có thể tiết kiệm thời gian và nguồn lực.

5.1.2 Code Demo

```cpp
#include <iostream>
#include <vector>
#include <string>

#define RED "\033[31m"
#define YELLOW "\033[33m"
#define GREEN "\033[32m"
#define RESET "\033[0m"

class Bacteria{
private:
    std::string dna_;
    std::vector<std::string> resistanceList_;
    std::vector<std::string> mutationHistory_;
    int generation_;
public:
    /// @brief : Default constructor
    Bacteria() : dna_(""), resistanceList_({}), mutationHistory_({}), generation_(0){
        std::cout << "\n[INFO] Initialize a " << RED << "'bacteria'" << RESET << " object via " <<
                YELLOW << "'default constructor'" << RESET << '\n';
    }
    /// @brief : Copy constructor using Deep Copy
    Bacteria(const Bacteria& other){
        dna_ = other.dna_;
        resistanceList_ = other.resistanceList_;
```

```
26          mutationHistory_ = other.mutationHistory_;
27          generation_ = other.generation_;
28
29          std::cout << "\n[INFO] Initialize a " << RED << "'bacteria'" << RESET << "
    object via " <<
30                  YELLOW << "'copy constructor'" << RESET << '\n';
31      }
32      /// @brief : Destructor
33      ~Bacteria(){
34          std::cout << "\n[INFO] Deleting the" << RED << "'bacteria'" << RESET << "
    object\n";
35      }
36      /// @brief : Clone method
37      Bacteria* clone() {
38          std::cout << "Cloning " << RED << "'bacteria'" << RESET << " object via "
39                  << YELLOW << "'clone()'" << RESET << "method\n";
40
41          Bacteria* newBacteria = new Bacteria(*this);
42
43          newBacteria->generation_ = this->generation_ + 1;
44
45          return newBacteria;
46      }
47
48      ///< @brief : Mutating
49      void mutate(const std::string& resistanceSubstance) {
50          this->dna_ += "_Mutated";
51          this->resistanceList_.push_back("Resist-" + resistanceSubstance);
52          this->mutationHistory_.push_back("Mutate-" + resistanceSubstance);
53      }
54
55      void printInfo() {
56          std::cout << "--- Bacteria Info ---" << "\n";
57          std::cout << "Gen: " << dna_ << "\n";
58          std::cout << "Generation: " << generation_ << "\n";
59          std::cout << "Resistance: ";
```

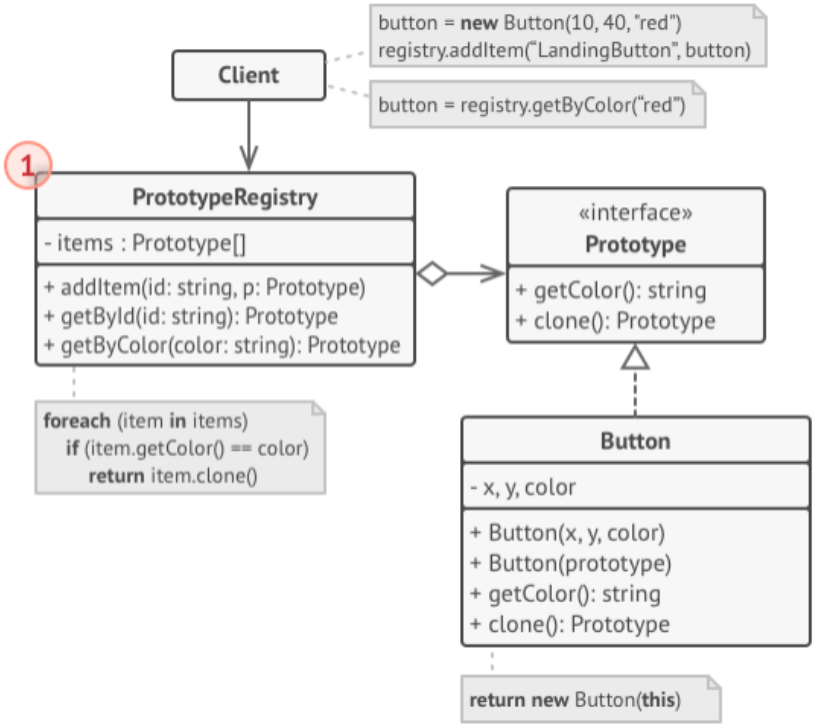
```
60          for (const auto& r : resistanceList_) std::cout << r << " ";
61          std::cout << "\n" << "\n";
62      }
63  };
64  int main()
65  {
66      std::ios_base::sync_with_stdio(false);
67
68      Bacteria* mother = new Bacteria();
69
70      mother->mutate("Penicillin");
71      mother->mutate("Ampicillin");
72      mother->mutate("Tetracycline");
73      mother->mutate("Streptomycin");
74
75      std::cout << "Mother after mutated:" << std::endl;
76      mother->printInfo();
77
78      std::cout << "Child cloned from Mother:" << std::endl;
79      Bacteria* child = mother->clone();
80
81      child->mutate("Chloramphenicol");
82
83      child->printInfo();
84
85      delete child;
86      delete mother;
87
88      return 0;
89  }
```

5.1.3 Output

```
[INFO] Initialize a 'bacteria' object via 'default constructor'
Mother after mutated:
-- Bacteria Info --
Gen: _Mutated_Mutated_Mutated_Mutated
Generation: 0
Resistance: Resist-Penicillin Resist-Ampicillin Resist-Tetracycline Resist-Streptomycin


Child cloned from Mother:
Cloning 'bacteria' object via 'clone()' method


[INFO] Initialize a 'bacteria' object via 'copy constructor'
-- Bacteria Info --
Gen: _Mutated_Mutated_Mutated_Mutated_Mutated
Generation: 1
Resistance: Resist-Penicillin Resist-Ampicillin Resist-Tetracycline Resist-Streptomycin
Resist-Chloramphenicol
```

5.2 Registry Prototype Implementation

Đây là dạng mở rộng hơn của Prototype, ta tiến hành kết hợp 2 pattern lại với nhau cụ thể trong trường hợp này là kết hợp giữa `Registry` và `Prototype` pattern. Mục tiêu của việc triển khai này sẽ tập trung vào việc ta có thể **quản lý và tái sử dụng**. Nó cung cấp cho ta một cách để có thể truy cập các `Prototype` thường dùng. Các `Prototypes` ấy sẽ được lưu trữ trong một tập hợp được xây dựng từ trước và luôn sẵn sàng để có thể được nhân bản [4].



Hình 6: Cấu trúc cho Registry Implementation

Trong cấu trúc này ta sẽ tạo thêm một lớp `PrototypeRegistry` để có thể lưu trữ các

5.2.1 Registry Pattern

Registry Pattern là một hướng đi `cấu trúc hóa` được sử dụng để tập trung hóa và quản lý truy cập tới các phần tử được chia sẻ hoặc là các phiên bản bên trong một phần mềm[5]. Mẫu thiết kế này nó liên quan tới việc khởi tạo một `registry` (thanh ghi) là kho lưu trữ trung tâm hoặc toàn cục, nơi các đối tượng sẽ đăng ký và lưu trữ với các mã định danh duy nhất.

Việc kết hợp `Registry` và `Prototype` tạo cho ta sự tiện lợi khi nó đã tiền xây dựng sẵn và khi cần ta chỉ cần truy cập tới đối tượng ta cần nhân bản thông quan một `key`, thông thường các đối tượng ấy sẽ được lưu trữ bên trong một `Hash Map`. Nhờ sự kết hợp đó nó cho ta thấy được các lợi ích như sau:

- **Access Efficiency:** Khi được triển khai đúng cách, `Registry` sẽ cho ta thấy được hiệu quả được cải thiện rõ rệt bằng việc cung cấp một các truy cập tối ưu đến các tài nguyên được chia sẻ. Các thành phần sẽ có thể được truy vấn các đối tượng một cách nhanh chóng từ thanh ghi mà không cần phải khởi tạo hoặc là tra cứu chúng nhiều lần.
- **Caching and Optimizations:** `Registry` có thể kết hợp với các chiến lược lưu bộ nhớ đệm `caching` để tối ưu hiệu quả. Nhờ vào điều này mà nhu cầu về các truy vấn lặp đi lặp lại và khởi tạo các đối tượng được giảm đi rõ rệt.
- **Centralized Resource Management:** Nhờ vào việc tập trung các tài nguyên chia sẻ vào một chỗ tại `Registry`, nên mẫu thiết kế này có thể đơn giản hóa cho việc tinh chỉnh quá trình có thể thêm hoặc bớt một cách dễ dàng. Thay vì là ta phải quản lý các nguồn tài nguyên trên nhiều đối tượng, nhiều nodes khác nhau thì bây giờ tất cả đều tập trung tại một `Registry` đơn lẻ.

5.2.2 Demo Code

```
#include <iostream>
#include <string>
#include <map>
#include <thread>
#include <chrono>
#include <map>
#include <vector>
```

```
8
9   #define RED "\033[31m"
10  #define YELLOW "\033[33m"
11  #define GREEN "\033[32m"
12  #define RESET "\033[0m"
13
14  class ServerPrototype
15  {
16  public:
17      ///< @brief: Pure virtual allowing each subclass to define its own method
18      virtual ServerPrototype* clone() const = 0;
19      virtual void showConfig() = 0;
20      virtual ~ServerPrototype() {}
21  };
22
23  class LinuxServer : public ServerPrototype {
24  private:
25      std::string name_;
26      std::string osVersion_;
27      int ramGB_;
28      std::vector<std::string>* installedPackages_;
29  public:
30      LinuxServer(std::string name, int ram) : name_(name), ramGB_(ram) {
31          this->osVersion_ = "Linux 6.17.4-arch2-1";
32          installedPackages_ = new std::vector<std::string>{"nginx", "mysql", "php"};
33
34          std::cout << "[System] Installing OS for Prototype " << name_
35                  << "via " << RED << "'default  constructor'" << RESET << '\n';
36      }
37
38      LinuxServer(const LinuxServer& other){
39          name_ = other.name_;
40          osVersion_ = other.osVersion_;
41          ramGB_  =  other.ramGB_;
42
43          installedPackages_ =  new std::vector<std::string>(*other.installedPackages_);
```

```
44
45          std::cout << "[System] Installing OS for Prototype " << name_
46                  << "via " << RED << "'copy  constructor'" << RESET << '\n';
47      }
48
49      ServerPrototype* clone() const override {
50          return new LinuxServer(*this);
51      }
52
53      void showConfig() override {
54          std::cout << "Server: " << name_ << " | OS: " << osVersion_ << " | RAM: " <<
    ramGB_ << "GB" << std::endl;
55      }
56
57      ~LinuxServer(){
58          delete installedPackages_;
59      }
60  };
61
62  class ServerRegistry
63  {
64  private:
65      std::map<std::string,  ServerPrototype*> templates;
66
67  public:
68      ServerRegistry() {
69          std::cout <<  "=== STARTING REIGSTRY ===" << std::endl;
70
71          templates["Standard"] = new LinuxServer("Standard_VPS", 4);
72          templates["HighMem"] = new LinuxServer("HighMem_VPS", 64);
73
74          std::cout << "=== COMPLETED INITIALIZATION ===" << std::endl;
75      }
76
77      ~ServerRegistry() {
78          for (auto const& [key, val] : templates) {
```

```
79              delete val;
80          }
81          templates.clear();
82      }
83
84      ServerPrototype* createServer(std::string type){
85          if(templates.find(type) != templates.end()){
86              return templates[type]->clone();
87          }
88          return nullptr;
89      }
90  };
91
92  int main() {
93      std::ios_base::sync_with_stdio(false);
94
95      ServerRegistry registry;
96
97      std::cout << "[INFO] Create a " << RED << "'Standard'" << RESET
98              << " server\n";
99      ServerPrototype* standardServer = registry.createServer("Standard");
100     if(standardServer) standardServer->showConfig();
101
102     std::cout << "[INFO] Create a " << RED << "'HigMem'" << RESET
103             << " server\n";
104     ServerPrototype* highMem = registry.createServer("HighMem");
105     if(highMem) highMem->showConfig();
106
107     std::cout << "[INFO] Create a " << RED << "'Standard'" << RESET
108             << " server\n";
109     ServerPrototype* standardServer2 = registry.createServer("Standard");
110     if(standardServer2) standardServer2->showConfig();
111
112     delete standardServer;
113     delete highMem;
114     delete standardServer2;
```

```
115
116     return 0;
117 }
```

5.2.3 Output

```
=== STARTING REIGSTRY ===
[System] Installing OS for Prototype Standard_VPSvia 'default constructor'
[System] Installing OS for Prototype HighMem_VPSvia 'default constructor'
=== COMPLETED INITIALIZATION ===


 [INFO] Create a 'Standard' server
[System] Installing OS for Prototype Standard_VPSvia 'copy constructor'
Server: Standard_VPS | OS: Linux 6.17.4-arch2-1 | RAM: 4GB


 [INFO] Create a 'HigMem' server
[System] Installing OS for Prototype HighMem_VPSvia 'copy constructor'
Server: HighMem_VPS | OS: Linux 6.17.4-arch2-1 | RAM: 64GB


 [INFO] Create a 'Standard' server
[System] Installing OS for Prototype Standard_VPSvia 'copy constructor'
Server: Standard_VPS | OS: Linux 6.17.4-arch2-1 | RAM: 4GB
```

6 Feature Checklist

Bảng 1: Checklist các hạng mục công việc cần thực hiện

Danh mục	ID	Nội dung công việc	Ưu tiên	Trạng thái
Documentation	D.1	**Main Report**: Viết báo cáo chính của dự án.	M	Done
	D.2	**Slide**: Thiết kế slide thuyết trình.	M	Done
Implementation	C.1	**Basic Prototype Demo Code**: Cài đặt mẫu Prototype cơ bản.	M	Done
	C.2	**Registry Prototype Demo Code**: Cài đặt Registry quản lý mẫu.	M	Done
Media	V.1	**Video**: Quay video demo sản phẩm.	M	Done

7 Conclusion

Prototype Pattern là một mẫu thiết kế phổ biến và mạnh mẽ, được ứng dụng nhiều trong thực tế đặc biệt là trong lĩnh vực phát triển phần mềm. Nó cho phép người dùng có thể sử dụng `clone()` để có thể khởi tạo một đối tượng mới mà không cần biết quá cụ thể về đối tượng, đặc

biệt là sẽ làm đơn giản vấn đề nếu như đối tượng đó quá phức tạp chứa nhiều lớp chồng lớp, cấu trúc chồng cấu trúc, hoặc là có chứa các logic phức tạp tốn nhiều thời gian để thực hiện khởi tạo.

Mẫu thiết kế này thể hiện được hiệu quả mạnh mẽ nếu như được xây dựng đúng cách nhưng nó sẽ khá phức tạp cho việc xây dựng `deep copy` yêu cầu có một kỹ năng thực tế và một kiến thức về thuật toán vững chắc. Prototype cung cấp cho ta một giải pháp khởi tạo đối tượng hiệu quả, giảm thiểu được bùng nổ các lớp trong quá trình phát triển, cũng như là tạo cho ta sự tùy biến và cho ta khả năng tái sử dụng các chức năng và thông tin của đối tượng.Done

Ngoài ra, ta có thể tối ưu hiệu quả của Prototype pattern bằng cách kếp hợp với Registry Pattern. Nó cho khả năng lưu trữ bộ nhớ đệm với một số đối tượng cụ thể, và các đối tượng này có thể được truy cập thông qua một `key` định danh duy nhất, và sau này nếu ta cần nhân bản đối tượng thì ta chỉ cần truy cập tới đối tượng ấy thông qua một `key`, việc này giúp cho ta có thể cải thiện tốc độ xử lý và tiết kiệm bộ nhớ.

Tài liệu tham khảo

[1] GeeksforGeeks, "Prototype Design Pattern." https://www.geeksforgeeks.org/system-design/prototype-design-pattern/, 2024. Accessed: Dec. 2, 2025.

[2] Patterns.dev, "Prototype pattern." https://www.patterns.dev/posts/prototype-pattern, 2024. Accessed: Dec. 2, 2025.

[3] A. Faraz, "Prototype pattern: Pros and cons." https://example.com/prototype-pros-cons, 2024. Accessed: Dec. 2, 2025.

[4] K. Ramjas, "Understanding prototype pattern in software design." https://example.com/prototype-pattern, 2023. Accessed: Dec. 2, 2025.

[5] GeeksforGeeks, "Registry design pattern." https://www.geeksforgeeks.org/registry-design-pattern/, 2024. Accessed: Dec. 2, 2025.