

Prototype Design Pattern - A Way to Clone an Object

Khang P. Nguyen¹, Nghia T. Hoang¹, Cao C. Phan¹ and Hoang N. Nguyen¹

¹University of Science - Vietnam National University

December 1, 2025

Outline

1. Introduction
2. Structures
3. Real-life examples
4. Conclusion
5. Conclusion

Outline

1. Introduction
2. Structures
3. Real-life examples
4. Conclusion
5. Conclusion

What is Prototype Pattern

Definition

Prototype Pattern is a **creational** design pattern that enables object duplication through **cloning** rather than **instantiation** (Chan, 2025)

What is Prototype Pattern

Definition

Prototype Pattern is a **creational** design pattern that enables object duplication through **cloning** rather than **instantiation**

Why should we use it?

This approach is particularly **useful** when object creation is **costly**, objects have **numerous** configurations, or you want to **decouple** object creation from its representation.

Problem

Description

You instantly need to create **1.000** objects `Solid` that has complicated *attributes, classes, and methods* such as (**Texture, 3D Model, Audio, Database, .etc**)

Problem

Description

You instantly need to create **1.000** objects `Solid` that has complicated *attributes, classes, and methods* such as (**Texture, 3D Model, Audio, Database, .etc**)

Naive Solution

Use a `for` loop `1000 times` to execute the command `new Soldier()`.

Problem

Description

You instantly need to create **1.000** objects `Solid` that has complicated *attributes, classes, and methods* such as (**Texture, 3D Model, Audio, Database, .etc**)

Naive Solution

Use a `for` loop `1000 times` to execute the command `new Soldier()`.

Problem

But for each time you initialize an object, which **MUST** load all of the data from disk (I/O), analyze configurations, and connect to the Database to get some attributes.

Problem

Description

You instantly need to create **1.000** objects `Solid` that has complicated *attributes, classes, and methods* such as (**Texture, 3D Model, Audio, Database, .etc**)

Naive Solution

Use a `for` loop 1000 times to execute the command `new Soldier()`.

The Consequence

- Spend a lot of CPU/RAM resources, **lag**, or "**not responding**" error.

Optimized Approach

Prototype

Create a single **prototype** object with all heavy assets **already loaded**. Then, simply `clone` it when needed. (GeeksforGeeks,)

This approach saves costly resources and time, especially when object creation is a **heavy** process.

Optimized Approach

Prototype

Create a single **prototype** object with all heavy assets **already loaded**. Then, simply `clone` it when needed. (GeeksforGeeks,)

This approach saves costly resources and time, especially when object creation is a **heavy** process.

Suppose a user creates a document with a specific layout, fonts, and styling, and wishes to create similar documents with slight modifications.

Optimized Approach

Document and Content Management Systems can use the prototype pattern to manage document templates. Users can `clone` an existing template and then make specific modifications.

Optimized Approach

Document and Content Management Systems can use the prototype pattern to manage document templates. Users can `clone` an existing template and then make specific modifications.

Game engines can use them to frequently `clone` complex characters or terrain objects. The Prototype approach allows efficient duplication without repeating costly initialization.

Analogy Example

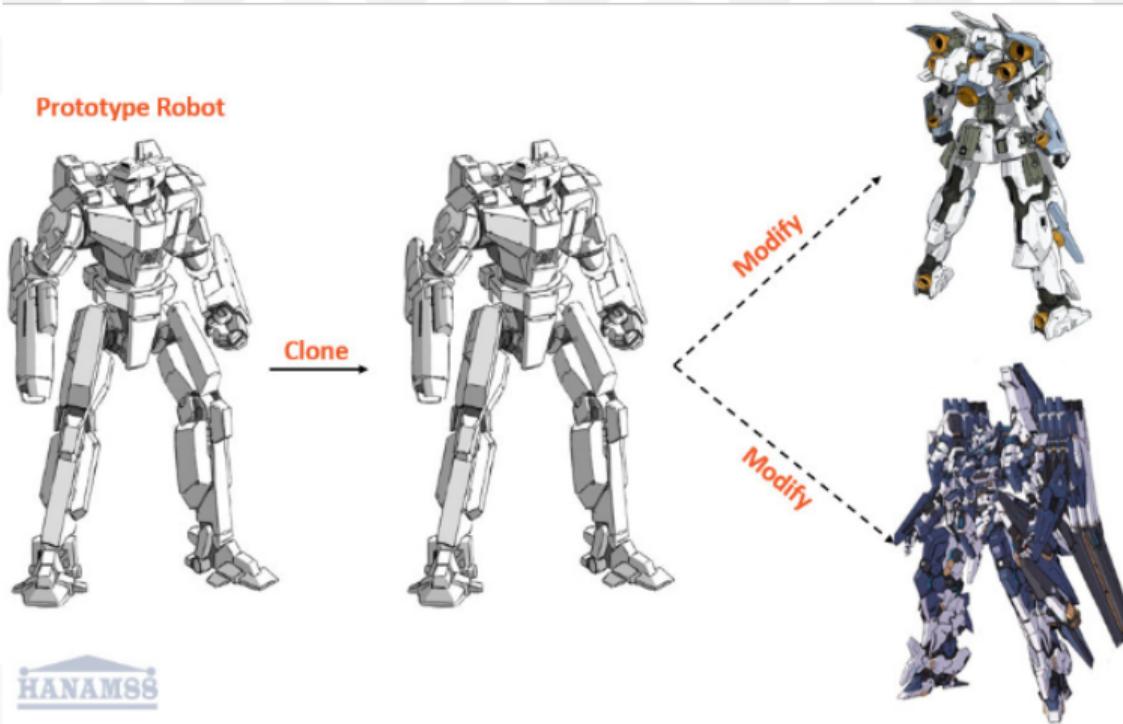


Figure 1 – Analogy Example for Prototype Pattern.

Basic Prototype Implementation

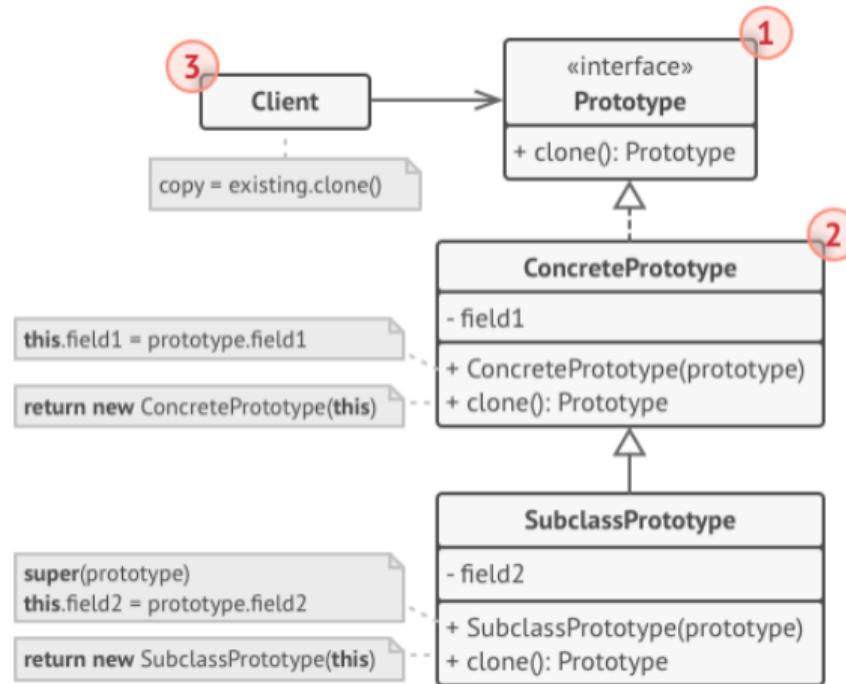


Figure 2 – Structure for basic implementation.

Basic Prototype Implementation

Components

- **Prototype Interface:** You need to define how the object will be **clone** by using `clone()`

Basic Prototype Implementation

Components

- **Prototype Interface:** You need to define how the object will be **clone** by using `clone()`
- **Concrete Prototype:** Implements the cloning method and stores the object data.

Basic Prototype Implementation

Components

- **Prototype Interface:** You need to define how the object will be **clone** by using `clone()`
- **Concrete Prototype:** Implements the cloning method and stores the object data.
- **Client:** Uses the `clone()` method to create new objects

Implementation of Bacteria Class

```
1 class Bacteria{
2 private:
3     std::string dna_;
4     std::vector<std::string> resistanceList_;
5     std::vector<std::string> mutationHistory_;
6     int generation_;
7 public:
8     /// @brief : Destructor
9     ~Bacteria(){
10         std::cout << "\n[INFO] Deleting the" << RED << "'bacteria'" << RESET <<
11         " object\n";
12     }
13 };
```

Implementation of Bacteria Class

```
1 //;< @brief : Mutating
2 void mutate(const std::string& resistanceSubstance) {
3     this->dna_ += "_Mutated";
4     this->resistanceList_.push_back("Resist-" + resistanceSubstance);
5     this->mutationHistory_.push_back("Mutate-" + resistanceSubstance);
6 }
7
8 void printInfo() {
9     std::cout << "--- Bacteria Info ---" << "\n";
10    std::cout << "Gen: " << dna_ << "\n";
11    std::cout << "Generation: " << generation_ << "\n";
12    std::cout << "Resistance: ";
13    for (const auto& r : resistanceList_) std::cout << r << " ";
14    std::cout << "\n" << "\n";
15 }
16
```

Implementation of Bacteria Class

```
1  /// @brief : Default constructor
2  Bacteria() : dna_(""), resistanceList_({}), mutationHistory_({}),
3  generation_(0){
4      std::cout << "\n[INFO] Initialize a " << RED << "'bacteria'" << RESET <<
5      " object via " <<
6          YELLOW << "'default constructor'" << RESET << '\n';
7  }
8
9  /// @brief : Copy constructor using Deep Copy
10 Bacteria(const Bacteria& other){
11     dna_ = other.dna_;
12     resistanceList_ = other.resistanceList_;
13     mutationHistory_ = other.mutationHistory_;
14     generation_ = other.generation_;
15
16     std::cout << "\n[INFO] Initialize a " << RED << "'bacteria'" << RESET <<
17     " object via " <<
18         YELLOW << "'copy constructor'" << RESET << '\n';
19 }
```

Implementation of Bacteria Class

```
1 // @brief : Clone method
2 Bacteria* clone() {
3     std::cout << "Cloning " << RED << "'bacteria'" << RESET << " object via "
4
5         << YELLOW << "'clone()'" << RESET << "method\n";
6
7
8     Bacteria* newBacteria = new Bacteria(*this);
9
10    newBacteria->generation_ = this->generation_ + 1;
11
12    return newBacteria;
}
```

Implementation of Bacteria Class

```
1 int main()
2 {
3     std::ios_base::sync_with_stdio(false);
4     Bacteria* mother = new Bacteria();
5
6     mother->mutate("Tetracycline");
7     mother->mutate("Streptomycin");
8
9     std::cout << "Mother after mutated:" << std::endl;
10    mother->printInfo();
11
12    std::cout << "Child cloned from Mother:" << std::endl;
13    Bacteria* child = mother->clone();
14    child->mutate("Chloramphenicol");
15
16    child->printInfo();
17    return 0;
18 }
```

Basic Implementation

```
[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
```

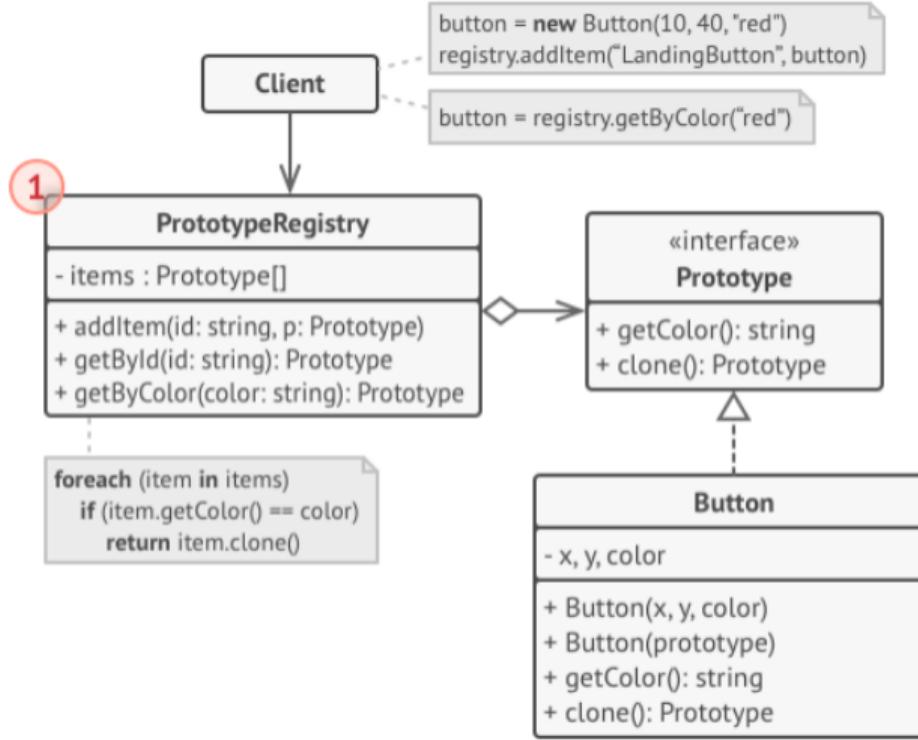
```
[INFO] Initialize a 'bacteria' object via 'copy constructor'  
--- Bacteria Info ---  
Gen: _Mutated_Mutated_Mutated_Mutated_Mutated  
Generation: 1  
Resistance: Resist-Penicillin Resist-Ampicillin Resist-Tetracycline Resist-Streptomycin  
Resist-Chloramphenicol
```

Basic Implementation

Conclusion

- Create objects **without** knowing their specific class.
- Avoiding complicated initialization.
- Client **DO NOT** need to know how to copy an object. All of these pieces are encapsulated inside the `clone()` method.

Registry Implementation



Registry Implementation

Definition

The **Registry Design Pattern** works *hand-in-hand* with the **Prototype Pattern** by acting as a **centralized store for prototypes**.

It allows you to register prototypes with unique keys and retrieve clones when needed, providing **flexibility** and avoiding tight **coupling** with specific classes (Ramjas, 2023).

Implementation of Server Class

```
1  class ServerPrototype
2  {
3      public:
4          ///@brief: Pure virtual allowing each subclass to define its own
5          // method
6          virtual ServerPrototype* clone() const = 0;
7          virtual void showConfig() = 0;
8          virtual ~ServerPrototype() {}
9      };
```

Implementation of Server Class

```
1  class LinuxServer : public ServerPrototype {
2  private:
3      std::string name_;
4      std::string osVersion_;
5      int ramGB_;
6  public:
7      LinuxServer(std::string name, int ram) : name_(name), ramGB_(ram) {
8          this->osVersion_ = "Linux 6.17.4-arch2-1";
9          std::cout << "[System] Installing OS for Prototype " << name_
10             << "via " << RED << "'default constructor'" << RESET << '\n';
11     }
12     LinuxServer(const LinuxServer& other){
13         name_ = other.name_;
14         osVersion_ = other.osVersion_;
15         ramGB_ = other.ramGB_;
16         std::cout << "[System] Installing OS for Prototype " << name_
17             << "via " << RED << "'copy constructor'" << RESET << '\n';
18     }
19 };
20 
```

Implementation of Server Class

```
1  ServerPrototype* clone() const override {
2      return new LinuxServer(*this);
3  }
4
5  void showConfig() override {
6      std::cout << "Server: " << name_ << " | OS: " << osVersion_ << " | RAM:
7      " << ramGB_ << "GB" << std::endl;
8 }
```

Implementation of Server Class

```
1  class ServerRegistry
2  {
3      private:
4          std::map<std::string, ServerPrototype*> templates;
5      public:
6          ServerRegistry() {
7              std::cout << "==== STARTING REIGSTRY ===" << std::endl;
8
9              templates["Standard"] = new LinuxServer("Standard_VPS", 4);
10             templates["HighMem"] = new LinuxServer("HighMem_VPS", 64);
11
12             std::cout << "==== COMPLETED INITIALIZATION ===" << std::endl;
13         }
14         ~ServerRegistry() {
15             for (auto const& [key, val]: templates) {
16                 delete val;
17             }
18             templates.clear();
19         }
20     };
21 }
```

Registry Prototype

Advantages

- **Reusability:** Prototypes are stored and reused, saving computational costs.

Registry Prototype

Advantages

- **Reusability:** Prototypes are stored and reused, saving computational costs.
- **Flexibility:** New prototypes can be added to the registry without altering existing code.

Registry Prototype

Advantages

- **Reusability:** Prototypes are stored and reused, saving computational costs.
- **Flexibility:** New prototypes can be added to the registry without altering existing code.
- **Decoupling:** Clients don't need to know the specific classes of object they work with.

Registry Prototype

Advantages

- **Reusability:** Prototypes are stored and reused, saving computational costs.
- **Flexibility:** New prototypes can be added to the registry without altering existing code.
- **Decoupling:** Clients don't need to know the specific classes of object they work with.
- **Centralized Management:** All prototypes are stored in one place, making them easy to manage.

When should we use?

When should we use?

- **Heavy Initialization:** Lower Initialization costs, as cloning faster than building a new object from scratch

When should we use?

When should we use?

- **Heavy Initialization:** Lower Initialization costs, as cloning faster than building a new object from scratch
- **Subclass Explosion:** Helpful for managing various objects with minor differences. Instead of creating multiple classes, you can clone and modify prototypes.

When should we use?

When should we use?

- **Heavy Initialization:** Lower Initialization costs, as cloning faster than building a new object from scratch
- **Subclass Explosion:** Helpful for managing various objects with minor differences. Instead of creating multiple classes, you can clone and modify prototypes.
- **Dynamic Configurations:** If you need to create objects at runtime, you can `clone` a base configuration and adjust it as necessary. (Faraz, 2024)

When should we not use?

When should we not use?

Deep Copy Challenges: If objects contain **complex nested** structures or references to other objects, implementing a `deep copy` mechanism for cloning can be challenging.

When should we not use?

When should we not use?

Deep Copy Challenges: If objects contain **complex nested** structures or references to other objects, implementing a **deep copy** mechanism for cloning can be challenging.

Increased Memory Usage: If objects are **large** or contain a **significant** amount of data, cloning objects can lead to increased memory usage.

When should we not use?

When should we not use?

Deep Copy Challenges: If objects contain **complex nested** structures or references to other objects, implementing a **deep copy** mechanism for cloning can be challenging. **Increased Memory Usage:** If objects are **large** or contain a **significant** amount of data, cloning objects can lead to increased memory usage. **Potential for Cloning Abuse:** Misuse or Excessive cloning of objects can lead to code complexity and maintenance issues.

Conclusion

Conclusion

The prototype pattern allows us to easily let objects **access** and **inherit** properties from other objects. Since the prototype chain allows us to access properties that aren't directly defined on the object itself, we can avoid **duplication** of methods and properties, thus **reducing** the amount of memory used (Patterns.dev,).

Conclusion

Conclusion

The prototype pattern allows us to easily let objects **access** and **inherit** properties from other objects. Since the prototype chain allows us to access properties that aren't directly defined on the object itself, we can avoid **duplication** of methods and properties, thus **reducing** the amount of memory used (Patterns.dev,).

Warning

While the Prototype pattern offers several benefits, it also comes with challenges related to **deep copying**, **memory usage** and **potential misuse**. But if you put it in the right scenarios, the Prototype pattern can be a valuable design tool for achieving flexibility and code reusability.

Thank You for Your Attention!

If you have any *questions*, please keep them in your *mind*.

Reference

CHAN, M. M. **Understanding the Prototype Design Pattern in C#**. 2025. <<https://chanmingman.wordpress.com/2025/11/30/understanding-the-prototype-design-pattern-in-c/>>. Accessed: Nov. 30, 2025.

FARAZ, K. **Prototype Design Pattern – Pros & Cons**. 2024. <<https://www.linkedin.com/pulse/prototype-design-pattern-pros-cons-kashif-faraz-hbjpf/>>. LinkedIn. Accessed: Dec. 01, 2025.

GeeksforGeeks. **Prototype Design Pattern**. <<https://www.geeksforgeeks.org/system-design/prototype-design-pattern/>>. Accessed: Nov. 30, 2025.

Patterns.dev. **Prototype Pattern**. <<https://www.patterns.dev/vanilla/prototype-pattern/>>. Accessed: Dec. 01, 2025.

RAMJAS, P. **Understanding the Prototype and Registry Design Patterns: A Comprehensive Guide with Examples**. 2023. <<https://medium.com/@ramjasprankur/understanding-the-prototype-and-registry-design-patterns-a-comprehensive-guide-with-examples-f9b27e81f5b3>>. Medium. Accessed: Dec. 01, 2025.