Prototype Design Pattern - A Way to Clone an Object

Khang P. Nguyen [1], Nghia T. Hoang [1], Cao C. Phan [1] and Hoang N. Nguyen [1]

[1]University of Science - Vietnam National University

December 2, 2025

Outline

Outline

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

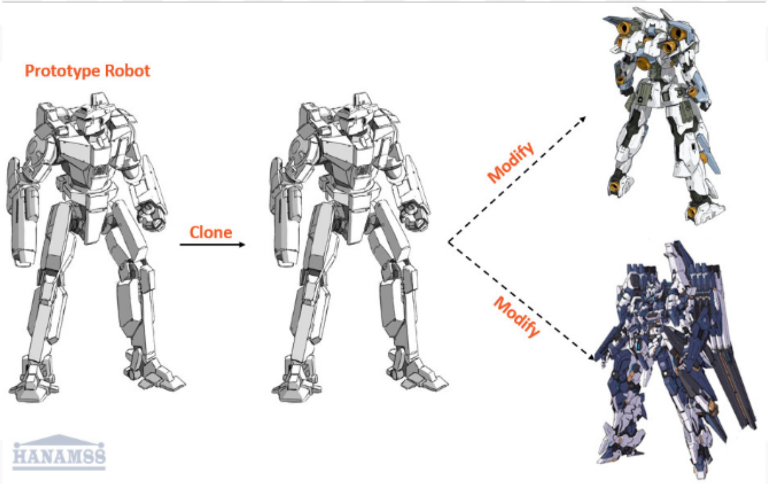


Figure 1 – Analogy Example for Prototype Pattern.

Basic Prototype Implementation

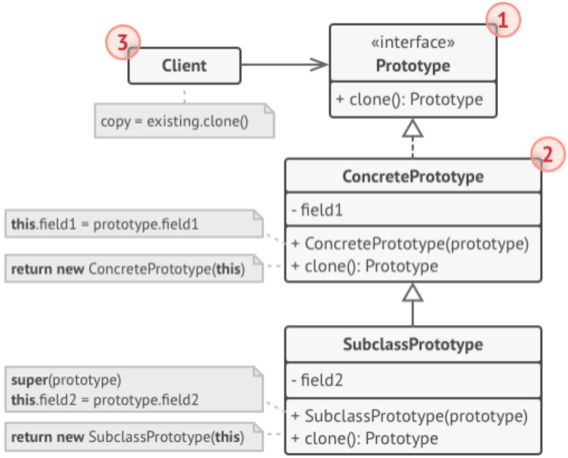


Figure 2 – Structure for basic implementation.

Basic Prototype Implementation

Components

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

Basic Prototype Implementation

Components

- **Prototype Interface**: You need to define how the objet 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 objet 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

```cpp
class Bacteria{
private:
    std::string dna_;
    std::vector<std::string> resistanceList_;
    std::vector<std::string> mutationHistory_;
    int generation_;
public:
    /// @brief : Destructor
    ~Bacteria(){
        std::cout << "\n[INFO] Deleting the" << RED << "'bacteria'" << RESET <<
    " object\n";
    }
};

```

Implementation of Bacteria Class

```cpp
    ///< @brief : Mutating
    void mutate(const std::string& resistanceSubstance) {
        this->dna_ += "_Mutated";
        this->resistanceList_.push_back("Resist-" + resistanceSubstance);
        this->mutationHistory_.push_back("Mutate-" + resistanceSubstance);
    }

    void printInfo() {
        std::cout << "--- Bacteria Info ---" << "\n";
        std::cout << "Gen: " << dna_ << "\n";
        std::cout << "Generation: " << generation_ << "\n";
        std::cout << "Resistance: ";
        for (const auto& r : resistanceList_) std::cout << r << " ";
        std::cout << "\n" << "\n";
    }

```

Implementation of Bacteria Class

```
    /// @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_;
        mutationHistory_ = other.mutationHistory_;
        generation_ = other.generation_;

        std::cout << "\n[INFO] Initialize a " << RED << "'bacteria'" << RESET <<
     " object via " <<
                YELLOW << "'copy constructor'" << RESET << '\n';
    }

```

Implementation of Bacteria Class

```cpp
    /// @brief : Clone method
    Bacteria* clone() {
        std::cout << "Cloning " << RED << "'bacteria'" << RESET << " object via "
                  << YELLOW << "'clone()'" << RESET << "method\n";

        Bacteria* newBacteria = new Bacteria(*this);

        newBacteria->generation_ = this->generation_ + 1;

        return newBacteria;
    }

```

Implementation of Bacteria Class

```cpp
int main()
{
    std::ios_base::sync_with_stdio(false);
    Bacteria* mother = new Bacteria();

    mother->mutate("Tetracycline");
    mother->mutate("Streptomycin");

    std::cout << "Mother after mutated:" << std::endl;
    mother->printInfo();

    std::cout << "Child cloned from Mother:" << std::endl;
    Bacteria* child = mother->clone();
    child->mutate("Chloramphenicol");

    child->printInfo();
    return 0;
}
```

Basic Implementation

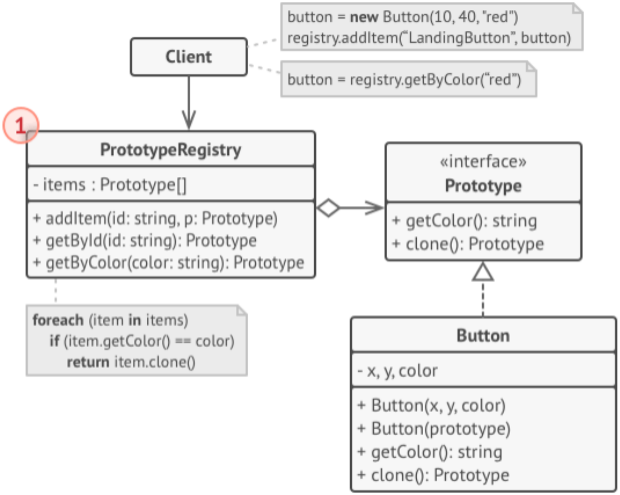
```
[INFO] Initialize a 'bacteria' object via 'default constructor'
Mother after mutated:
--- Bacteria Info ---
Gen:  _Mutated_Mutated_Mutated_Mutated
Generation:  0
Resistance:  Resist-Penicillin Resist-Ampicillin Resist-Tetracycline Resist-Streptomycin


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

```cpp
class ServerPrototype
{
public:
    ///< @brief: Pure virtual allowing each subclass to define its own method
    virtual ServerPrototype* clone() const = 0;
    virtual void showConfig() = 0;
    virtual ~ServerPrototype() {}
};
```

Implementation of Server Class

```
    class LinuxServer : public ServerPrototype {
    private:
        std::string name_;
        std::string osVersion_;
        int ramGB_;
    public:
        LinuxServer(std::string name, int ram) : name_(name), ramGB_(ram) {
            this->osVersion_ = "Linux 6.17.4-arch2-1";
            std::cout << "[System] Installing OS for Prototype " << name_
                << "via " << RED << "'default constructor'" << RESET << '\n';
        }
        LinuxServer(const LinuxServer& other){
            name_ = other.name_;
            osVersion_ = other.osVersion_;
            ramGB_  =  other.ramGB_;
            std::cout << "[System] Installing OS for Prototype " << name_
                    << "via " << RED << "'copy constructor'" << RESET << '\n';
        }
    };

```

Implementation of Server Class

```
    ServerPrototype* clone() const override {
        return new LinuxServer(*this);
    }

    void showConfig() override {
        std::cout << "Server: " << name_ << " | OS: " << osVersion_ << " | RAM: " << ramGB_ << "GB" << std::endl;
    }

```

Implementation of Server Class

```cpp
int main() {
    std::ios_base::sync_with_stdio(false);

    ServerRegistry registry;

    std::cout << "[INFO] Create a " << RED << "'Standard'" << RESET
              << " server\n";
    ServerPrototype* standardServer = registry.createServer("Standard");
    if(standardServer) standardServer->showConfig();

    std::cout << "[INFO] Create a " << RED << "'HigMem'" << RESET
              << " server\n";
    ServerPrototype* highMem = registry.createServer("HighMem");
    if(highMem) highMem->showConfig();

    std::cout << "[INFO] Create a " << RED << "'Standard'" << RESET
              << " server\n";
    ServerPrototype* standardServer2 = registry.createServer("Standard");
    if(standardServer2) standardServer2->showConfig();
}
```

Implementation of Server Class

```
=== 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
```

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.
- **Decouping:** 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.
- **Decouping:** 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 than building a new object from scratch

When should we use?

When should we use?

- **Heavy Initialization:** Lower Initialization costs, as cloning faster than 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 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, implementation 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, implementation 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, implementation 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 issuses.

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.