C++ Design Patterns

Łukasz Gut

Agenda

What are design patterns?

Design pattern - a characteristic solution to commonly occurring problem in software engineering and software design.

It is important to say, that a design pattern is not a complete solution, it’s more like a guideline or a template, that can be applied with different twist to a particular codebase.

Why should you learn design patterns?

Design patterns are programmers' best weapons when dealing with software design. They are tested and proven development paradigms.

Design patterns also shine in cooperative environment. They define a common language that can be used to better express design ideas and are commonly known across all kinds of developers throughout the world.

Classification

Design patterns are divided into three distinct groups:

1. Creational patterns - provide object creation mechanisms
2. Structural patterns - provide tips on how to organize your classes into larger structures
3. Behavioral patterns - identify common communication patterns between objects and realize these patterns

Singleton

Singleton - a **creational** design pattern that ensures existence of only a single instance of a particular class. It also provides a global access point to that instance.

Common usage:

- Application
- Global settings
- Single database connection
- Global event bus

Singleton

Pros:

- Global variable
- Provides easy to use shared resource interface
- Is initialized only once

Cons:

- Global variable
- Tricky in multithreaded world
- Violates SRP
- Can be a code smell if used careless

Factory

Factory - a **creational** design pattern that provides an interface for object creation in a base class and allows subclasses to alter the type actual type of object that will be created.

Common usage:

- Object pools

Factory

Pros:

- Decoupling of the creator and the object itself.
- *SRP Principle*. Single part of code is responsible for object creation.
- *Open/Closed Principle*. You can introduce new types of products into the program without breaking existing client code.

Cons:

- Boilerplate.

Builder

Builder - a **creational** design pattern that provides a way to construct highly complex objects. It is usually next step when factory pattern becomes too complex.

Common usage:

- Options GUI
- Basically if your constructor takes more than 3 arguments...

Builder

Pros:

- Let’s you construct objects step by step. No need for monstrous constructors.
- Deferred/recursive construction.
- *SRP Principle*. Object construction is decoupled into special entity.

Cons:

- Increased code complexity. Needs more classes.

Visitor

Visitor **-** a **behavioral** design pattern that lets you separate algorithms from the data that these algorithms operate on.

Common usage:

- Algorithms on complex data structures

Visitor

Pros:

- *SRP Principle.* Each visitor has a single responsibility.
- *Open/Closed Principle.* Visitor pattern let's us define new behaviors (open for extension), without changing objects that are operated on (closed for modification)

Cons:

- Each time a new class is introduced, you have to update code in your visitors.
- Visitors don't have access to private members (in C++ this is possible via friend relation, but is not recommended)

CRTP

CRTP (Curiously recurring template pattern) - a **behavioral** *C++* design pattern that lets you imitate dynamic dispatch via static binding, resulting in super fast code.

Common usage:

- When performance matters

CRTP

Pros:

- No polymorphism, everything is statically bound.
- Common interface, same as with standard polymorphism.

Cons:

- Not possible to store heterogeneous types in a single container (Although, I might have found a solution… :))
- Requires neat tricks to be well implemented.

Polymorphism with std::variant and std::visit

Polymorphism - ba bla.

Common usage:

-

Polymorphism with std::variant and std::visit

Pros:

- Value semantics, no dynamic allocation.
- No need for a base class, classes can be unrelated.
- Methods can have different signatures, is not possible with virtual ones.

Cons:

- All types need to be known at compile time.
- Each “virtual” method needs a separate visitor.
- Can potentially waste memory. Size of std::variant is the size of the largest struct.

Summary

Design patterns are an essential part of programmer's toolkit. They should be used to improve development speed and code quality.

Bibliography

- https://en.wikipedia.org/
- https://refactoring.guru/
- https://www.bfilipek.com/

Thank you.
Questions?