**Design Patterns**

TODO: consolidate singleton notes, add why nots, clean up code examples.

TODO: main.cpp client class at top of code example file.

# **Abstract Factory**

## Overview

Abstract Factory Design Pattern (AFDP)

* AFDP is a creational design pattern that defines an Abstract Factory Class that creates Families of related objects indirectly but doesn't specify their concrete sub-class (sub-classes are also factories).

Fundamental Design

* The general creation process is the same in the sub-classes as in the abstract parenty factory but the concrete sub-class factories create different but related classes.

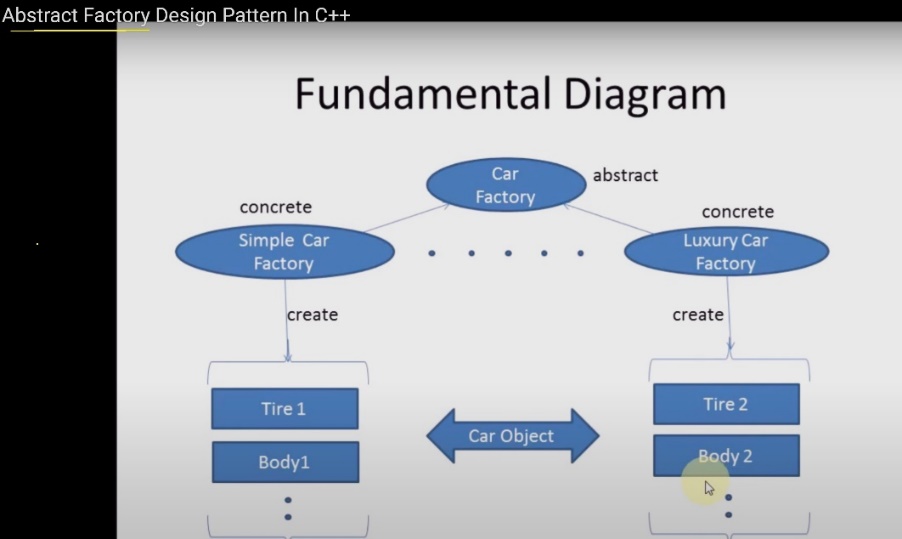
Why/When

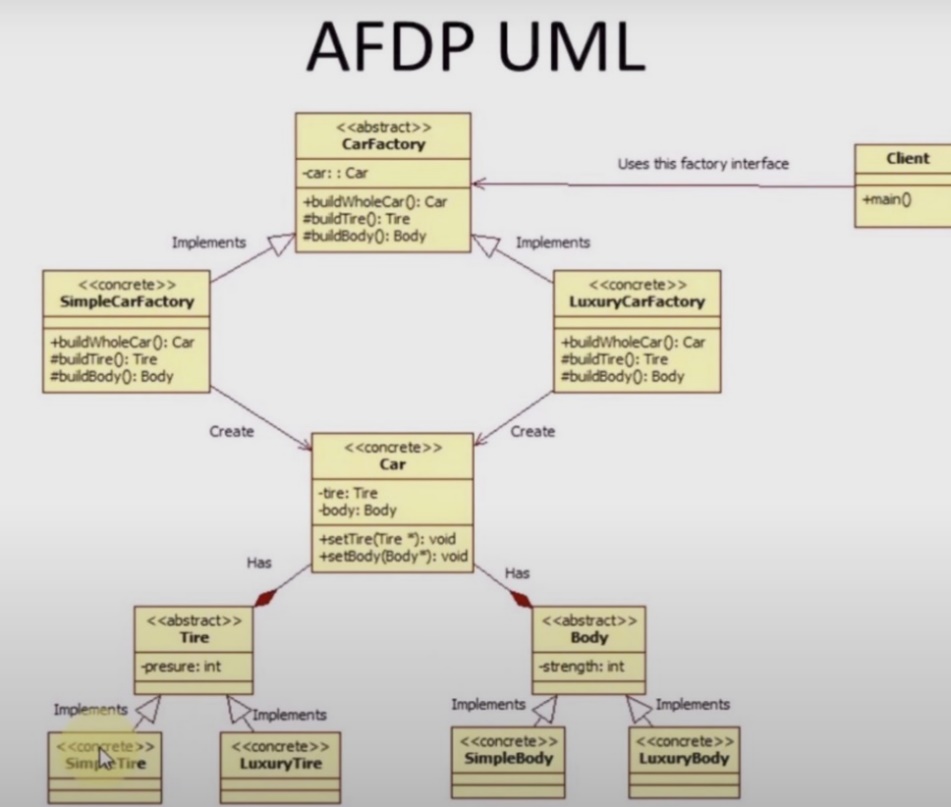
* Need a system to be independent of how objects are created and composed.
* Need a system configurable with multiple families of objects.
* Client is hidden from the creation process of the objects and it's composition.
* Show interface but not implementation.

Concrete Class v Abstract Class v Interface

* An **interface** has only pure virtual functions and no data and can’t be instantiated directly.
* An **abstract** class is meant to be used as a base class where at least one function is declared purely virtual and hence can not be instantiated..
* A **concrete** class is an ordinary class which has no purely virtual functions and hence can be instantiated directly. Concrete classes of an abstract class need to define the abstract classes’ pure virtual functions with an override keyword.

## Diagrams





## Code Example

### Main.cpp

////////////////// Abstract\_Factory.cpp (Main.cpp) ///////////////////////

// Abstract Factory Pattern | | [AbstractFactory\_video](https://www.youtube.com/watch?v=blDsmrsyOME&list=PLm0fFUL4gEt_Fxr324K9WswUff_Pub7CN&index=2&t=380s)

#include <iostream>

#include "CarFactory.cpp"

// Toggle Car type, commenting out factories I don’t’ want active.

#define SIMPLE\_CAR 1 // #define LUXURY\_CAR 1

int main() {

#ifdef SIMPLE\_CAR

CarFactory\* factory = new SimpleCarFactory;

#elif LUXURY\_CAR

CarFactory\* factory = new LuxuryCarFactory;

#endif

// Doesn't know which buildWholeCar() factory at compile time

Car\* car = factory->buildWholeCar();

car->printDetails(); // depending on which #define is uncommented up top.

### CarFactory.cpp

//////////////////////// CarFactory.cpp ///////////////////////////////////////

#include "Car.cpp"

class CarFactory {

public:

virtual Car\* buildWholeCar() = 0;

protected:

virtual Tire\* buildTire() = 0;

virtual Body\* buildBody() = 0;

private:

Car\* car;

};

class SimpleCarFactory : public CarFactory {

Tire\* buildTire() {

return new SimpleTire();

}

Body\* buildBody() {

return new SimpleBody();

}

Car\* buildWholeCar() {

Car\* car = new Car("SimpleCar");

car->setTire(buildTire());

car->setBody(buildBody());

return car;

}

};

class LuxuryCarFactory : public CarFactory {

Tire\* buildTire() {

return new LuxuryTire();

}

Body\* buildBody() {

return new LuxuryBody();

}

Car\* buildWholeCar() {

Car\* car = new Car("LuxuryCar");

car->setTire(buildTire());

car->setBody(buildBody());

return car;

}

};

### Car.cpp

//////////////////////// Car.cpp ///////////////////////////////////////

class Tire {

public:

Tire(string n, int presure) :name(n), presure(presure) {};

string getName() { return name; }

int getPresure() { return presure; }

protected:

string name;

int presure;

};

class SimpleTire : public Tire {

public:

SimpleTire() :Tire("SimpleTire", 75) {}

};

class LuxuryTire : public Tire {

public:

LuxuryTire() :Tire("LuxuryTire", 100) {}

};

class Body {

public:

Body(string n, int strength) :name(n), strength(strength) {}

string getName() { return name; }

int getStrength() { return strength; }

protected:

string name;

int strength;

};

class SimpleBody : public Body {

public:

SimpleBody() :Body("SimpleBody", 75) {}

};

class LuxuryBody : public Body {

public:

LuxuryBody() :Body("LuxuryBody", 100) {} // uses abstract class' initializer list contructor.

};

///// Car Class /////

class Car {

public:

Car(string n) :name(n) {}

void setTire(Tire\* t) { tire = t; }

void setBody(Body\* b) { body = b; }

void printDetails() {

cout << endl << "Car: " << name << endl;

cout << "Tire: " << tire->getName() << " Presure: " << tire->getPresure() << endl;

cout << "Body: " << body->getName() << " Strength: " << body->getStrength() << endl << endl;

}

protected:

string name;

Tire\* tire;

Body\* body;

};

///////////////////////////////////////////////////////////////////////////////

//////////////////////// CarFactory.cpp ///////////////////////////////////////

# **Adapter**

## Overiew

Adapter Pattern

* Structural design pattern that allows the interface of an existing class to be used as a another interface.
* Acts as a bridge class between two interfaces.

Physical Example

(USB Type A --> USB Type C adapter)

Functional Design

* For Example, if a Client wanted to call some external API which is not compatible. Client calls Paint(x,y); You want to use a library that has a Paint(x,y,z) function. Inside of Paint(x,y), there exists an interface adapter that calls Paint(x,y,z) from the external API library.
* Increases complexity by routing your call.

Why / When

* Integration of old code
* Code Reuse of existing functionality.
* Interoperability: bridging two incompatible interfaces
* Client-Server Communication: translate requests and responses between client and server.
* 3rd party APIs

## Diagrams



## Code Example

// Adapter.cpp | [Adapter YouTube Example](https://www.youtube.com/watch?v=BLaz_Nct1eI&list=PLk6CEY9XxSIDZhQURp6d8Sgp-A0yKKDKV&index=8)

// Example: you have an Indian socket but want to plug it into a USA socket.

#include <iostream>

#include <memory>

using namespace std;

// Socket we currently have

class IndianSocket

{

public:

// Can only define virtual functions in one socket Type

// Then in the adapter you override them.

// YOU can write as many functions you want to call other socket type functions.

virtual void indiancharge(int) = 0;

virtual void myOtherCharge() = 0;

};

// Socket which we want to use

class USASocket

{

public:

void usacharge() { std::cout << "USA Plug charging" << std::endl; }

};

// Socket which we want to use

class GSocket

{

public:

void germanyCharge() { std::cout << "German Plug adapter used" << std::endl; }

void gcharge()

{

std::cout << "G Plug charging" << std::endl;

}

// Need to abstract a virtual function here in order to define in the socket adapter

};

// This is the adapter, used to charge with USA Socket

class SocketAdapter : public IndianSocket, public USASocket, public GSocket

{

public:

void myOtherCharge()

{

germanyCharge();

}

void indiancharge(int type)

{

switch (type)

{

case 1:

usacharge();

break;

case 2:

gcharge();

break;

default:

break;

}

}

};

int main()

{

// client only supports Indian Charger.

// unique\_ptr<IndianSocket> socket = make\_unique<SocketAdapter>();

// socket->indiancharge(1); // you don't want to see what's there,

// socket->indiancharge(2);

unique\_ptr<IndianSocket> socket2 = make\_unique<SocketAdapter>();

socket2->myOtherCharge();

system("pause"); }

# **Bridge**

## Overview

Bridge Pattern

* Structural design pattern that  separate the abstraction from the implementation.
* Acts as a bridge class between two interfaces.

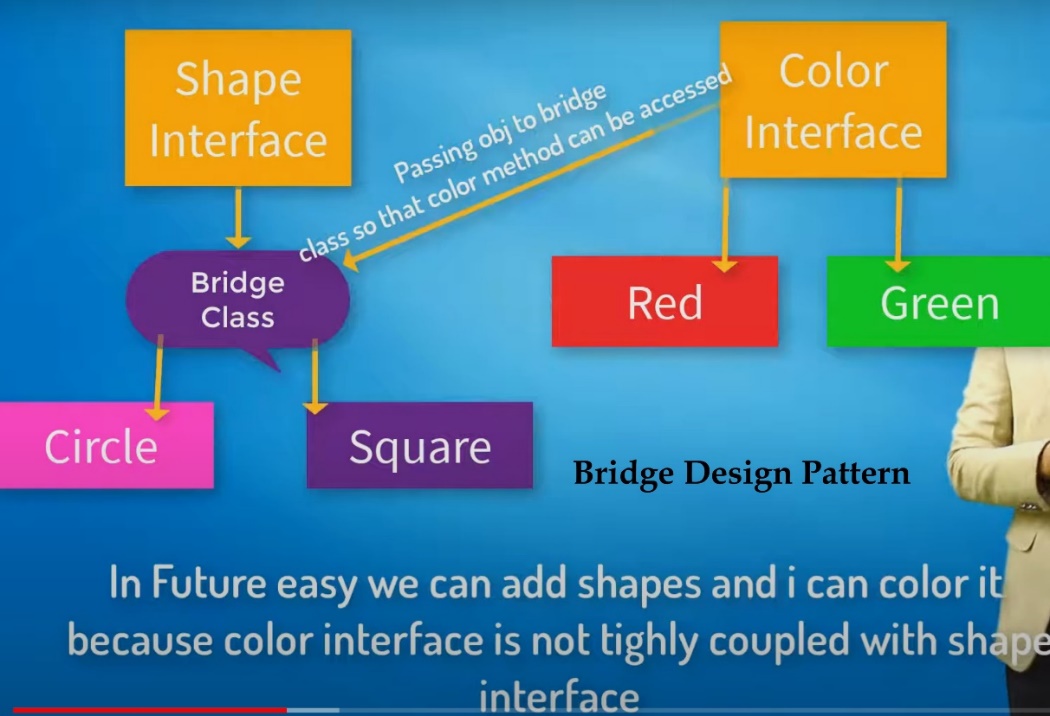
Functional Design

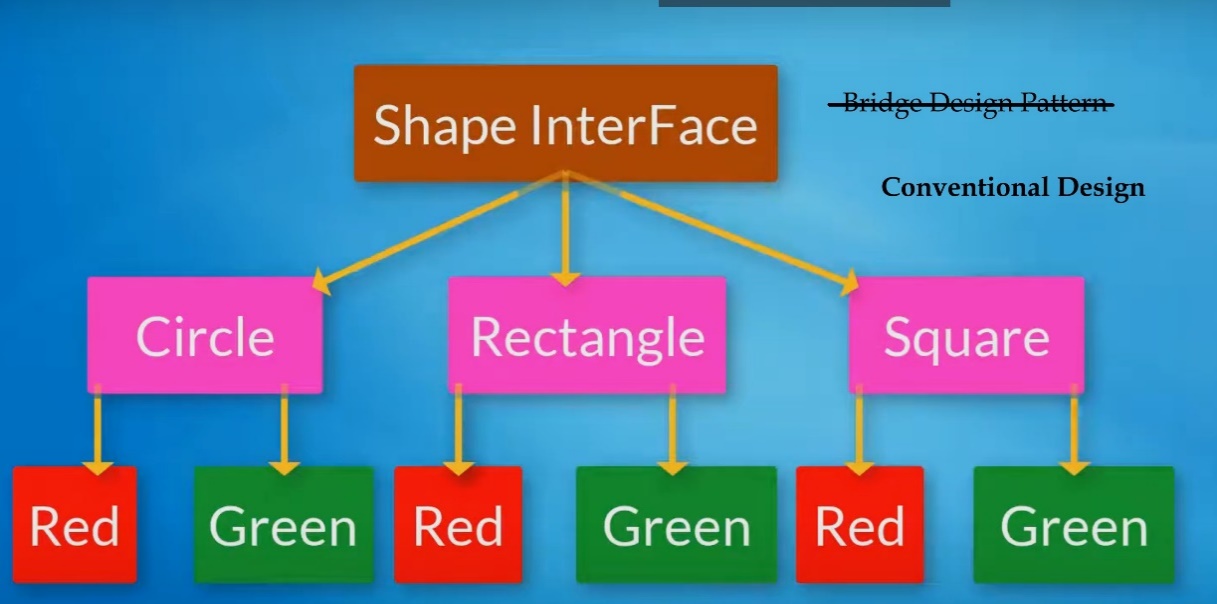
* Both can evolve in future without affecting each other.
* The Bridge Pattern increases the loose coupling between class abstraction and its implementation.
* The abstraction contains a reference to the implementer. Children of the abstraction are referred to as refined abstractions, and children of the implementer are concrete implementers.

Why / When

* Compared with convential design, Bridge pattern uses less classes, smaller excecutable, is more readable, less coupled, and no static bindings.
* When you prefer composition over inheritance”. It becomes handy when you must subclass different times in ways that are orthogonal with one another.
* Use bridge pattern to map orthogonal class hierarchies

## Diagrams





## Code Example

class fillColorImp **// Color Interface**

{

public:

virtual void fillColor() = 0;

};

class greenColor : public fillColorImp

{

public:

void fillColor()

{

cout << "Its is Green color" << endl;

}

};

class redColor : public fillColorImp

{

public:

void fillColor()

{

cout << "Its is red color" << endl;

}

};

**/////////////////////**

**// Shape Interface**

class Shape {

public:

virtual void colorIt() = 0;

virtual void drawIt() = 0;

};

**// bridge is a child of shape**

class bridge : public Shape

{

public:

bridge(fillColorImp\* obj)

{

cout << "\ninside constructor of bridge class\n" << endl;

colorObj1 = obj;

}

// virtual void colorIt() = 0; these are available to bridge

// virtual void drawIt() = 0;

protected:

fillColorImp\* colorObj1;

};

**// square is a child of bridge (& a grandchild of Shape via bridge parent)**

class square : public bridge

{

public:

// Passes in fillColor interface \*ptr and calls the bridge constructor.

// Whatever object is passed to square constructor (eg. fillColorImp\* obj)

// is passed to bridge constructor.

// bridge contructor : **square 'is-a' bridge**, **NOT 'has-a'**

square(fillColorImp\* obj) : bridge(obj) {}

void colorIt()

{

colorObj1->fillColor();

}

void drawIt()

{

cout << "Squar drawan without color" << endl;

}

};

// shape : rectangle : bridge

class Rectangle : public bridge

{

public:

// fillColorImp\* colorObj1; available to this class because's its a bridge.

Rectangle(fillColorImp\* obj) :bridge(obj) {}

void colorIt()

{

colorObj1->fillColor();

}

void drawIt()

{

cout << "Squar drawan without color" << endl;

}

};

int main() {

Shape\* shapeobj1;

fillColorImp\* ObjRed = new redColor;

fillColorImp\* ObjGreen = new greenColor;

//UseCase1

cout << "\nUsecase1" << endl;

cout << "------------------------------------" << endl;

shapeobj1 = new square(ObjRed);

shapeobj1->colorIt();

shapeobj1->drawIt();

// changes square obj to contain a green color object via the bridge

shapeobj1 = new square(ObjGreen);

shapeobj1->colorIt();

shapeobj1->drawIt();

//UseCase2

cout << "\n\nUsecase2 " << endl;

cout << "------------------------------------" << endl;

shapeobj1 = new Rectangle(ObjRed);

shapeobj1->colorIt();

shapeobj1->drawIt();

std::cout << "\n";

system("pause");

}

# **Builder**

## Overview

Bridge Pattern

* Creational Design Pattern used to construct a complex object step by step.
* The pattern separates the construction of a complex object from its representation.

Functional Design

* 5 components: Product, Builder, Concrete Builder, Director, & Client.
* **Product** is typically a class with attributes representing the different parts that the Builder constructs
* The **Builder** is an interface or an abstract class that declares the construction steps for building a complex object.
* **ConcreteBuilder** classes implement the Builder interface, providing specific implementations for building each part of the product.
* **Director** object tells all the builders how they will build the object. The construction of Product.
* **Client** is the code that initiates the construction of the complex object.

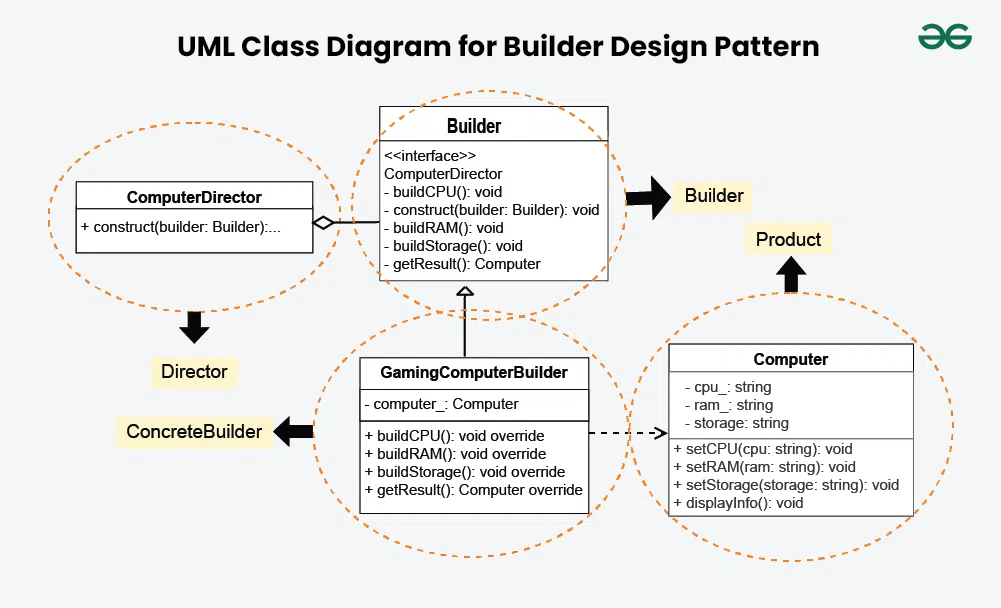
When / Why

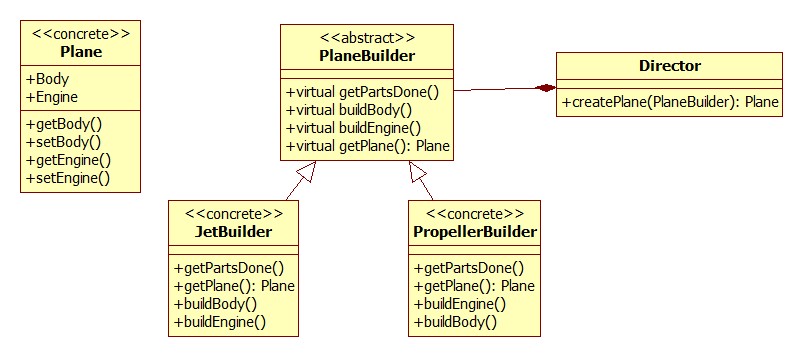
* Create complex objects with a large number of optional components or configuration parameters.
* Examples: Complex Object Construction, Step-by-Step construction, multi parameter constructors avoidance, immutable objects, configurable objects, common interface for multiple representations.

Why Not?

* Only simple objects, performance, immutable objects with final fields, complexity, coupling

## Diagrams





## Code Example

// Builder.cpp | [Builder Design Pattern YouTube video](https://www.youtube.com/watch?v=D1CnNAszv_M&list=PLk6CEY9XxSIDZhQURp6d8Sgp-A0yKKDKV&index=7)

// Your end product - after builder finishes, it returns the whole plane.

class Plane {

string \_plane; // plane type name, not parent object or separate object.

string \_body;

string \_engine;

public:

Plane(string planeType) :\_plane{ planeType } {}

void setEngine(string type) { \_engine = type; }

void setBody(string body) { \_body = body; }

string getEngine() { return \_engine; }

string getBody() { return \_body; }

void show() {

cout << "Plane Type: " << \_plane << endl

<< "Body Type: " << \_body << endl

<< "Engine Type: " << \_engine << endl << endl;

}

};

// PlaneBuilder Abstract Class

// Means all builders should have atleast these methods

class PlaneBuilder {

public:

virtual void getPartsDone() = 0;

virtual void buildBody() = 0;

virtual void buildEngine() = 0;

//virtual ~PlaneBuilder(){}

Plane\* getPlane() { return \_plane; }

protected:

Plane\* \_plane;

};

// PlaneBuilder concrete class

// knows only how to build Propeller Plane

class PropellerBuilder : public PlaneBuilder {

public:

void getPartsDone() { \_plane = new Plane("Propeller Plane"); }

void buildEngine() { \_plane->setEngine("Propeller Engine"); }

void buildBody() { \_plane->setBody("Propeller Body"); }

//~PropellerBuilder(){delete \_plane;}

};

// PlaneBuilder concrete class

// Knows only how to build Jet Plane

class JetBuilder : public PlaneBuilder {

public:

void getPartsDone() { \_plane = new Plane("Jet Plane"); }

void buildEngine() { \_plane->setEngine("Jet Engine"); }

void buildBody() { \_plane->setBody("Jet Body"); }

//~JetBuilder(){delete \_plane;}

};

// Defines steps and tells to the builder that build in given order.

class Director{

public:

// different builders can be passed in that create different planes

Plane\* createPlane(PlaneBuilder\* builder) { // plane needs to be constructed in this specific order.

builder->getPartsDone(); // Step 1

builder->buildBody(); // Step 2

builder->buildEngine(); // Step 3

return builder->getPlane(); // Step 4

}

private:

PlaneBuilder\* builder;

};

int main() {

Director dir;

JetBuilder jb;

PropellerBuilder pb;

Plane\* jet = dir.createPlane(&jb);

Plane\* pro = dir.createPlane(&pb);

jet->show();

pro->show();

delete jet;

delete pro;

system("pause");

return 0;

}

# **Chain of Responsibility**

## Overview

Chain of Responsibility

* Behavorial Design pattern where a manager object pass requests along a chain of handlers.
* Each handler decides whether to process the request or pass it along to next handler.

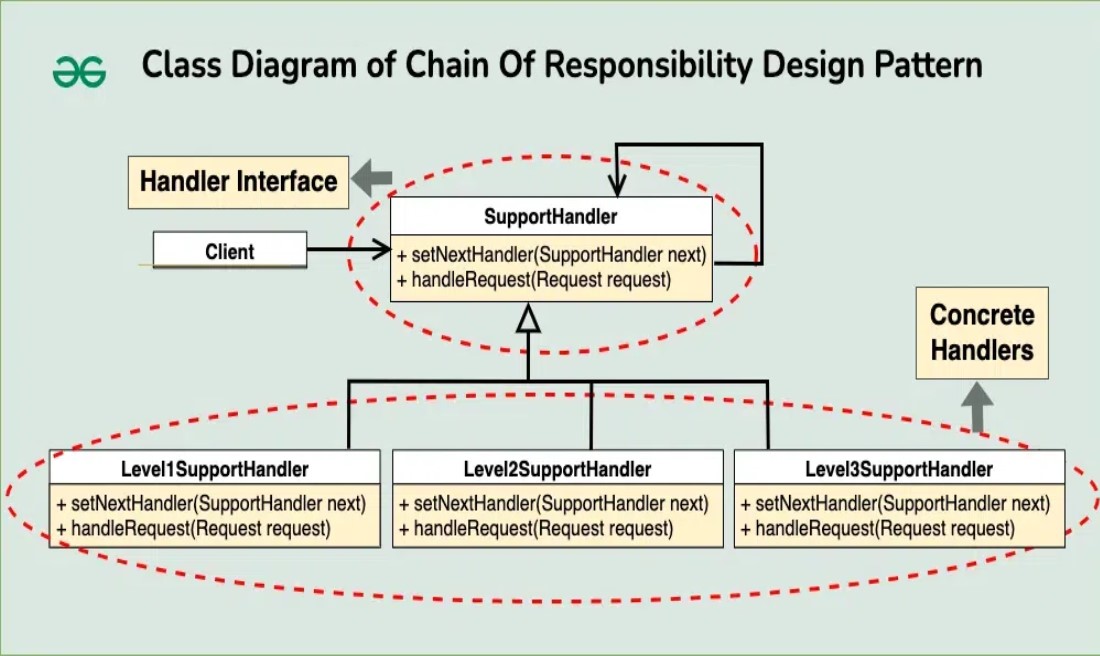
Functional Design

* Build commands or requests and hand them off appropriately.
* Send a request to other objects without knowing who is going to handle it.
* Handler Interface, Concrete Handlers, Client

When / Why

* This pattern encourages loose coupling between sender and receiver.
* Provides freedom in handling the request. Dynamic Order of Handling.
* Simplified Object interactions
* Enhanced Maintainability

## Diagrams



## Code Example

// Chain.h | Chain of Command Pattern

// Handler interface will work with event structs.

// In a chain of command, you got to be something with a request (hence handler interface and struct event).

// Build a concrete set of handlers that implements the IHandler interface, including a base parent handler.

#pragma once

#include <string>

#include <iostream>

struct Event {

int id = -1;

std::string message = "";

};

// Interface Handler to handle events

// base option for a way to add to the chain of responsibility

class IHandler {

public:

virtual ~IHandler() {}

virtual IHandler\* SetNext(IHandler\* handler) = 0;

virtual Event Handle(Event event) = 0;

};

// Concrete class that acts as the main parent for the handlers.

class BaseHandler : public IHandler {

public:

BaseHandler() : next\_handler(nullptr) {}

IHandler\* SetNext(IHandler\* handler) override

{

next\_handler = handler;

// setting next handler and returning handler allows us to chain handlers.

return handler;

}

// event == request, often those two variable names are interchangeable.

// base case, if no specialized event handled, this will get called.

Event Handle(Event event) override

{

if (next\_handler) {

return next\_handler->Handle(event);

} // the next\_handler is a member variable that will change each time it

goes through the chain of responsibility.

// if it falls through, it will instantiate a new event.

return event;

}

private:

IHandler\* next\_handler; // this is unique for each child class if setNext is run for each handler you want to chain.

};

class HandleEvent1 : public BaseHandler {

public:

Event Handle(Event event) override

{

if (event.id == 1)

{

std::cout << "-> handling event with id 1\n";

std::cout << " event message: " << event.message << std::endl;

return event;

}

return BaseHandler::Handle(event); // falls through to base class event handle call, which returns an event.

}

};

class HandleEvent2 : public BaseHandler {

public:

Event Handle(Event event) override {

if (event.id == 2) {

std::cout << "-> handling event with id 2\n";

std::cout << " event message: " << event.message << std::endl;

return event;

}

return BaseHandler::Handle(event);

}

};

class HandleEvent3 : public BaseHandler {

public:

Event Handle(Event event) override {

if (event.id == 3) {

std::cout << "-> handling event with id 3\n";

std::cout << " event message: " << event.message << std::endl;

return event;

}

return BaseHandler::Handle(event);

}

};

// Chain\_Of\_Responsibility.cpp | https://www.youtube.com/watch?v=yXJHXDy5oig

#include "chain.h"

int main() {

HandleEvent1\* handleevent1 = new HandleEvent1;

HandleEvent2\* handleevent2 = new HandleEvent2;

HandleEvent3\* handleevent3 = new HandleEvent3;

// this first part returns handlerevent2, so you can go right into ->SetNext(handlerevent3)

handleevent1->SetNext(handleevent2)->SetNext(handleevent3); // this sets the chain of responsibility.

// handleevent1->SetNext(handleevent2); // or you could do it on separate lines.

// handleevent2->SetNext(handleevent3);

Event an\_event;

an\_event.id = 3;

an\_event.message = "ID 3 being handle by HandleEvent1 handler class! \n\n";

handleevent1->Handle(an\_event);

/\*{

std::cout << "TEST1\n";

Event an\_event;

an\_event.id = 1;

an\_event.message = "why hello there from TEST1";

Event returned\_event = handleevent1->Handle(an\_event);

std::cout << "returned event id: " << returned\_event.id << std::endl;

}

{

std::cout << "TEST2\n";

Event an\_event;

an\_event.id = 2;

an\_event.message = "why hello there from TEST2";

Event returned\_event = handleevent1->Handle(an\_event);

std::cout << "returned event id: " << returned\_event.id << std::endl;

}

{

std::cout << "TEST3\n";

Event an\_event;

an\_event.id = 3;

an\_event.message = "why hello there from TEST3";

Event returned\_event = handleevent1->Handle(an\_event);

std::cout << "returned event id: " << returned\_event.id << std::endl;

}

{

std::cout << "TEST4\n";

Event an\_event;

an\_event.id = 4;

an\_event.message = "why hello there from TEST4";

Event returned\_event = handleevent1->Handle(an\_event);

std::cout << "returned event id: " << returned\_event.id << std::endl;

}\*/

system("pause");

}

# **Command**

## Overview

Command Pattern

* Command Pattern is a behavioral design pattern that turns a request into a stand-alone object that contains all information about the request.
* Parameterization of clients of different requests, queuing requests, and undoable operations.

Functional Design

* Encapsulates a request as an object, allowing for the separation of sender and receiver.
* The pattern supports undoable(action or a series of actions that can be reversed or undone in a system) operations by storing the state or reverse commands.

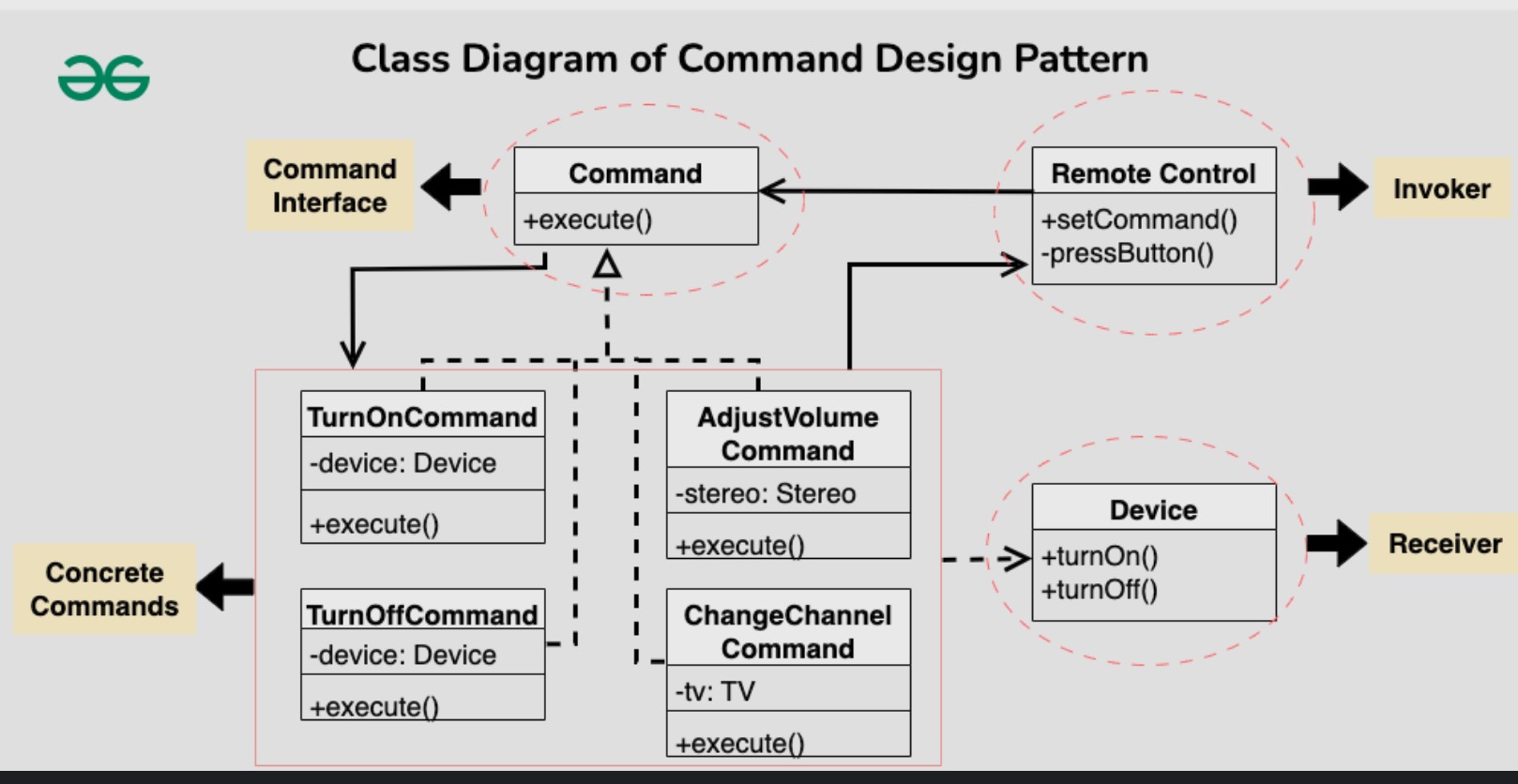
Components

* **Command Interface** – rulebook that all command classes follow.
* **Concrete Command** Classes – specific commands.
* **Invoker** – remote control responsible for initiating command execution. Holds a ref to cmd.
* **Receiver** – device that knows how to perform the actual operation associated with a command

Why / Why

* Decoupling the sender of request from object performing request.
* Undo / Redo functionality needed.
* Queues and Logging.

## Diagrams



## Code Example

// Command.h | Base definitions of what a command is.

#include <string>

#include <iostream>

// COMMAND INTERFACE:Build commands and use polymorphism to process variety of commands

class ICommand {

public:

virtual ~ICommand() {}

virtual void Execute() const = 0;};

// example of a simple command

class LogCommand : public ICommand {

public:

// don't allow to construct in any other way, need to pass in command

LogCommand(std::string data) : pay\_load\_(data) {}

// Simple because no separate receiver class

void Execute() const override {

std::cout << "printstring: " << pay\_load\_ << '\n';

}

// create a logger command,

// different function to choose where to log to // TODO:

private:

std::string pay\_load\_;

public:

LogCommand() = delete;

};

// A class in your code that handles your main logic.

// Delegation Pattern crossover, command comes in, then command gets delegated to Receiver

// Reciever interface, have multiple receivers, and change receivers on the fly.

class Reciever {

public:

void ProcessA(const std::string& proc\_a\_stuff) {

std::cout << "Reciever(ProcessA) is working on: " << proc\_a\_stuff << '\n';

}

void ProcessB(const std::string& proc\_b\_stuff) {

std::cout << "Reciever(ProcessB) is working on: " << proc\_b\_stuff << '\n';

}

//... todo: more business logic

};

class ComplexCommand : public ICommand {

public:

ComplexCommand(Reciever\* receiver, std::string a, std::string b)

: reciever\_(receiver), proc\_a\_details(a), proc\_b\_details(b)

{}

// 'this->' prevents confusion with multiple inheritance.

void Execute() const override {

this->reciever\_->ProcessA(proc\_a\_details); // keep logic on the commands as simple as possible.

this->reciever\_->ProcessB(proc\_b\_details); // all logic should be checked and handled by receiver.

}

private:

// If it were it's own object (instead of a ptr), Reciever receiver, would be instantiated here,

// we'd have a copy of the business logic inside every command.

Reciever\* reciever\_; // not owned- probably should be refactored to a shared ptr

std::string proc\_a\_details;

std::string proc\_b\_details;

};

class Invoker {

public:

// Invoker typically handles your main loop.

// Passes commands off to Receivers.

~Invoker() {

// delete on\_startup\_command;

// delete on\_mid\_command;

// delete on\_shutdown;

}

void SetStartupCommand(ICommand\* command) {

on\_startup\_command = command; // better method to use pointers than copying them like this. needs optimized.

}

void SetMidCommand(ICommand\* command) {

on\_mid\_command = command;

}

void SetShutdownCommand(ICommand\* command) {

on\_shutdown = command;

}

void DoTheWork() {

if (on\_startup\_command) {

on\_startup\_command->Execute();

}

if (on\_mid\_command) {

on\_mid\_command->Execute();

}

if (on\_shutdown) {

on\_shutdown->Execute();

}

}

private:

ICommand\* on\_startup\_command = nullptr;

ICommand\* on\_mid\_command = nullptr;

ICommand\* on\_shutdown = nullptr;

};

// Command.cpp | [Command Pattern YouTube video](https://www.youtube.com/watch?v=yDkIK3JfHkw&list=PLalVdRk2RC6otl3oBU2cn-P6DWi1y1PS3&index=14)

#include "command.h"

int main() {

// simple command demo

LogCommand lc("this is my message to log");

lc.Execute();

//complex command demo

Reciever rec;

ComplexCommand cc(&rec, "string a", "string b");

cc.Execute();

// ^^ Above are simple examples of how commands work directly in main individually, but not used in practice.

// Typically, you have an Invoker which wraps all your core business logic.

// Invoker handles a bunch of different commands. Allows you to batch together commands.

// invoking sets of commands demo

Invoker invoker;

//invoker.SetStartupCommand(new LogCommand("starting up"));

//invoker.SetMidCommand(&cc);

//invoker.SetShutdownCommand(new LogCommand("shutting down"));

//invoker.DoTheWork();

system("pause");

}

// Difference between an Invoker and a Receiver?

# **Composite**

## Overview

Composite Pattern

* Composite is a structural design pattern that allows composing objects into a tree-like structure and work with the it as if it was a singular object.
* Treat the individual object in the same way as the collection of those individual objects

Functional Design

* The Composite class represents the complex components that may have children.
* The Composite objects delegate the actual work to their children.

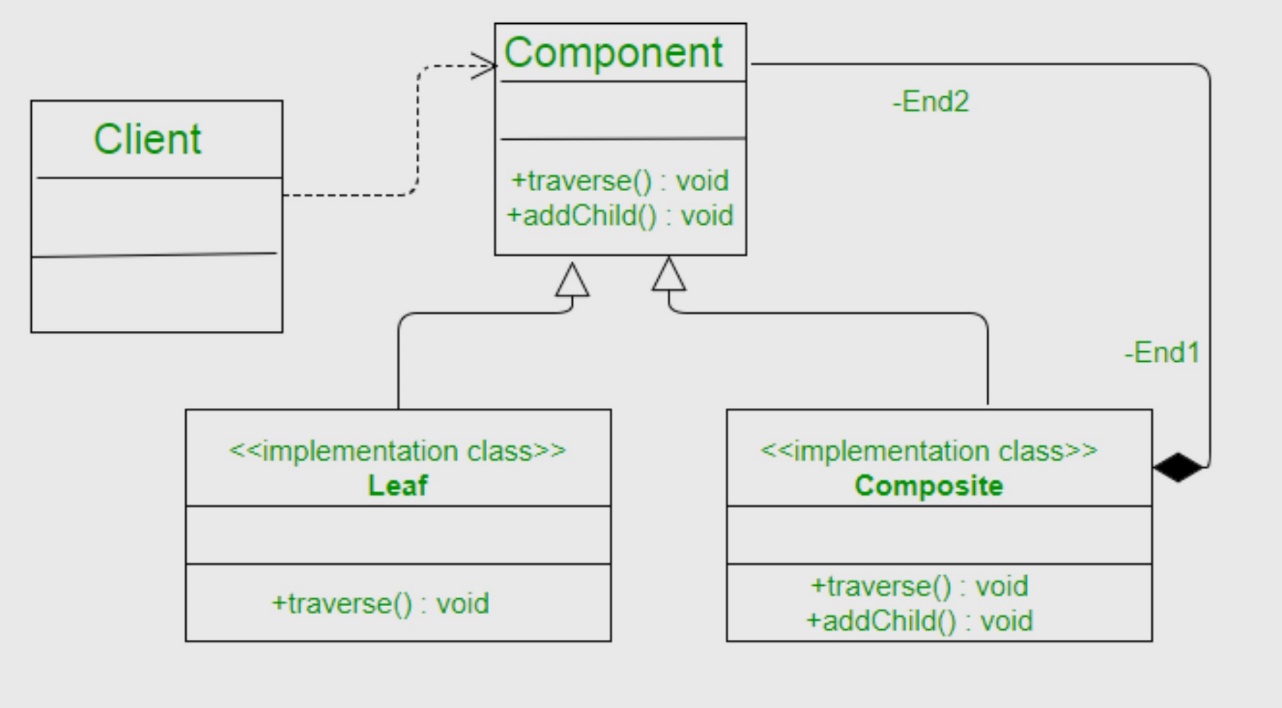
Components

* **Component** – interface for objects in composition, manages child components
* **Leaf** – primitive object behavior defined by the component interface
* **Composite** – class with child components, which are leaf or other composites
* **Client** – manipulates objects in the composition through component interface.

Why / Why

* Uniform Interface, Hierarchial Structures, Flexible, Scalable, Common Operations,
* Use when clients need to ignore difference between composition of object and individual objs.

## Diagrams



Dksks

## Code Example

// Composite Pattern | https://refactoring.guru/design-patterns/composite/cpp/example

#include <algorithm>

#include <iostream>

#include <list>

#include <string>

// The base Component class declares common operations for both simple and

// complex objects of a composition.

class Component {

protected:

Component\* parent\_;

\* Optionally, the base Component can declare an interface for setting and

\* accessing a parent of the component in a tree structure. It can also

\* provide some default implementation for these methods.

public:

virtual ~Component() {}

void SetParent(Component\* parent) {

this->parent\_ = parent;

}

Component\* GetParent() const {

return this->parent\_;

}

\* In some cases, it would be beneficial to define the child-management

\* operations right in the base Component class. This way, you won't need to

\* expose any concrete component classes to the client code, even during the

\* object tree assembly. The downside is that these methods will be empty for

\* the leaf-level components.

virtual void Add(Component\* component) {}

virtual void Remove(Component\* component) {}

\* You can provide a method that lets the client code figure out whether a

\* component can bear children.

virtual bool IsComposite() const {

return false;

}

\* The base Component may implement some default behavior or leave it to

\* concrete classes (by declaring the method containing the behavior as

\* "abstract").

virtual std::string Operation() const = 0;

};

// The Leaf class represents the end objects of a composition.

// A leaf can't have any children.

// Usually, it's the Leaf objects that do the actual work,

// Composite objects only delegate to their sub-components.

class Leaf : public Component {

public:

std::string Operation() const override {

return "Leaf";

}

};

// The Composite class represents the complex components that may have children.

// Usually, the Composite objects delegate the actual work to their children and

// then "sum-up" the result.

class Composite : public Component {

protected:

std::list<Component\*> children\_;

public:

\* A composite object can add or remove other components (both simple or

\* complex) to or from its child list.

void Add(Component\* component) override {

this->children\_.push\_back(component);

component->SetParent(this);

}

\* Have in mind that this method removes the pointer to the list but doesn't

\* frees the memory, you should do it manually or better use smart pointers.

void Remove(Component\* component) override {

children\_.remove(component);

component->SetParent(nullptr);

}

bool IsComposite() const override {

return true;

}

\* The Composite executes its primary logic in a particular way. It traverses

\* recursively through all its children, collecting and summing their results.

\* Since the composite's children pass these calls to their children and so

\* forth, the whole object tree is traversed as a result.

std::string Operation() const override {

std::string result;

for (const Component\* c : children\_) {

if (c == children\_.back()) {

result += c->Operation();

}

else {

result += c->Operation() + "+";

}

}

return "Branch(" + result + ")";

}

};

// The client code works with all of the components via the base interface.

void ClientCode(Component\* component) {

// ...

std::cout << "RESULT: " << component->Operation();

// ...

}

// Thanks to the fact that the child-management operations are declared in the

// base Component class, the client code can work with any component, simple or

// complex, without depending on their concrete classes.

void ClientCode2(Component\* component1, Component\* component2) {

// ...

if (component1->IsComposite()) {

component1->Add(component2);

}

std::cout << "RESULT: " << component1->Operation();

// ...

}

// This way the client code can support the simple leaf components...

int main() {

Component\* simple = new Leaf;

std::cout << "Client: I've got a simple component:\n";

ClientCode(simple);

std::cout << "\n\n";

// complex composites

Component\* tree = new Composite;

Component\* branch1 = new Composite;

Component\* leaf\_1 = new Leaf;

Component\* leaf\_2 = new Leaf;

Component\* leaf\_3 = new Leaf;

branch1->Add(leaf\_1);

branch1->Add(leaf\_2);

Component\* branch2 = new Composite;

branch2->Add(leaf\_3);

tree->Add(branch1);

tree->Add(branch2);

std::cout << "Client: Now I've got a composite tree:\n";

ClientCode(tree);

std::cout << "\n\n";

std::cout << "Client: I don't need to check the components classes.”

ClientCode2(tree, simple);

std::cout << "\n";

return 0;

}

# **CRTP**

## Overview

**CRTP** (Curiously Recurring Template Pattern)

* Avoids usage of Virtual Ptrs and VTable with CRTP by declaring a class that derives from a class template instantiation using the class itself as a template argument. F-bound polymorphism.

**Virtual method vs CRTP benchmark**

Performance gain from CRTP is because the use of a VTable dispatch has been circumvented.

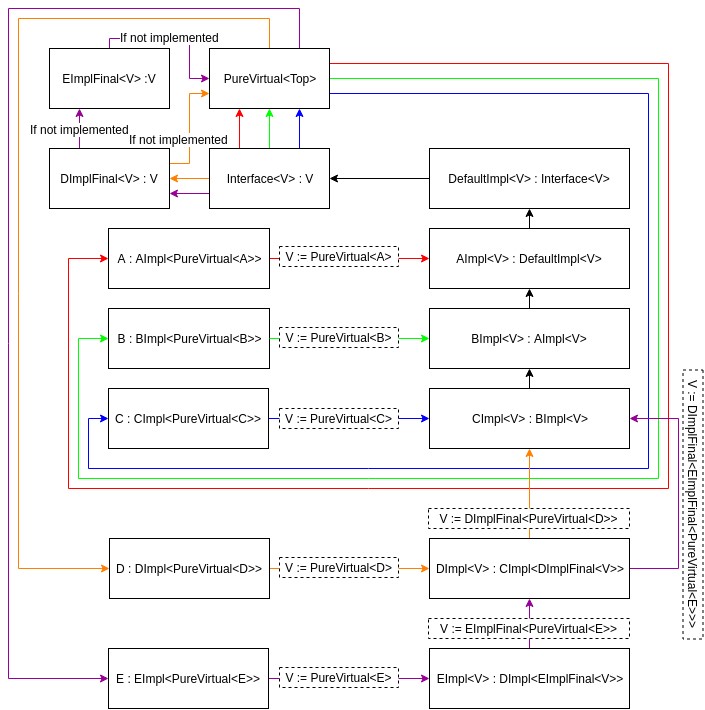
Functional Design

* Allows class hierarchies to be extended by substituting different base classes.
* **Static Polymorphism** : ypically, the base class template will take advantage of the fact that member function bodies (definitions) are not instantiated until long after their declarations, and will use members of the derived class within its own member functions, via the use of a [cast](https://en.wikipedia.org/wiki/Type_conversion).

Why / Why

Avoiding the cost of virtual function calls while retaining some of the hierarchical benefits.

## Diagrams



## Code Example

Virtual Pointer Manipulation Example:

// A C++ program to demonstrate that we can directly

// manipulate VPtr. Note that this program is based

// on the assumption that compiler store vPtr in a

// specific way to achieve run-time polymorphism.

#include <iostream>

**using** **namespace** std;

#pragma pack(1)

// A base class with virtual function foo()

**class** CBase {

**public**:

**virtual** **void** foo() noexcept

    {

        cout << "CBase::Foo() called" << endl;

    }

**protected**:

**int** mData;

};

// A derived class with its own implementation

// of foo()

**class** CDerived : **public** CBase {

**public**:

**void** foo() noexcept

    {

        cout << "CDerived::Foo() called" << endl;

    }

**private**:

**char** cChar;

};

// Driver code

**int** main()

{

    // A base type pointer pointing to derived

    CBase\* pBase = **new** CDerived;

    // Accessing vPtr

**int**\* pVPtr = \*(**int**\*\*)pBase;

    // Calling virtual method

    ((**void** (\*)())pVPtr[0])();

    // Changing vPtr

**delete** pBase;

    pBase = **new** CBase;

    pVPtr = \*(**int**\*\*)pBase;

    // Calls method for new base object

    ((**void** (\*)())pVPtr[0])();

**return** 0;

}

**CRTP Demo:**

// Image program (similar to above) to demonstrate

// working of CRTP

#include <chrono>

#include <iostream>

**using** **namespace** std;

**typedef** std::chrono::high\_resolution\_clock Clock;

// To store dimensions of an image

**class** Dimension {

**public**:

    Dimension(**int** \_X, **int** \_Y)

    {

        mX = \_X;

        mY = \_Y;

    }

**private**:

**int** mX, mY;

};

// Base class for all image types. The template

// parameter T is used to know type of derived

// class pointed by pointer.

**template** <**class** T>

**class** Image {

**public**:

**void** Draw()

    {

        // Dispatch call to exact type

**static\_cast**<T\*>(**this**)->Draw();

    }

    Dimension GetDimensionInPixels()

    {

        // Dispatch call to exact type

**static\_cast**<T\*>(**this**)->GetDimensionInPixels();

    }

**protected**:

**int** dimensionX, dimensionY;

};

// For Tiff Images

**class** TiffImage : **public** Image<TiffImage> {

**public**:

**void** Draw()

    {

        // Uncomment this to check method dispatch

        // cout << "TiffImage::Draw() called" << endl;

    }

    Dimension GetDimensionInPixels()

    {

**return** Dimension(dimensionX, dimensionY);

    }

};

// There can be more derived classes like PngImage,

// BitmapImage, etc

// Driver code

**int** main()

{

    // An Image type pointer pointing to Tiffimage

    Image<TiffImage>\* pImage = **new** TiffImage;

    // Store time before virtual function calls

**auto** then = Clock::now();

    // Call Draw 1000 times to make sure performance

    // is visible

**for** (**int** i = 0; i < 1000; ++i)

        pImage->Draw();

    // Store time after virtual function calls

**auto** now = Clock::now();

    cout << "Time taken: "

         << std::chrono::duration\_cast<std::chrono::nanoseconds>(now - then).count()

         << " nanoseconds" << endl;

**return** 0;

}

lksdks

# **Dependency Injection**

## Overview

Dependency Injection Pattern

* The usefulness of Dependency Injection is in that the implementation of a class is more flexible and independent by decreasing the amount of coupling between a client and another service.
* The exact decoupling is the removal of a client class’s dependency on the implementation of an interface, instead requiring the implementation to be passed in.Sldk

Functional Design

If you want to use this technique, you need classes that fulfill four basic roles. These are:

1. The **service** you want to use.
2. The **client** that uses the service.
3. An **interface** that’s used by the client and implemented by the service.
4. The **injector** which creates a service instance and injects it into the client.

Dependencies

* There are three ways a component (object or function) can get a hold of its dependencies:

1. The component can create the dependency, typically using the new operator.
2. The component can look up the dependency, by referring to a global variable.
3. The component can have the dependency passed to it where it is needed.

* The third option is the most viable, since it removes the responsibility of locating the dependency from the component. The dependency is simply handed to the component.
* Making a class's dependencies explicit and requiring that they be injected into it is a good way of making a class more reusable, testable and decoupled from others.

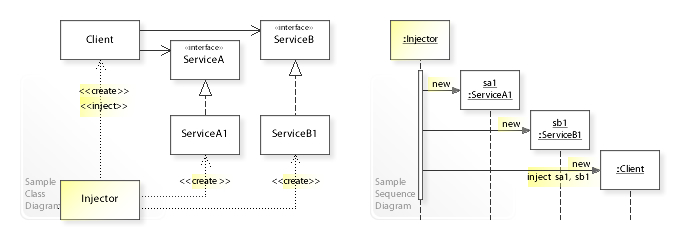
Types of Dependency Injection

* Setter - Adding a setter method that accepts the dependency, works for optional dependencies
* Constructor – add an argument to the constuctor signature to accept dependency (common)
* Property – setting the public fields fo the class directly.

Why / Why

* Dependency injection is a design pattern that allows a class to receive its dependencies from an external source, rather than creating them internally. This makes the class more flexible and easier to test, as its dependencies can be easily swapped out for mock objects during testing.

## Diagrams



## Code Example

**#include <iostream>**

**class** **GasolineSource** {

public:

**virtual** **void** FuelUp() **=** 0;

**virtual** **~**GasolineSource() **=** **default**;

};

**class** **GasStation** **:** **public** GasolineSource {

public:

**virtual** **void** FuelUp() {

std**::**cout **<<** "Pumping gas at gas station" **<<** std**::**endl;

}

};

**class** **FuelCan** **:** **public** GasolineSource {

public:

**virtual** **void** FuelUp() {

std**::**cout **<<** "Pumping gas from fuel can" **<<** std**::**endl;

}

};

**class** **Car** {

GasolineSource **\***gasolineService **=** nullptr;

public:

*// The dependency for a source of gasoline is passed in*

*// through constructor injection*

*// as opposed to hard-coded into the class definition.*

Car(GasolineSource **\***service)

**:** gasolineService(service) {

*// If the dependency was not defined, throw an exception.*

**if**(gasolineService **==** nullptr){

**throw** std**::**invalid\_argument("service must not be null");

}

}

**void** getGasoline() {

std**::**cout **<<** "Car needs more gasoline!" **<<** std**::**endl;

*// Abstract away the dependency implementation with polymorphism.*

gasolineService**->**FuelUp();

}

};

**#include <iostream>**

**class** **GasolineSource** {

public:

**virtual** **void** FuelUp() **=** 0;

**virtual** **~**GasolineSource() **=** **default**;

};

**class** **GasStation** **:** **public** GasolineSource {

public:

**virtual** **void** FuelUp() {

std**::**cout **<<** "Pumping gas at gas station" **<<** std**::**endl;

}

};

**class** **FuelCan** **:** **public** GasolineSource {

public:

**virtual** **void** FuelUp() {

std**::**cout **<<** "Pumping gas from fuel can" **<<** std**::**endl;

}

};

**class** **Car** {

GasolineSource **\***gasolineService **=** nullptr;

public:

*// The dependency for a source of gasoline is passed in*

*// through constructor injection*

*// as opposed to hard-coded into the class definition.*

Car(GasolineSource **\***service)

**:** gasolineService(service) {

*// If the dependency was not defined, throw an exception.*

**if**(gasolineService **==** nullptr){

**throw** std**::**invalid\_argument("service must not be null");

}

}

**void** getGasoline() {

std**::**cout **<<** "Car needs more gasoline!" **<<** std**::**endl;

*// Abstract away the dependency implementation with polymorphism.*

gasolineService**->**FuelUp();

}

};

# **Decorator**

## Overview

Decorator Pattern

he decorator pattern is an alternative to subclassing. Subclassing adds behavior at compile time, and the change affects all instances of the original class.

Functional Design

* Component Interface – common interface for both concrete components and decorators.
* Concrete Component - basic objects or classes that implement the Component interface. They are the objects to which we want to add new behavior or responsibilities.
* Decorator - This is an abstract class that also implements the Component interface and has a reference to a Component object. Decorators are responsible for adding new behaviors to the wrapped Component object.
* Concrete Decorator - Extends the Decorator, adding specific behaviors to the Component.

Use Cases

* **Adding Features to GUI Widgets:**We can use the Decorator Pattern to add features like borders, scrollbars, or tooltips to GUI components.
* **Text Formatting:** Decorators can be applied to text elements to add formatting such as fonts, colors, or styles.
* **Input/ Output Streams:** In C++, decorators can be used to add functionality like buffering, compression, or encryption to input/output streams.
* **Coffee or Food Customization**

Why / When

* Open-Close Principle, Flexibility, Reusable Code, Composition over inheritance, dynamic behavior modification, and clear code structure.
* Adding behavior dynamically.

Why Not

* Complexity
* Leads to large number of classes
* Order of decoration can affect final behavior

Additional Notes from Example

// ### DECORATOR ###

// Adds functionality to classes that inherit from the same interface

// Generally contains a ptr to the base interface and through polymorphism

// is able to add this functionality to that brother/sister class.

// The Decorator itself becomes the final version of the class that is instantiated

// Decorator pattern allows behavior to be added to an individual object, dynamically,without affecting behavior of other object from the same class.

// Decorators support compositional construction versus top-down

// Decorators can support or augment the 'Facade' pattern.

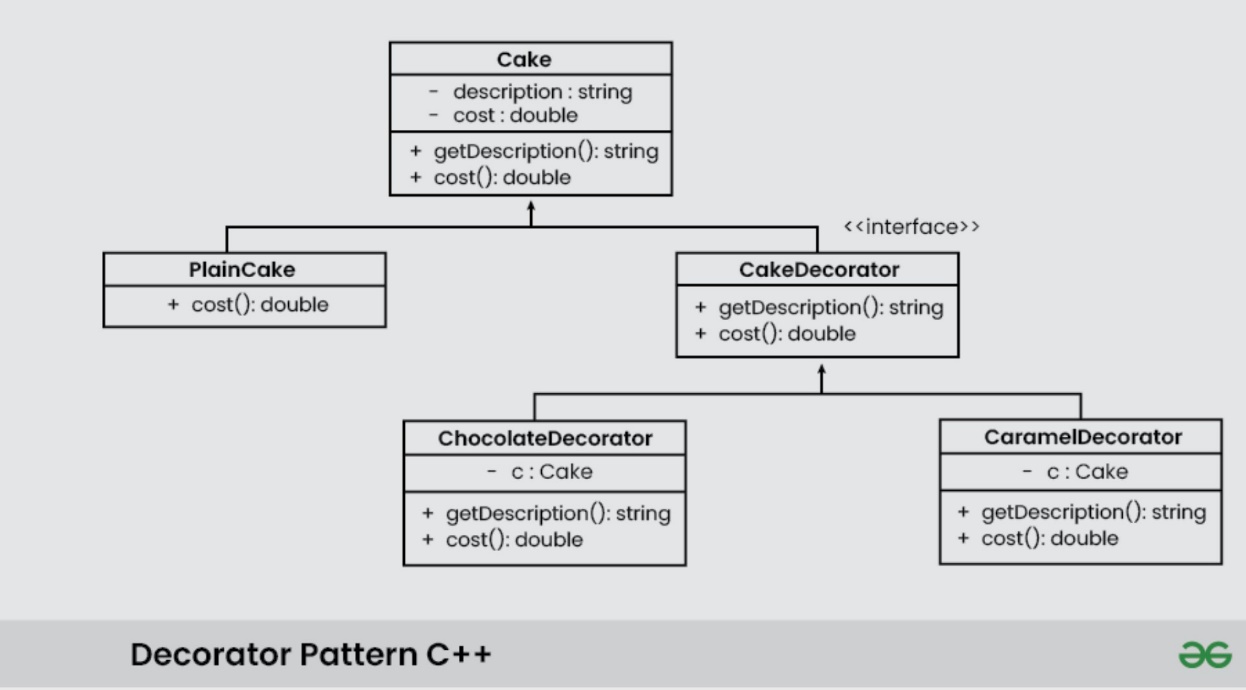
// Dynamically adding and removing responsibilities.

// 'Facade' and 'Adapter' re-does the interface in a more simplified way (typically used with bringing in libraries).

// When and Where?

// Use a decorator pattern when you start creating large numbers of inherited classes. Decorator allows you to create a few base classes and then have extensions/decorations to modify sub-classes. Extensions are dynamic that can be added or removed on the fly.

## Diagrams



## Code Example

struct WebPage

{

virtual void display() = 0;

virtual ~WebPage() = default;

};

struct BasicWebPage : WebPage

{

std::string html;

void display() override

{

std::cout << "Basic WEB page html: " << html << std::endl;

}

};

struct WebPageDecorator : WebPage

{

WebPageDecorator(std::unique\_ptr<WebPage> webPage) : \_webPage(std::move(webPage)) {}

void display() override

{

\_webPage->display();

}

private:

std::unique\_ptr<WebPage> \_webPage; // \*\_webPage

};

struct AuthenticatedWebPage : WebPageDecorator

{

AuthenticatedWebPage(std::unique\_ptr<WebPage> webPage) : WebPageDecorator(std::move(webPage))

{}

void authenticateUser()

{

std::cout << "authentification done" << std::endl;

}

void display() override

{

authenticateUser();

WebPageDecorator::display();

}

};

struct AuthorizedWebPage : WebPageDecorator

{

AuthorizedWebPage(std::unique\_ptr<WebPage> webPage) :

WebPageDecorator(std::move(webPage)) {}

void authorizedUser()

{

std::cout << "authorized done" << std::endl;

}

void display() override

{

authorizedUser();

WebPageDecorator::display();

}

};

// pure virtual, interface

struct Shape {

virtual ~Shape() = default;

// pure virtual | interface | cannot instantiate directly

virtual std::string GetName() const = 0;

};

struct Circle : Shape {

void Resize(float factor) { radius \*= factor; }

std::string GetName() const override {

return std::string("A circle of radius ") + std::to\_string(radius);

}

// Properties

float radius = 10.0f;

};

struct ColoredShape : Shape

{

ColoredShape(const std::string& color, Shape\* shape)

: color(color), shape(shape) {}

std::string GetName() const override {

return shape->GetName() + " which is colored " + color;

}

std::string color;

Shape\* shape;

// pointer allows this to be a flyweight.

// reuse circle over and over, and you want it to the be same.

// However, modifying one pointer value, will modify all of the circles.

// would need to cast \*shape to a circle to Resize a coloredShape.

};

// Decorator.cpp | <https://www.youtube.com/watch?v=rtfDbdSV4a8>

#include <iostream>

#include <cstdlib>

#include <iostream>

#include <memory>

#include "Shape.h"

#include "Webpage.h"

int main() {

// shape test

Circle circle;

ColoredShape colored\_shape("red", &circle);

std::cout << colored\_shape.GetName() << std::endl;

// you would want casting hidding behind your classes.

auto\* circle\_casted = dynamic\_cast<Circle\*>(colored\_shape.shape);

if (circle\_casted)

{

circle\_casted->Resize(10.f);

std::cout << "Resized!\n";

}

//webpage test, makes a basicWebPage ptr, not an object, no constructor.

std::unique\_ptr<WebPage> myPage = std::make\_unique<BasicWebPage>();

myPage = std::make\_unique<AuthorizedWebPage>(std::move(myPage));

myPage = std::make\_unique<AuthenticatedWebPage>(std::move(myPage));

myPage->display();

std::cout << std::endl;

system("pause");}

## Code Example 2

// Item.h | example of a Decorator pattern

// Give Item an interface for games.

#pragma once

#include <string>

enum ITEMTYPE

{

UNKNOWN,

ARMOR,

WEAPON

};

// Standard Inheritance Model with an Abstract class as root.

class Item {

public:

virtual ~Item() = default;

virtual std::string GetName() const = 0;

// Do we want to give people the option to give an item a name?

virtual void SetName(const std::string& new\_name) { name = new\_name; }

protected:

Item(std::string n) : name(n) {}

Item() = default;

std::string name;

};

// Decorat-ee

class ConcreteWeapon : public Item {

public:

ConcreteWeapon(const ConcreteWeapon&) = delete;

ConcreteWeapon(ConcreteWeapon&&) = delete;

ConcreteWeapon(std::string n, int d) : Item(n), damage(d) {}

virtual std::string GetName() const override { return name; }

int GetDamage() const { return damage; }

protected:

int damage;

};

// Decorat-ee

class ConcreteArmor : public Item {

public:

ConcreteArmor(const ConcreteArmor&) = delete;

ConcreteArmor(ConcreteArmor&&) = delete;

ConcreteArmor(std::string n, int dr) : Item(n), damage\_res(dr) {}

virtual std::string GetName() const override { return name; }

int GetDamageRes() const { return damage\_res; }

protected:

int damage\_res;

};

ITEMTYPE WhatIsItemType(Item\* item)

{

auto\* the\_cast\_to\_armor = dynamic\_cast<ConcreteArmor\*>(item);

if (the\_cast\_to\_armor) {

return ITEMTYPE::ARMOR;

}

auto\* the\_cast\_to\_weapon = dynamic\_cast<ConcreteArmor\*>(item);

if (the\_cast\_to\_weapon) {

return ITEMTYPE::WEAPON;

}

return ITEMTYPE::UNKNOWN;

}

std::string WhatIsItemTypeString(Item\* item) {

auto\* the\_cast\_to\_armor = dynamic\_cast<ConcreteArmor\*>(item);

if (the\_cast\_to\_armor) {

return "Armor";

}

auto\* the\_cast\_to\_weapon = dynamic\_cast<ConcreteWeapon\*>(item);

if (the\_cast\_to\_weapon) {

return "Weapon";

}

return "Unknown";

}

class MagicDecorator : public Item {

public:

MagicDecorator(Item\* item, int mod) : item\_(item), magic\_modifier\_(mod) { }

virtual void SetName(const std::string& new\_name) override { magic\_name\_ = new\_name; }

virtual std::string GetName() const override

{

std::string namestring = magic\_name\_; // if you use item\_->GetName(), it'll use the base class's name.

namestring += " | ";

if (magic\_modifier\_ != 0) {

namestring += (magic\_modifier\_ < 0) ? "-" : "+";

}

namestring += std::to\_string(magic\_modifier\_) + " ";

namestring += item\_->GetName() + " | ";

namestring += GetItemTypeString();

return namestring;

}

std::string GetItemTypeString() const { return WhatIsItemTypeString(item\_); }

int GetMagicMod() { return magic\_modifier\_; }

// Decorator Properties

protected:

Item\* item\_; // Pure interface, we're inheriting from it

// Since weapon is an item, we can pass in weapon

std::string magic\_name\_;

int magic\_modifier\_;

};

// Decorator.cpp | <https://www.youtube.com/watch?v=PrVhfdQ1WRk>

// Decorator 2

#include "item.h"

#include <iostream>

int main() {

ConcreteWeapon weapon1("Basic Sword", 3); // can use with flyweight pattern, if weapon1 is const, use weapon1 everywhere?

MagicDecorator magic\_weapon1(&weapon1, 2);

MagicDecorator x2\_magic\_weapon1(&magic\_weapon1, 2);

//magic\_weapon1.SetName("Armageddon");

std::cout << weapon1.GetName() << " damage: " << weapon1.GetDamage() << '\n';

std::cout << magic\_weapon1.GetName() << " damage: " << (weapon1.GetDamage() + magic\_weapon1.GetMagicMod()) << '\n';

//std::cout << x2\_magic\_weapon1.GetName() << " damage: " << (weapon1.GetDamage() + x2\_magic\_weapon1.GetMagicMod()) << '\n';

// MagicWeapon weapon2("Magic Sword", 3, 2);

// std::cout << weapon2.GetName() << " damage: " << weapon2.GetDamage() << '\n';

system("pause");}

# **Facade**

## Overview

Facade Pattern

* Structural DP provides a simplified interface to a set of interfaces or subsystems.
* An interface acts as a facade or entry point to a more complex system.
* Facade Pattern: Occurs when you already have an interface, and you want to simplify it by having an interface on top of that interface.

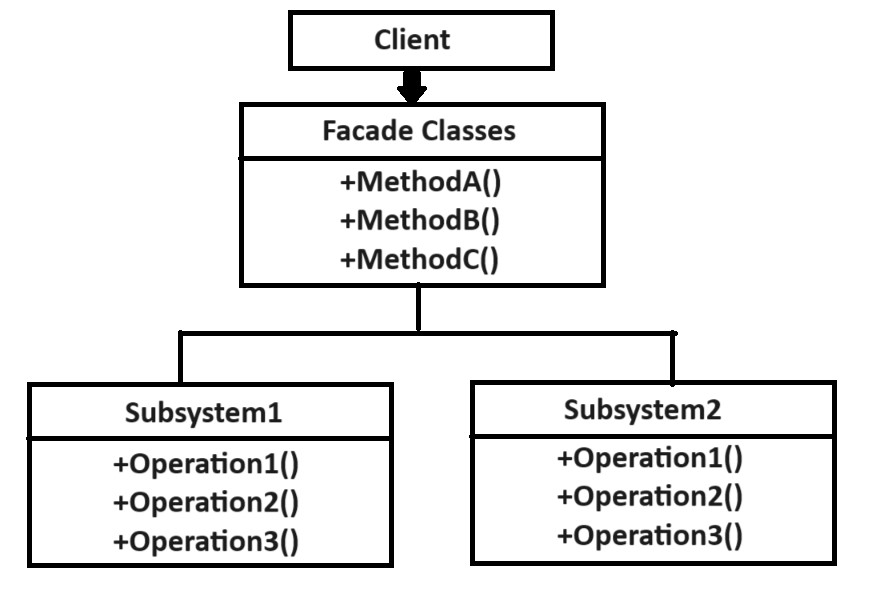
Functional Design

* **Client** represents the class or component that uses the Facade Pattern.
* **Facade Class** is the class that implements the Facade Pattern. It provides a simplified interface to the client and delegates calls to the underlying subsystems.
* **Subsystem 1 and Subsystem 2** are the components or classes that perform specific operations. The client interacts with these subsystems through the facade.

Why / When

* Simplified Interface, Reduced Complexity, Loose Coupling, EZ Maintenance, Improved Testing.
* **Use Cases**: Simplified Library/AP, Subsystem Integration, Legacy Code Integration, Graphics and GUI Systems, Database Access, Complex Config Management, Logging, Debugging, Security, Resource Management, Multi-Platform Dev, Unit Testing, State Management.

## Diagrams



## Code Example

// Facade.cpp | <https://www.youtube.com/watch?v=fsCof7BBHTs>

// Facade Pattern: Occurs when you already have an interface, and you want to

simplify it by having an interface on top of that interface.

// Armor Builder Facade for the Armor class; 'Facade Over Top' the armor interface.

class Armor

{

protected:

virtual void donArmor() = 0; // pure virtual

public:

std::string ArmorType;

};

class HeavyArmor : public Armor {

void donArmor() override {

ArmorType = "Heavy";

}

public:

HeavyArmor() { donArmor(); }

};

class MediumArmor : public Armor {

void donArmor() override {

ArmorType = "Medium";

}

public:

MediumArmor() { donArmor(); }

};

class LightArmor : public Armor {

void donArmor() override {

ArmorType = "Light";

}

public:

LightArmor() { donArmor(); }

};

class ArmorBuilderFacade

{

public:

ArmorBuilderFacade()

{

heavyarmor\_ = new HeavyArmor();

mediumarmor\_ = new MediumArmor();

lightarmor\_ = new LightArmor();

}

~ArmorBuilderFacade()

{

delete heavyarmor\_;

delete mediumarmor\_;

delete lightarmor\_;

}

// public operations that are abstracted away. We want to make automatic.

std::string GetHeavyArmor() { return heavyarmor\_->ArmorType; }

std::string GetMediumArmor() { return mediumarmor\_->ArmorType; }

std::string GetLightArmor() { return lightarmor\_->ArmorType; }

private:

HeavyArmor\* heavyarmor\_;

MediumArmor\* mediumarmor\_;

LightArmor\* lightarmor\_;

// Anytime we want one of these armor types, we want to call a public function above

};

/// ### MAIN ###

#include "armorBuilderFacade.h"

int main()

{

// The point: the facade contains armor class, don't need to #include "armor.h'

// main doesn't need to know about the armor class,facade handles armor interface

ArmorBuilderFacade abf;

std::cout << abf.GetHeavyArmor() << '\n';

std::cout << abf.GetMediumArmor() << '\n';

std::cout << abf.GetLightArmor() << '\n';

system("pause");

}

# **Factory**

## Overview

Factory Pattern

* Creates Object for you, rather you instatiating object directly.
* Call a function that behaves like a factory function, tell factory function what type of object you want.

Functional Design

* Factory Design Pattern also called, 'Virtual Constructor'.
* No virtual constructor in C++ (can't construct abstract class in C++).

How to implement?

// Define an interface or an abstract class for creating an object

// but let the subclasses decide which class to instantiate.

Why / Why

* Less code change if we change object creation process.

## Diagrams

Diagram

Description automatically generated

## Code

class Toy {

protected:

string name;

float price;

public:

virtual void prepareParts() = 0;

virtual void combineParts() = 0;

virtual void assembleParts() = 0;

virtual void applyLabel() = 0;

virtual void showProduct() = 0;

};

class Car : public Toy {

public:

void prepareParts() { cout << "Preparing Car Parts" << endl; }

void combineParts() { cout << "Combining Car Parts" << endl; }

void assembleParts() { cout << "Assembling Car Parts" << endl; }

void applyLabel() { cout << "Applying Car Label" << endl; name = "Car"; price = 10; }

void showProduct() { cout << "Name: " << name << endl << "Price: " << price << endl; }

};

class Bike : public Toy {

public:

void prepareParts() { cout << "Preparing Bike Parts" << endl; }

void combineParts() { cout << "Combining Bike Parts" << endl; }

void assembleParts() { cout << "Assembling Bike Parts" << endl; }

void applyLabel() { cout << "Applying Bike Label" << endl; name = "Bike"; price = 20; }

void showProduct() { cout << "Name: " << name << endl << "Price: " << price << endl; }

};

class Plane : public Toy {

public:

void prepareParts() { cout << "Preparing Plane Parts" << endl; }

void combineParts() { cout << "Combining Plane Parts" << endl; }

void assembleParts() { cout << "Assembling Plane Parts" << endl; }

void applyLabel() { cout << "Applying Plane Label" << endl; name = "Plane"; price = 30; }

void showProduct() { cout << "Name: " << name << endl << "Price: " << price << endl; }

};

class ToyFactory {

public:

static Toy\* createToy(int type) {

Toy\* toy = NULL;

switch (type) {

case 1: {

toy = new Car;

break;

}

case 2: {

toy = new Bike;

break;

}

case 3: {

toy = new Plane;

break;

}

default: {

cout << "invalid toy type please re-enter type" << endl;

return NULL;

}

}

toy->prepareParts();

toy->combineParts();

// Advantage to a Factory: you can add preprocessing here in future

// without accounting for construction behavior of individual objects

toy->assembleParts();

toy->applyLabel();

return toy;

}

};

// Factory Pattern | [Factory Pattern](https://www.youtube.com/watch?v=XyNWEWUSa5E&list=PLk6CEY9XxSIDZhQURp6d8Sgp-A0yKKDKV&index=1)

// Creational Design Pattern

#include <iostream>

#include "ToyFactory.cpp"

using namespace std;

int main() {

int toyType = 0;

while (1)

{

cout << endl << "Enter toyType or Zero for exit" << endl;

cin >> toyType;

if (!toyType)

break;

Q: When can you use a class directly to call a function versus needing to instantiate a local object then call a function?

Q: What does it mean when a class has no defined constructor?

Q: What does it mean when a class has no defined constructor with private members uninitialized?

// Creating objects dynamically at runtime, Client doesn't know anything about how object is constructed.

Toy\* v = ToyFactory::createToy(toyType); //

if (v) {

v->showProduct();

delete v;

}

}

cout << "Exit.." << endl;

system("pause");

}

# **Flyweight**

## Overview

Flyweight Pattern

* A Structural design pattern provides ways to decrease object count.
* Flyweight is used when we need to create a large number of similar objects.

Functional Design

* Essentially, a Map with extra steps.
* Flyweight ensures I only add an image once, and then share it when any other class/function needs it.
* One important feature of flyweight objects is that they are immutable.
* Instance of an item versus a definition of an item, like a MMO Inventory problem.

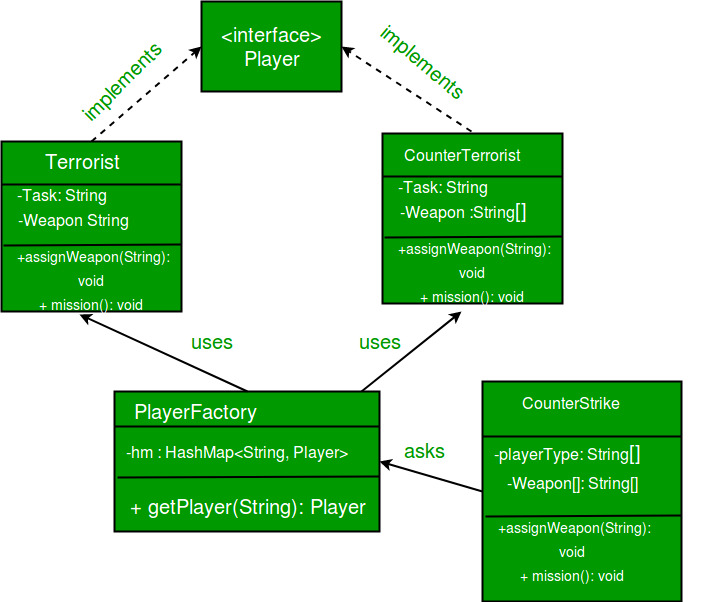
Components

* **Flyweight Interface**: This defines the methods for accessing and manipulating the intrinsic state.
* **Concrete Flyweight**: Implementations of the Flyweight interface that store and manage the intrinsic state. They are typically lightweight and capable of being shared.
* **Flyweight Factory**: A factory class responsible for creating and managing flyweight objects. It ensures that flyweights are shared and reused when possible.

Why / When

* Memory Efficiency, Performance, Object Reuse, Scalability, Simplified Design, Reduced Overhead, Improved Maintenance, Resource Management, Customization, and Consistency
* **Use Cases**: Text processing, Graphic design applications, and Game development (trees, rocks).

## Diagrams



## Code

class Image

{

public:

virtual ~Image() {};

virtual void Print() = 0;

};

class JPG : public Image

{

public:

virtual void Print() override

{

std::cout << "I am a JPG!" << std::endl;

}

};

class PNG : public Image

{

public:

virtual void Print() override

{

std::cout << "I am a PNG!" << std::endl;

}

};

class GIF : public Image

{

public:

virtual void Print() override

{

std::cout << "I am a GIF!" << std::endl;

}

};

class Flyweight\_ImageManager

{

public:

Flyweight\_ImageManager();

~Flyweight\_ImageManager();

template<typename T>

void AddImage(const std::string& resourceName);

void DeleteImage(const std::string& resourceName);

Image\* GetImage(const std::string& resourceName);

private:

std::map<std::string, Image\*> Images;

};

// ### .CPP File ###

Flyweight\_ImageManager::Flyweight\_ImageManager()

: Images() {}

Flyweight\_ImageManager::~Flyweight\_ImageManager()

{

std::cout << "----- DESTRUCTOR ------" << std::endl;

for (std::pair<std::string, Image\*> p : Images)

delete p.second;

Images.clear();

}

template <typename T>

void Flyweight\_ImageManager::AddImage(const std::string& resourceName)

{

std::map<std::string, Image\*>::iterator it = Images.find(resourceName);

if (it == Images.end())

{

Images[resourceName] = new T();

std::cout << "ADD - Added New Image! " << std::endl;

}

else

{

// Could also add more logic to have the same name with multiple image types.

// Could just cout that "Image file name already exists".

delete it->second;

it->second = new T();

std::cout << "ADD - Replaced Image! " << std::endl;

}

}

void Flyweight\_ImageManager::DeleteImage(const std::string& resourceName)

{

std::map<std::string, Image\*>::iterator it = Images.find(resourceName);

if (it != Images.end())

{

delete it->second;

Images.erase(it);

std::cout << "DELETED IMAGE - Image Found " << std::endl;

}

else

{

std::cout << "DELETE IMAGE FAIL - Image NOT Found!" << std::endl;

}

};

Image\* Flyweight\_ImageManager::GetImage(const std::string& resourceName)

{

std::map<std::string, Image\*>::iterator it = Images.find(resourceName);

if (it != Images.end())

{

std::cout << "GetImage - Image Found! " << std::endl;

return it->second;

}

std::cout << "GetImage - No Image Found! " << std::endl;

return nullptr;

};

// Flyweight.cpp | https://www.youtube.com/watch?v=CY0iCHSH7oM

int PrintMenuAndGetChoice()

{

std::cout << "\n1) Add image \n2) Delete image \n3) Get image \n4) Exit program\n?: ";

int choice;

std::cin >> choice;

return choice;

}

void AddImage(Flyweight\_ImageManager\* fManager)

{

std::cout << "---\n ADD \n----\n";

std::cout << "NAME: ";

std::string name;

std::cin >> name;

std::cout << "1) JPG \n2) GIF \n3) PNG \n4) Return\n?: ";

int choice;

std::cin >> choice;

switch (choice)

{

case 1:

fManager->AddImage<JPG>(name);

break;

case 2:

fManager->AddImage<GIF>(name);

break;

case 3:

fManager->AddImage<PNG>(name);

break;

default:

break;

};

return;

}

void DeleteImage(Flyweight\_ImageManager\* fManager)

{

std::cout << "---\n DELETE \n----\n";

std::cout << "NAME: ";

std::string name;

std::cin >> name;

fManager->DeleteImage(name);

}

void GetImage(Flyweight\_ImageManager\* fManager)

{

std::cout << "---\n GET \n----\n";

std::cout << "NAME: ";

std::string name;

std::cin >> name;

Image\* image = fManager->GetImage(name);

if (image == nullptr)

std::cout << "NULL PTR returned" << std::endl;

else

image->Print();

}

int main()

{

// This specific flyweight pattern example is used with image objects

// via the map data structure in the Flyweight\_ImageManager class.

Flyweight\_ImageManager\* Manager = new Flyweight\_ImageManager();

int choice = 0;

bool getChoice = true;

do

{

choice = PrintMenuAndGetChoice();

if (choice == 1)

AddImage(Manager);

else if (choice == 2)

DeleteImage(Manager);

else if (choice == 3)

GetImage(Manager);

else

getChoice = false;

} while(getChoice);

system("pause");

}

# **Interpreter**

## Overview

Interpreter Pattern

* The Interpreter design pattern is a behavioral design pattern that defines a way to interpret and evaluate language grammar or expressions.
* The Interpreter design pattern provides a mechanism to evaluate sentences in a language by representing their grammar as a set of classes.

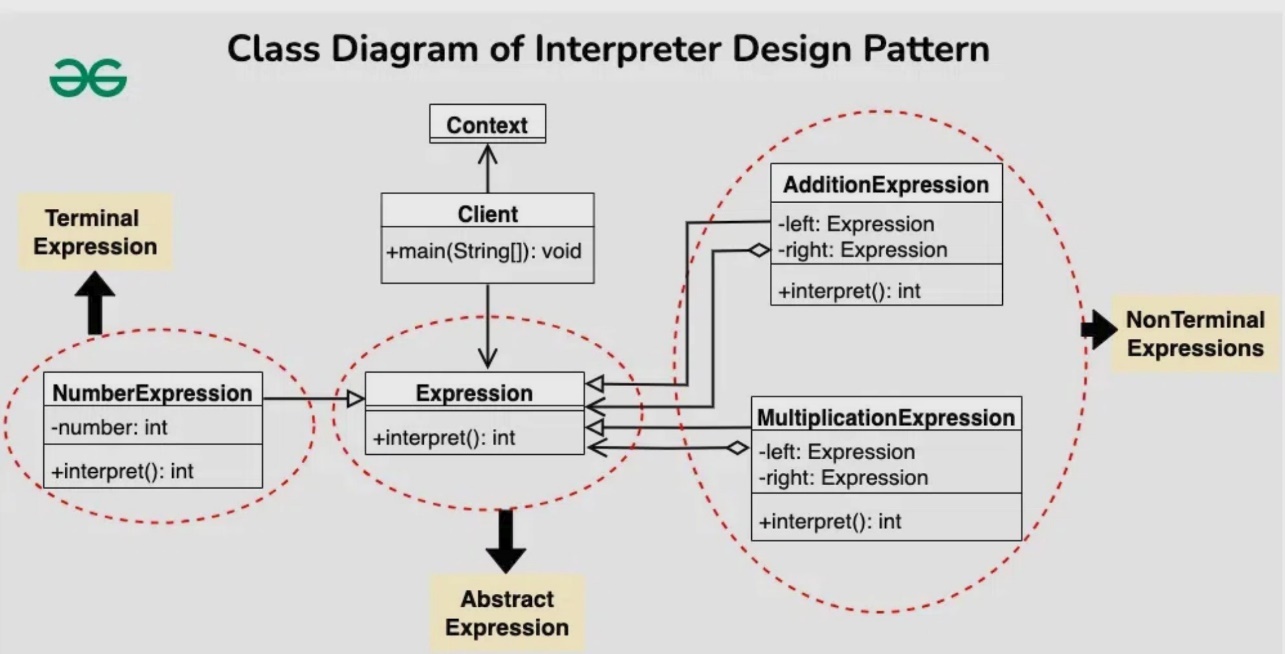
Functional Design / Components

* **Abstract Expression** – Interface with abstract interpret method for all concrete expressions.
* **Terminal Expression** – Concrete exp., building blocks interpreter uses to interpret language.
* **Nonterminal Expression** – Responsbile for handling composite expressions, interpretation logic
* **Context** – Global class that is maintained and modified during the interpretation process.
* **Client** – Creates the abstract syntax tree (AST) and invokes interpret method on the root.
* **Interpreter** - The interpreter is responsible for coordinating the interpretation process.

Why / When

* Modularity, Separation of Concerns, Extensibility,
* Domain-specific languages, interpreting grammar, adding new operations frequently, avoid complex grammar parsers.

## Diagrams



## Code Example

// Interpreter.cpp | <https://sourcemaking.com/design_patterns/interpreter/cpp/1>

class Thousand;

class Hundred;

class Ten;

class One;

class RNInterpreter

{

public:

RNInterpreter(); // ctor for client

RNInterpreter(int) {}

// ctor for subclasses, avoids infinite loop

int interpret(const char\*); // interpret() for client

virtual void interpret(const char\* input, int& total)

{

// for internal use

int index;

index = 0;

if (!strncmp(input, nine(), 2))

{

total += 9 \* multiplier();

index += 2;

}

else if (!strncmp(input, four(), 2))

{

total += 4 \* multiplier();

index += 2;

}

else

{

if (input[0] == five())

{

total += 5 \* multiplier();

index = 1;

}

else

index = 0;

for (int end = index + 3; index < end; index++)

if (input[index] == one())

total += 1 \* multiplier();

else

break;

}

char\* myString = (char\*)input;

//strcpy\_s((char\*)input, sizeof(input), input[index]);

int r = strcpy\_s(myString, sizeof(input), (const char\*)input[index]);

} // remove leading chars processed

protected:

// cannot be pure virtual because client asks for instance

virtual char one() { return ' '; }

virtual const const char\* four() { return nullptr; }

virtual char five() { return ' '; }

virtual const char\* nine() { return nullptr; }

virtual int multiplier() { return 0; }

private:

RNInterpreter\* thousands;

RNInterpreter\* hundreds;

RNInterpreter\* tens;

RNInterpreter\* ones;

};

class Thousand : public RNInterpreter

{

public:

// provide 1-arg ctor to avoid infinite loop in base class ctor

Thousand(int) : RNInterpreter(1) {}

protected:

char one()

{

return 'M';

}

const const char\* four()

{

return nullptr;

}

char five()

{

return '\0';

}

const char\* nine()

{

return nullptr;

}

int multiplier()

{

return 1000;

}

};

class Hundred : public RNInterpreter

{

public:

Hundred(int) : RNInterpreter(1) {}

protected:

char one()

{

return 'C';

}

const const char\* four()

{

return "CD";

}

char five()

{

return 'D';

}

const const char\* nine()

{

return "CM";

}

int multiplier()

{

return 100;

}

};

class Ten : public RNInterpreter

{

public:

Ten(int) : RNInterpreter(1) {}

protected:

char one()

{

return 'X';

}

const const char\* four()

{

return "XL";

}

char five()

{

return 'L';

}

const char\* nine()

{

return "XC";

}

int multiplier()

{

return 10;

}

};

class One : public RNInterpreter

{

public:

One(int) : RNInterpreter(1) {}

protected:

char one()

{

return 'I';

}

const const char\* four()

{

return "IV";

}

char five()

{

return 'V';

}

const char\* nine()

{

return "IX";

}

int multiplier()

{

return 1;

}

};

RNInterpreter::RNInterpreter()

{

// use 1-arg ctor to avoid infinite loop

thousands = new Thousand(1);

hundreds = new Hundred(1);

tens = new Ten(1);

ones = new One(1);

}

int RNInterpreter::interpret(const char\* input)

{

int total;

total = 0;

thousands->interpret(input, total);

hundreds->interpret(input, total);

tens->interpret(input, total);

ones->interpret(input, total);

if (strcmp(input, ""))

// if input was invalid, return 0

return 0;

return total;

}

int main()

{

RNInterpreter interpreter;

char input[20];

std::cout << "Enter Roman Numeral: ";

while (std::cin >> input)

{

std::cout << " interpretation is " << interpreter.interpret(input) << std::endl;

std::cout << "Enter Roman Numeral: ";

}

}

# **Iterator**

## Overview

Iterator Pattern

* Behavioral design pattern for custom iterators based on the classes they are going to work around.
* Provides a way to access the elements of an aggregate object (like a list) sequentially without exposing its underlying representation, type of structure.

Functional Design

* iterator works like an interface to work with any sort of data type.
* Iterators can change the iterator dynamically to allow the iterator to traverse through the data by a different method.

Components

1. Iterator Interface/Abstract Class

Defines the interface for accessing and traversing elements in the collection. It typically includes methods like hasNext(), next(), and optionally remove().

2. Concrete Iterator

Implements the Iterator interface and maintains the current position in the traversal of the aggregate. It provides the actual implementation for the traversal operations defined in the Iterator interface.

3. Aggregate Interface/Abstract Class

Defines the interface for creating an Iterator object. It typically includes a method like createIterator() that returns an Iterator object for the collection.

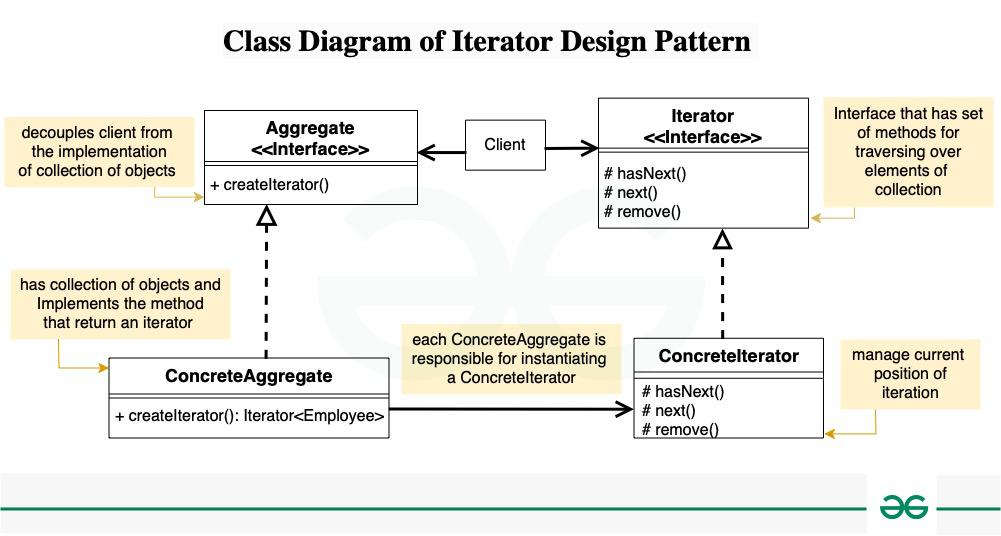
4. Concrete Aggregate

Implements the Aggregate interface and represents the collection of objects. It provides the implementation for creating an Iterator object that can traverse its elements.

Why / Why

* Need access the elements of a collection sequentially without exposing its underlying representation.
* Sequential access, decoupling iteration logic, multiple iterators, simplifying client code.

## Diagrams



## Code Example

// Iterator.cpp | https://www.youtube.com/watch?v=f-icHMaFIp8&list=PLalVdRk2RC6otl3oBU2cn-P6DWi1y1PS3&index=19

#include <iostream>

#include <string>

#include <vector>

// T = type we are iterating through, U = container class

template <typename T, typename U>

class Iterator

{

public: // it\_type = iterator variable name.

typedef typename std::vector<T>::iterator it\_type; // needs to be defined based on these type names.

// iterator only holds weak references of things - so it doesn't need to manually delete any data.

Iterator(U\* container) : \_data\_ref(container) { // member initializer to store member container ptr. during construction.

\_it = \_data\_ref->\_data.begin(); // constructor member assignment for the iterator of the member container ptr.

}

void First() {

\_it = \_data\_ref->\_data.begin();

}

void Next() {

\_it++;

}

bool AtEnd() {

return (\_it == \_data\_ref->\_data.end());

}

it\_type Current() {

return \_it;

}

private:

U\* \_data\_ref; // Iterator doesn't need to hold all the data, just the current data that it's at.

it\_type \_it; // iterator variable typenamed above as a vector<T> iterator

};

template <typename T>

class Container { // bags or storage class are other names for container class

friend class Iterator<T, Container>; // friend class defined in Container class for the custom iterator class above.

// As long as you instantiate the iterator custom class, this container class,

// the iterator class will be able to access private members of this container. Only the iterator can access data.

public:

void Add(T new\_data) {

\_data.push\_back(new\_data);

}

// 'this' is a pointer to this container when creating an Iterator of this custom iterator type, the constructor requires a container ptr.

Iterator<T, Container>\* CreateIterator() {

return new Iterator<T, Container>(this);

}

// TODO: Create a proper to destructor for iterator.

private:

std::vector<T> \_data; // data of the container class needs to match iterator parameters.

};

int main() {

// iterator pattern shines here because no way to get to normally get data from our container template.

// other than to create an iterator. The iterator is generic. All it needs is the right type and the right container.

// But how the iterator is designed, is totally up to the programmer.

// Works great because you can create new iterators based on how you want to specifically sort things for example

// if loads of people online are spamming sort button, instead of trying to use existing iterator, just create a new one.

Container<int> our\_data\_bag;

for (int i = 0; i < 10; i++) {

our\_data\_bag.Add(i);

}

// it's a container of ints and each iterator is also an int.

Iterator<int, Container<int>>\* our\_iterator = our\_data\_bag.CreateIterator();

// auto\* our\_iterator = our\_data\_bag.CreateIterator();

for (our\_iterator->First(); !our\_iterator->AtEnd(); our\_iterator->Next()) {

std::cout << \*our\_iterator->Current() << std::endl;

}

system("pause");

}

# **Mediator**

## Overview

Mediator Pattern

* The Mediator design pattern is a behavioral pattern that defines an object, the mediator, to centralize communication between various components or objects in a system.

Functional Design

* Mediator enables the decoupling of objects by introducing a layer in between so that the interaction between objects happens via the layer.

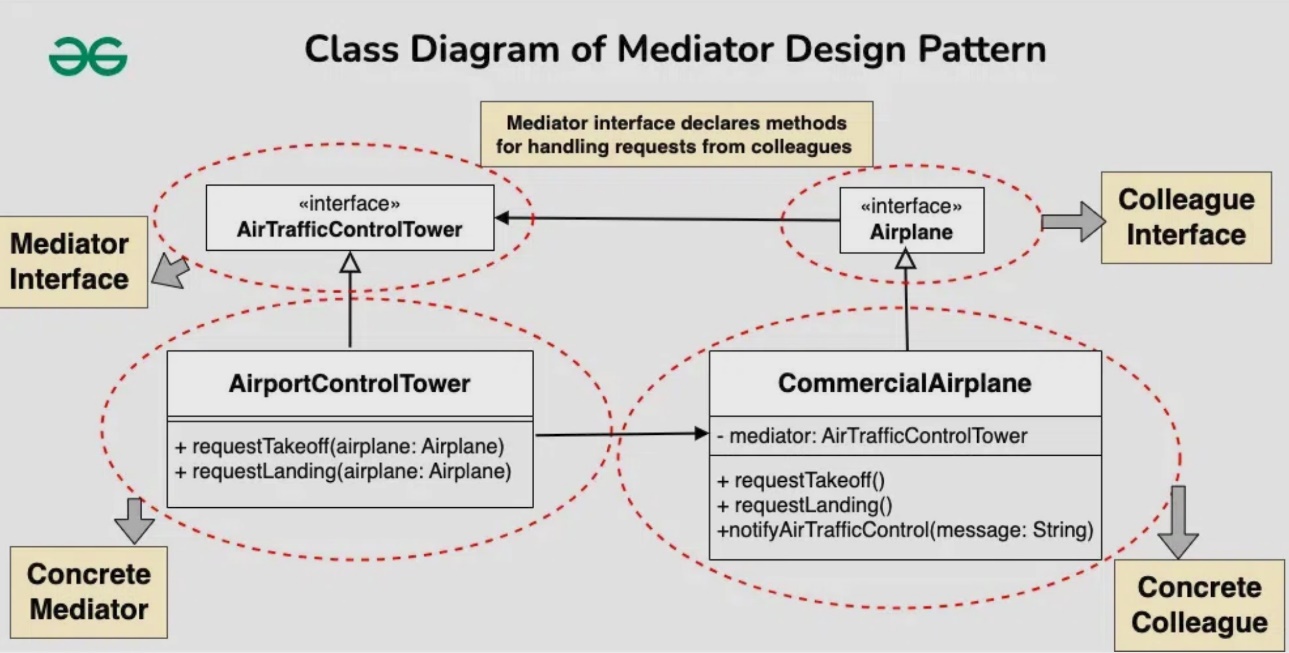
Components

* Mediator – interface communication contract, def. methods, concrete mediators facilitate interactions among colleagues.
* Colleague – components that interact with each other via the mediator without knowing other.
* Concrete Mediator – coordinates communication between concrete colleague objects.
* Concrete colleague – specific implementation of colleague interface.

Why / Why

Complex communication, loose coupling, centralized control, behavior changes, reusability

## Diagrams



## Code Example

// Mediator.cpp | <https://refactoring.guru/design-patterns/mediator/cpp/example>

// This pattern might be best showcased in Unreal Engine.

\* The Mediator interface declares a method used by components to notify the

\* mediator about various events. The Mediator may react to these events and

\* pass the execution to other components.

class BaseComponent;

class Mediator {

public:

virtual void Notify(BaseComponent\* sender, std::string event) const = 0;

};

/\*\*

\* The Base Component provides the basic functionality of storing a mediator's

\* instance inside component objects.

\*/

class BaseComponent {

protected:

Mediator\* mediator\_;

public:

BaseComponent(Mediator\* mediator = nullptr) : mediator\_(mediator) {

}

void set\_mediator(Mediator\* mediator) {

this->mediator\_ = mediator;

}

};

/\*\*

\* Concrete Components implement various functionality. They don't depend on

\* other components. They also don't depend on any concrete mediator classes.

\*/

class Component1 : public BaseComponent {

public:

void DoA() {

std::cout << "Component 1 does A.\n";

this->mediator\_->Notify(this, "A");

}

void DoB() {

std::cout << "Component 1 does B.\n";

this->mediator\_->Notify(this, "B");

}

};

class Component2 : public BaseComponent {

public:

void DoC() {

std::cout << "Component 2 does C.\n";

this->mediator\_->Notify(this, "C");

}

void DoD() {

std::cout << "Component 2 does D.\n";

this->mediator\_->Notify(this, "D");

}

};

/\*\*

\* Concrete Mediators implement cooperative behavior by coordinating several

\* components.

\*/

class ConcreteMediator : public Mediator {

private:

Component1\* component1\_;

Component2\* component2\_;

public:

ConcreteMediator(Component1\* c1, Component2\* c2) : component1\_(c1), component2\_(c2) {

this->component1\_->set\_mediator(this);

this->component2\_->set\_mediator(this);

}

void Notify(BaseComponent\* sender, std::string event) const override {

if (event == "A") {

std::cout << "Mediator reacts on A and triggers following operations:\n";

this->component2\_->DoC();

}

if (event == "D") {

std::cout << "Mediator reacts on D and triggers following operations:\n";

this->component1\_->DoB();

this->component2\_->DoC();

}

}

};

/\*\*

\* The client code.

\*/

void ClientCode() {

Component1\* c1 = new Component1;

Component2\* c2 = new Component2;

ConcreteMediator\* mediator = new ConcreteMediator(c1, c2);

std::cout << "Client triggers operation A.\n";

c1->DoA();

std::cout << "\n";

std::cout << "Client triggers operation D.\n";

c2->DoD();

delete c1;

delete c2;

delete mediator;

}

int main() {

ClientCode();

return 0;

}

# **Memento**

## Overview

Memento Pattern

* Memento Design Pattern is a behavioral design pattern that provides a mechanism for capturing an object’s internal state and restoring it to that state at a later time.
* This pattern is useful when we need to implement features like undo/redo functionality.

Components

* **Originator**: This is the object whose state we want to capture. The Originator creates a Memento object to save its state and uses it to restore its state.
* **Memento**: This is an object that stores the state of the Originator. It has two primary interfaces, getState() returns the saved stateand setState(state) sets the state to a specific value.
* **Caretaker**: This is responsible for holding and managing Memento objects. It doesn’t modify the Memento but can save and restore the state of the Originator.

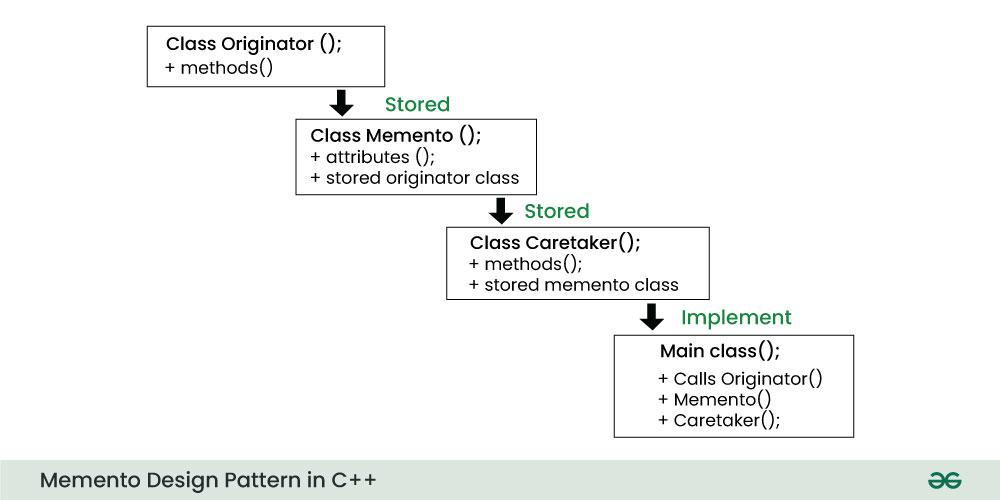
Why

* Easy State Management, Encapsulation of Object State, Undo/Redo, Snapshot and Rollback, Flexibility, Maintains Object Integrity, Extensibility, Improved Testing and Debugging.

Why Not

* Sdlksd

## Diagrams



## Code Example

// Mediator.cpp | <https://refactoring.guru/design-patterns/mediator/cpp/example>

// This pattern might be best showcased in Unreal Engine.

\* The Mediator interface declares a method used by components to notify the

\* mediator about various events. The Mediator may react to these events and

\* pass the execution to other components.

class BaseComponent;

class Mediator {

public:

virtual void Notify(BaseComponent\* sender, std::string event) const = 0;

};

\* The Base Component provides the basic functionality of storing a mediator's

\* instance inside component objects.

class BaseComponent {

protected:

Mediator\* mediator\_;

public:

BaseComponent(Mediator\* mediator = nullptr) : mediator\_(mediator) {

}

void set\_mediator(Mediator\* mediator) {

this->mediator\_ = mediator;

}

};

\* Concrete Components implement various functionality. They don't depend on

\* other components. They also don't depend on any concrete mediator classes.

class Component1 : public BaseComponent {

public:

void DoA() {

std::cout << "Component 1 does A.\n";

this->mediator\_->Notify(this, "A");

}

void DoB() {

std::cout << "Component 1 does B.\n";

this->mediator\_->Notify(this, "B");

}

};

class Component2 : public BaseComponent {

public:

void DoC() {

std::cout << "Component 2 does C.\n";

this->mediator\_->Notify(this, "C");

}

void DoD() {

std::cout << "Component 2 does D.\n";

this->mediator\_->Notify(this, "D");

}

};

Concrete Mediators implement cooperative behavior by coordinating several compenents

class ConcreteMediator : public Mediator {

private:

Component1\* component1\_;

Component2\* component2\_;

public:

ConcreteMediator(Component1\* c1, Component2\* c2) : component1\_(c1), component2\_(c2) {

this->component1\_->set\_mediator(this);

this->component2\_->set\_mediator(this);

}

void Notify(BaseComponent\* sender, std::string event) const override {

if (event == "A") {

std::cout << "Mediator reacts on A and triggers following operations:\n";

this->component2\_->DoC();

}

if (event == "D") {

std::cout << "Mediator reacts on D and triggers following operations:\n";

this->component1\_->DoB();

this->component2\_->DoC();

}

}

};

// Main.cpp

void ClientCode() {

Component1\* c1 = new Component1;

Component2\* c2 = new Component2;

ConcreteMediator\* mediator = new ConcreteMediator(c1, c2);

std::cout << "Client triggers operation A.\n";

c1->DoA();

std::cout << "\n";

std::cout << "Client triggers operation D.\n";

c2->DoD();

delete c1;

delete c2;

delete mediator;

}

int main() {

ClientCode();

return 0;

}

# **Model View Controller (MVC)**

## Overview

Model-View-Controller (MVC) Pattern

Model – View – View Model Pattern

* The Model View Controller (MVC) design pattern specifies that an application consists of a data model, presentation information, and control information separated into 3 separate objects.

Components

* Model – Data and methods to modify data.
* View – Displaying the data, the front end like XAML/XML, to output to the screen.
* Controller - Acts as an intermediary between the Model and the View. Contains references to the Model and View objects. Provides methods to update the Models and Views.

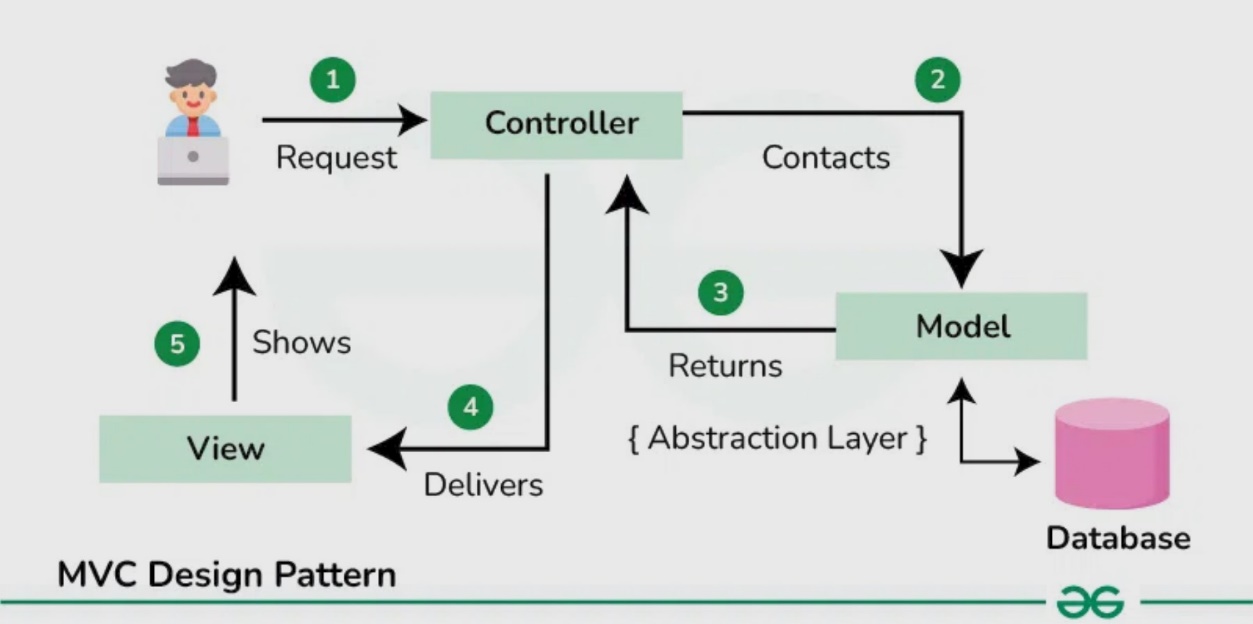
Why

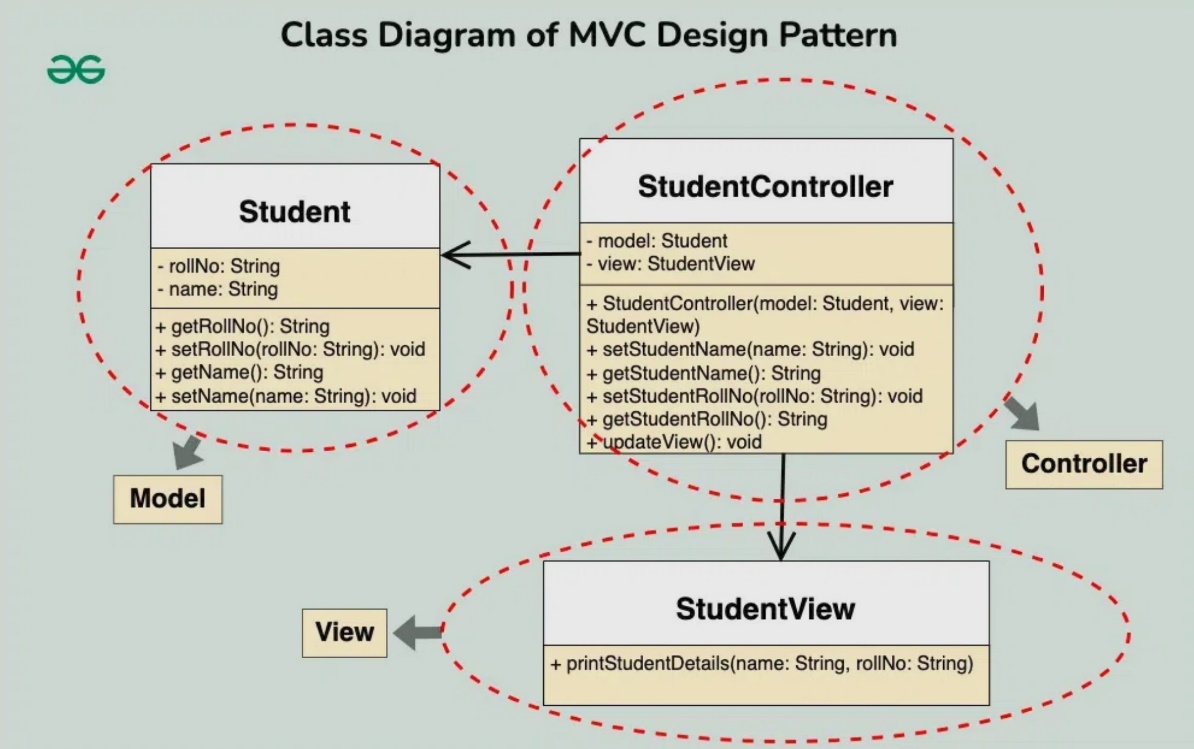
* Separation of concerns, modularity, flexibility, parallel development, code reuse.

Why Not

* Complexity, learning curve, overhead, over-engineering, increased file count.

## Diagrams





## Code Example

// Model\_View\_Controller.cpp | https://www.youtube.com/watch?v=LyoOkle-n50

#include <iostream>

#include <string>

// Everything UI related

// Demonstrate the separation of responsbilities of each component and flow of data.

namespace View

{

class UI

{

public:

// We know this goes to the View portion of pattern, but it interacts with the user.

void prompt(std::string s)

{

std::cout << s << std::endl;

}

void prompt(int i)

{

std::cout << i << std::endl;

}

int getInput()

{

int x;

std::cin >> x;

return x;

}

};

};

// Everything data and storage related, and services on that data

namespace Model

{

class Storage

{

private:

int x;

public:

void storeValue(int n)

{

x = n;

}

int retrieve()

{

return x;

}

};

};

// Everything that is business logic related

namespace Controller // ViewModel

{

class Logic

{

View::UI\* ui;

Model::Storage\* store;

public:

Logic(View::UI\* u, Model::Storage\* s)

{

ui = u;

store = s;

doLogic();

}

void doLogic()

{

// ask user for number using UI object (view)

ui->prompt("Enter a number: ");

// store number in Storage object (Model)

int i = ui->getInput();

store->storeValue(i);

// retrieve number from storage object (Model)

int x = store->retrieve();

// display that to the user using the UI object (view)

ui->prompt("Here is the number you entered: ");

ui->prompt(x);

ui->prompt("\n");

}

};

};

int main()

{

View::UI u;

Model::Storage s;

Controller::Logic logic(&u, &s);

system("pause");

}

# **Null Object**

## Overview

Null Object Pattern

* The Null Object Design Pattern is a [behavioral design pattern](https://www.geeksforgeeks.org/behavioral-design-patterns/) that is used to provide a consistent way of handling null or non-existing objects.

Functional Design

* The Null Object pattern simplifies the use of dependencies that can be undefined.
* Client, Abstract Dependency, Concrete Dependency, NullObject components.
* The Null Object Pattern is useful in situations where you want to provide a default or no-op implementation of an obj’s behavior to avoid null checks and handle null references gracefully.

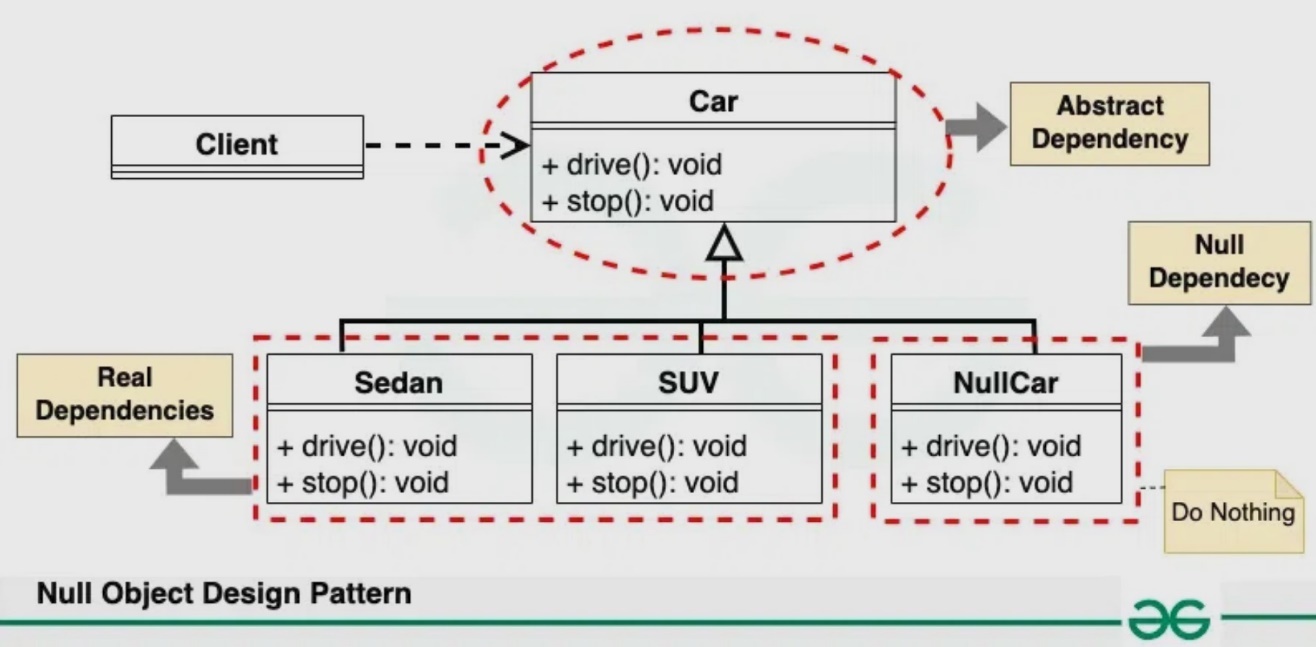
Why / When

* Default Behavior, Avoid Null Checks, Consistent Interface, Simplifying Client Code.

Why Not

* Complex Behavior, Performance Consideration, Confusion with Real Objects.

## Diagrams



## Code Example

// ObjectAnimal.h

class IAnimal {

public:

virtual ~IAnimal() = default;

virtual void MakeSound() const = 0;

};

class Dog : public IAnimal {

public:

virtual void MakeSound() const override {

std::cout << "Wolf Bark Growl\n";

}

};

// allows you to put in as a default parameter that gets passed into functions

// as your building the functionality out, and you want to have it exit before

// cascading to a bunch of other function calls, but still goes through call stack

// without crashing, linker/logic error checker.

class NullAnimal : public IAnimal {

public:

virtual void MakeSound() const override {}

};

// NullObject.cpp | [Null Object Design Pattern YouTube](https://www.youtube.com/watch?v=e3epYFGrS88&list=PLalVdRk2RC6otl3oBU2cn-P6DWi1y1PS3&index=20)

// Main.cpp

#include "object.h"

// real instantiation that you are accessing with references

// there is no such instantiation possible with an interface

void control\_animal(const IAnimal& animal) {

animal.MakeSound();

}

// this one can pass null into here.

void control\_animal2(const IAnimal\* animal) {

animal->MakeSound(); // need a null check

}

// Design a project where you don't have to do Null checks.

// YOU'LL be able to know if you program runs correctly and don't need ptrs.

// Only accept references which are actual instantiations.

int main() {

Dog dog1;

control\_animal(dog1);

// probably should make it a proxy, and have one staticly initialized at all times.

NullAnimal nullanimal;

// it's tries to take over or call, but there's nothing to call

control\_animal(nullanimal);

system("pause");

}

# **Object Pool**

## Overview

Object Pool Pattern

* Object pool is a container which contains some amount of objects. So, when an object is taken from the pool, it is not available in the pool until it is put back.

Functional Design

* Creational design pattern where the cost of initializing a class instance is very high.

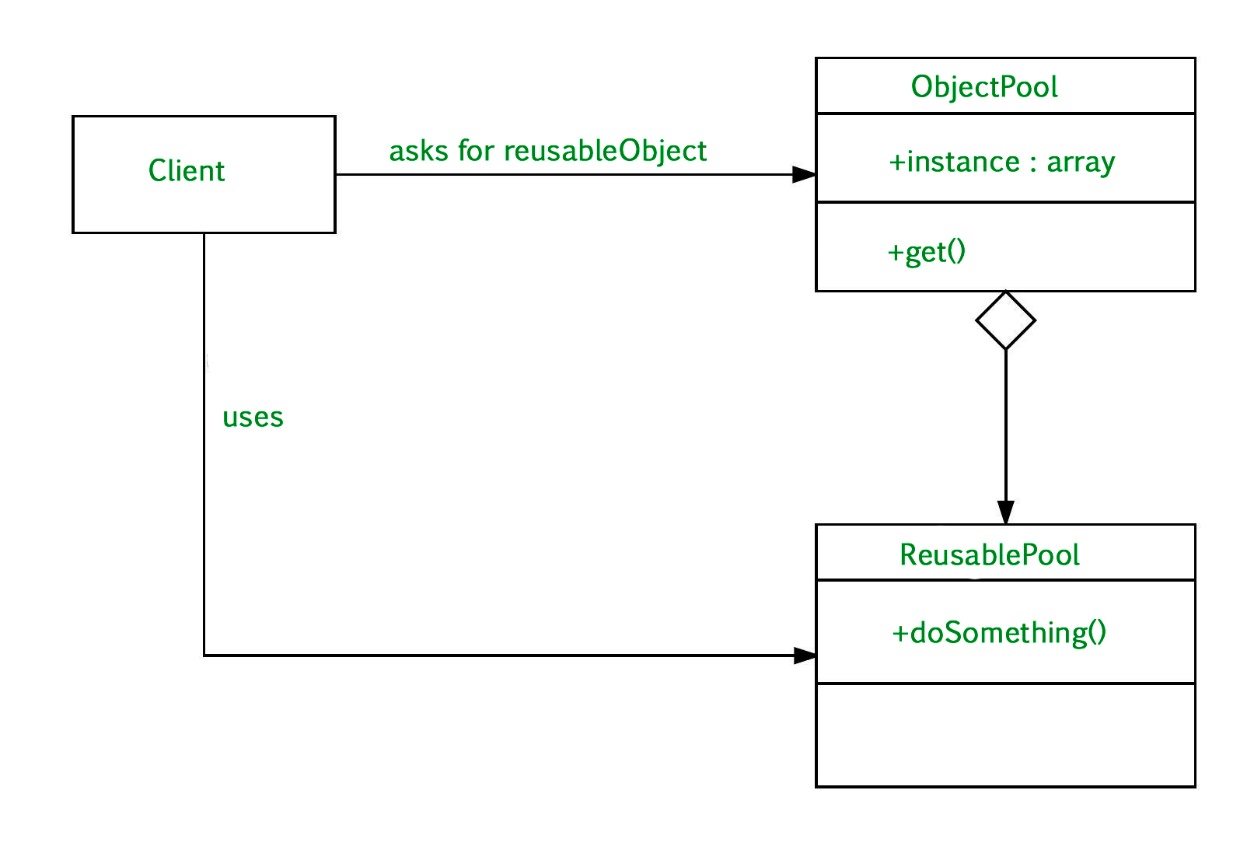
Components of Object Pool Design Pattern

* **Client** : This is the class that uses an object of the PooledObject type.
* **ReuseablePool**: The PooledObject class is the type that is expensive or slow to instantiate, or that has limited availability, so is to be held in the object pool.
* **ObjectPool** : The Pool class is the most important class in the object pool design pattern. ObjectPool maintains a list of available objects and a collection of objects that have already been requested from the pool.

Why / When

* Object Pool pattern is used for caching, keeping memory aligned, not re-creating variables, avoids constant allocating and de-allocating of memory.
* Create the object once, pull it when you need it, put it back into the pool when done.
* When we have a work to allocates or deallocates many objects
* When we know that we have a limited num of objects that will be in memory at the same time

## Diagrams



Alkjd

## Code Example

**// Resource.h**

// Resource is just an integer for illustration purposes.

class Resource {

private:

int value;

public:

Resource() {

value = 0;

}

void reset() {

value = 0;

}

int getValue() {

return value;

}

void setValue(int number) {

value = number;

}

};

// Object Pool

class ObjectPool {

private:

ObjectPool() {} // default constructor

static ObjectPool\* instance; // \*\* NEED to define where it is used.

std::list<Resource\*> resources;

public:

// Static method for accessing class instance.

// Part of Singleton design pattern. Return ObjectPool instance.

static ObjectPool\* getInstance() {

if (instance == 0) {

instance = new ObjectPool;

}

return instance;

}

// Returns instance of Resource.

// New resource will be created if all the resources

// were used at the time of the request. return Resource instance.

Resource\* getResource()

{

// Resources is a member variable list, checks if a Resource\* is empty

if (resources.empty()) {

std::cout << "Creating new." << std::endl;

// if resource list empty, create a new resource and return it.

return new Resource;

}

else {

std::cout << "Reusing existing." << std::endl;

Resource\* resource = resources.front();

resources.pop\_front();

return resource;

}

}

// Return resource back to the pool. The resource must be initialized back

// to the default settings before someone else attempts to use it.

// param object Resource instance.

// return void

void returnResource(Resource\* object) {

object->reset();

resources.push\_back(object);

}

};

// **Q:** Difference between Object Pool and Prototype?

**// ObjectPool.cpp | Main.cpp |** | [**ObjectPool Design Pattern YouTube**](https://www.youtube.com/watch?v=lNTYs72Hi_0&list=PLalVdRk2RC6otl3oBU2cn-P6DWi1y1PS3&index=4)

#include "objectpool.h"

// Object Pool is a manager, so you only want one instance of it for

// allocating and de-allocating objects.

ObjectPool\* ObjectPool::instance = nullptr;

int main() {

ObjectPool\* pool = ObjectPool::getInstance();

Resource\* one;

Resource\* two;

/\* Resources will be created. \*/

one = pool->getResource();

one->setValue(10);

std::cout << "one = " << one->getValue() << " [" << one << "]" << std::endl;

two = pool->getResource();

two->setValue(20);

std::cout << "two = " << two->getValue() << " [" << two << "]" << std::endl;

pool->returnResource(one);

pool->returnResource(two);

// Resources will be reused.

// Notice that the value of both resources were reset back to zero.

one = pool->getResource();

std::cout << "one = " << one->getValue() << " [" << one << "]" << std::endl;

two = pool->getResource();

std::cout << "two = " << two->getValue() << " [" << two << "]" << std::endl;

system("pause");

# **Observer**

## Overview

Observer Pattern

* The Observer Pattern is a behavioral design pattern that defines a one-to-many dependency between objects, meaning that when one object (the subject) changes its state, all its dependents (observers) are notified and updated automatically.

Object Pool pattern is used for caching, keeping memory aligned, not re-creating variables, avoids constant allocating and de-allocating of memory. Create the object once, pull it when you need it,

put it back into the pool when done.

A pattern to allow notifications to be sent to anyone who subscribes.

Overall the point is to not have users manually check something, but rather be sent a notification whenever a certain event happens to a subject.

Need a publishing service (subject). Something that goes with your business logic.

Need an array of all the subscribers / observers.

THE POINT:

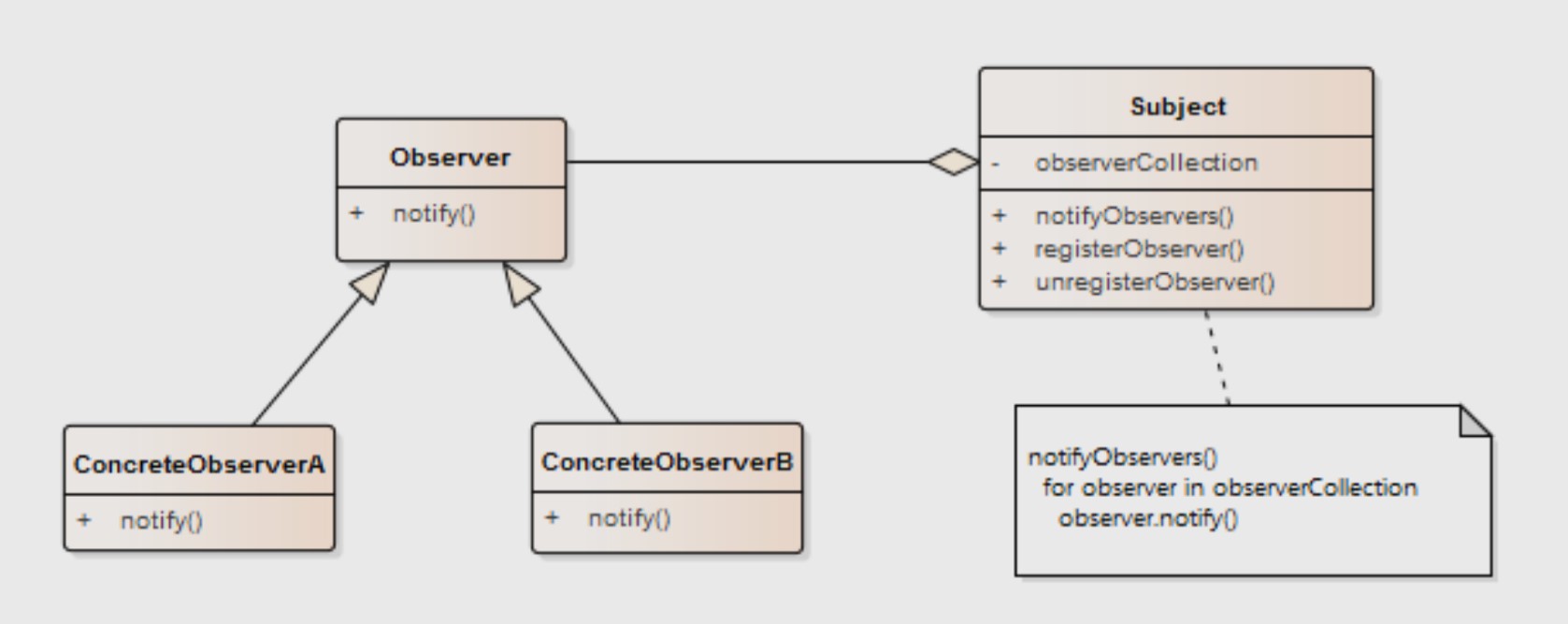
The observer pattern can get as complex as you want, to simplify some of the things down, once you have a lot of things going on, but you want other things to know about it too, without having direct access to the whole class.

Q: Difference between Object Pool and Prototype?

Why / When

* **Decoupling**: The Observer Pattern promotes loose coupling between subjects and observers. Subjects don’t need to know the concrete types of their observers.
* **Scalability**: You can easily add or remove observers without modifying the subject. This makes it a flexible solution for systems with dynamic requirements.
* **Reusability**: Observers can be reused in different contexts, provided they adhere to the observer interface or class.
* **Event** **Handling**: The pattern is instrumental in event handling systems, such as GUI frameworks, where components need to respond to user actions.

## Diagrams



## Code Example

#include "subscriber.h"

#include <list>

// Can be done with Composition pattern as well, the 'pimple' idiom.

// Instead of base concrete class, 'Subject', you would instead, you would have the subject class already pre - defined, and then just add it.

// Below the 'Subject' class, we would have another class that doesn't inherit from 'Subject', but has a private member of type Subject pointer.

// Subject class needs to forward all the main functions from Subject.

// and use the private member to add an observers to observer list.

class SubjectInterface {

public:

virtual ~SubjectInterface() {};

virtual void Add(SubscriberInterface\* sub) = 0; // pointers make it easy to move observers around.

virtual void Remove(SubscriberInterface\* sub) = 0;

virtual void Notify() = 0;

};

class Subject : public SubjectInterface {

public:

// List is great because we can use simple methods with low overhead from the STL.

void Add(SubscriberInterface\* sub) override {

\_subs.push\_back(sub);

};

void Remove(SubscriberInterface\* sub) override {

\_subs.remove(sub);

};

void Notify() override {

if (\_subs.empty())

return;

std::list<SubscriberInterface\*>::iterator it = \_subs.begin();

while (it != \_subs.end()) {

(\*it)->Update(\_latest\_message);

it++;

}

};

void AdvanceAndUpdateAll() {

static int num\_updates = 0;

\_latest\_message = "Update Number \"";

\_latest\_message += std::to\_string(++num\_updates);

\_latest\_message += "\" Processed!";

Notify();

}

// TODO: Add some cleanup steps, no duplicateso

private:

// important that all subscribers / observers use the same interface.

std::list<SubscriberInterface\*> \_subs;

std::string \_latest\_message = "default message";

};

// list of the Subject class indirectly.

#pragma once

#include <string>

#include <iostream>

#include "subject.h"

class SubscriberInterface {

public:

virtual ~SubscriberInterface() {}

friend class Subject;

protected:

// Typically called by the subject in the notify

virtual void Update(const std::string& message) = 0;

};

class Subscriber : public SubscriberInterface {

public:

Subscriber() {

\_unique\_id = ++\_unique\_subscribers;

}

~Subscriber() {}

protected:

void Update(const std::string& message) override {

std::cout << "SUBID: \"" << \_unique\_id << "\" recieved message -> " << message << std::endl;

};

private:

int \_unique\_id = -1;

static int \_unique\_subscribers;

};

int Subscriber::\_unique\_subscribers = 0;

// **Observer.cpp** | [Observer Design Pattern YouTube](https://www.youtube.com/watch?v=Th1A2szPctI&list=PLalVdRk2RC6otl3oBU2cn-P6DWi1y1PS3&index=13)

// Subscriber pattern

// Observe what's going on elsewhere without being directly involved.

#include "subject.h"

int main() {

Subject\* channel = new Subject();

Subscriber\* subscriber = new Subscriber();

channel->Add(subscriber);

channel->AdvanceAndUpdateAll();

channel->AdvanceAndUpdateAll();

channel->AdvanceAndUpdateAll();

channel->AdvanceAndUpdateAll();

channel->AdvanceAndUpdateAll();

Subscriber\* subscriber2 = new Subscriber();

channel->Add(subscriber2);

channel->AdvanceAndUpdateAll();

channel->AdvanceAndUpdateAll();

channel->AdvanceAndUpdateAll();

channel->AdvanceAndUpdateAll();

channel->AdvanceAndUpdateAll();

system("pause");

}

# **Observer 2**

## Overview

See above

## Code Example

class car {

public:

int getPosition();

void setPosition(int newPosition);

void setTmp(int newTmp);

void attach(class observer\* obs);

void detach(class observer\* obs);

void notify();

void notifyTmp();

private:

int position;

int tmp;

std::vector<class observer\*> observerList;

std::vector<class observer\*> observerTmpList;

};

#include "car.h"

using namespace std;

class observer

{

public:

observer(car\* car) {

\_car = car;

\_car->attach(this);

}

virtual void update() = 0;

protected:

car\* getCar() {

return \_car;

}

private:

car\* \_car;

};

class Leftobserver : public observer {

public:

Leftobserver(car\* car) : observer(car) {}

void update() {

int pos = getCar()->getPosition();

if (pos < 0) {

cout << "left side" << endl;

}

}

};

class Rightobserver : public observer {

public:

Rightobserver(car\* car) : observer(car) {}

void update() {

int pos = getCar()->getPosition();

if (pos > 0) {

cout << "right side" << endl;

}

}

};

class Middleobserver : public observer {

public:

Middleobserver(car\* car) : observer(car) {}

void update() {

int pos = getCar()->getPosition();

if (pos == 0) {

cout << "running in middle" << endl;

}

}

};

#include <iostream>

#include <vector>

#include <algorithm>

#include "observer.h"

#include "car.h"

// ctrl+k, ctrl+o to switch between .h and .cpp

using namespace std;

int car::getPosition()

{

return position;

}

void car::setPosition(int newPosition)

{

position = newPosition;

notify();

}

void car::setTmp(int newTmp)

{

tmp = newTmp;

notifyTmp();

}

void car::attach(class observer\* obs)

{

observerList.push\_back(obs);

}

void car::detach(class observer\* obs)

{

observerList.erase(std::remove(observerList.begin(), observerList.end(), obs), observerList.end());

}

void car::notify()

{

for (int i = 0; i < observerList.size(); ++i)

{

observerList[i]->update();

}

}

void car::notifyTmp()

{

for (int i = 0; i < observerTmpList.size(); ++i)

{

observerTmpList[i]->update();

}

}

// Observer\_Pattern.cpp | https://www.youtube.com/watch?v=rF3TfkknQK0

// Observer Design Pattern is a pattern that defines a one-to-many

// dependencies between objects so that when one object changes state,

// all it's dependents are notified and update automatically.

// Subject has list of observers.

#include <iostream>

#include <algorithm>

#include "car.h"

#include "observer.h"

int main()

{

// subject

car\* c = new car();

// observers - one car to many observers

Leftobserver Leftobserver(c);

Rightobserver rightobserver(c);

Middleobserver middleobserver(c);

cout << "hit left right button to drive a car in your city!!! and press break to close" << endl;

char pressedButton;

bool breakLoop = false;

while (breakLoop == false) {

cin >> pressedButton;

switch (pressedButton) {

case 108: { // l --> pressed for left side

c->setPosition(-1);

break;

}

case 99: { // c --> pressed for center

c->setPosition(0);

break;

}

case 114: { // r --> pressed for right side

c->setPosition(1);

break;

}

case 98: { // b --> pressed for break

breakLoop = true;

break;

}

default: {

cout << "please drive carfully!!" << endl;

break;

}

}

}

cout << "Byee..." << endl;

system("pause");

}

// There are observer classes that will be called when the subjects state changes

// Subjects have a list of observer classes.

// The subjects have event dispatchers that the observers listen for

// when the subject's state changes and it sends out notifies.

# **Prototype**

## Overview

Prototype Pattern

* A creational design pattern that specifically focuses on creating objects efficiently by copying an existing object, which we call the Prototype.

Functional Design

* Creating an object is more expensive than copying an object.
* All Object Initial State is common and takes time to build.
* Hide the complexity of creating new instance from the user.
* Which classes to create are specified at runtime.

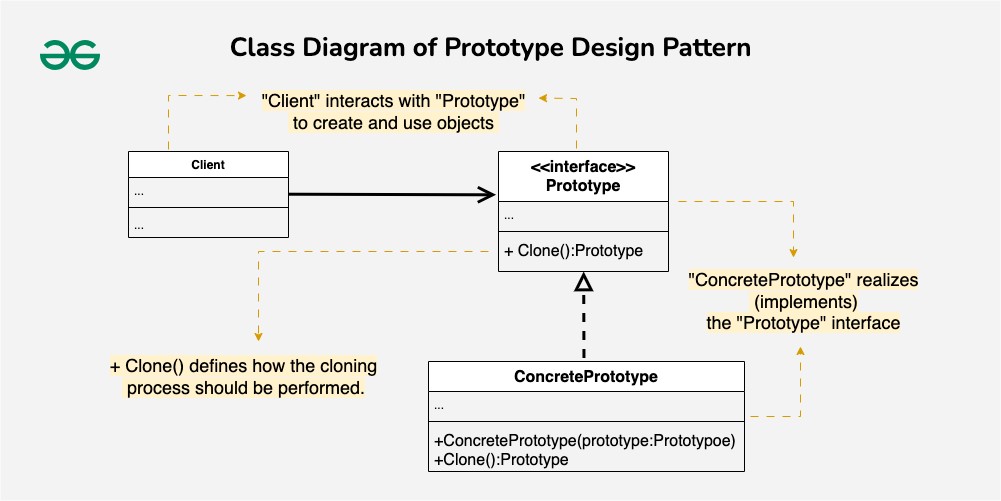
Why / When

* Well, it’s all about saving time and resources. Instead of creating new objects from scratch, the Prototype Pattern lets us create new ones by copying an existing prototype.

Why not

* Complexity, Not suitable for all scenarios, memory usage

## Diagrams



## Code Example

// Prototype.cpp | [Prototype Design Pattern YouTube](https://www.youtube.com/watch?v=KBkkEKNlE6I&list=PLk6CEY9XxSIDZhQURp6d8Sgp-A0yKKDKV&index=6)

// Store some dummy object that you always want, so you will always get that object.

// And then only you will start working with that object. Don't create object again and again.

/\*\* Bullet is the base Prototype \*/

class Bullet

{

public:

Bullet() {}

Bullet(string bulletName, float speed, float firePower, float damagePower)

: \_bulletName(bulletName), \_speed(speed), \_firePower(firePower), \_damagePower(damagePower)

{}

virtual ~Bullet() {}

virtual unique\_ptr<Bullet> clone() = 0;

void fire(float direction)

{

\_direction = direction;

cout << "Name : " << \_bulletName << endl

<< "Speed : " << \_speed << endl

<< "FirePower : " << \_firePower << endl

<< "DamagePower : " << \_damagePower << endl

<< "Direction : " << \_direction << endl << endl;

}

protected:

string \_bulletName = "";

float \_speed = 0.0f;

float \_firePower = 0.0f;

float \_damagePower = 0.0f;

float \_direction = 0.0f;

};

/\*\* SimpleBullet is a Concrete Prototype \*/

class SimpleBullet : public Bullet

{

public:

SimpleBullet(string bulletName, float speed, float firePower, float damagePower) :

Bullet(bulletName, speed, firePower, damagePower)

{

}

// This is the method you'll be using for PROTOTYPING your object.

// Whoever is implementing this abstract class, they have to

// their own implementation of this clone, and return their own type.

unique\_ptr<Bullet> clone() override

{

return make\_unique<SimpleBullet>(\*this);

}

};

/\*\* GoodBullet is the Concrete Prototype \*/

class GoodBullet : public Bullet

{

public:

GoodBullet(string bulletName, float speed, float firePower, float damagePower)

: Bullet(bulletName, speed, firePower, damagePower)

{

}

unique\_ptr<Bullet> clone() override

{

return std::make\_unique<GoodBullet>(\*this);

}

};

/\*\* Opaque Bullet type, avoids exposing concrete implementations \*/

enum BulletType

{

SIMPLE,

GOOD

};

/\*\* BulletFactory is the client \*/

class BulletFactory

{

private:

unordered\_map<BulletType, unique\_ptr<Bullet>, hash<int> > m\_Bullets;

public:

BulletFactory()

{

m\_Bullets[SIMPLE] = make\_unique<SimpleBullet>("Simple Bullet", 50, 75, 75);

m\_Bullets[GOOD] = make\_unique<GoodBullet>("Good Bullet", 75, 100, 100);

}

unique\_ptr<Bullet> createBullet(BulletType BulletType)

{

return m\_Bullets[BulletType]->clone();

}

};

int main()

{

// When the BulletFactory is created, one instance of a bullet per bullet type

// is created and stored in the factory's unordered\_map.

BulletFactory bulletFactory;

// Creates a copy of a SIMPLE bullet already stored from the factory,

// instead of instatiating a whole new object

// All other properties of bullet are set on the original instance of the bullet except direction.

// Direction is set when the Bullet->Fire(direction) is called.

unique\_ptr<Bullet> Bullet = bulletFactory.createBullet(SIMPLE);

Bullet->fire(90);

Bullet = bulletFactory.createBullet(GOOD);

Bullet->fire(100);

}

# **Proxy**

## Overview

Prototype Pattern

**Proxy.cpp** | [**Proxy Pattern**](https://www.youtube.com/watch?v=9yY6Bc2cgYU)

* A structural design pattern that provides a surrogate or placeholder for another object to control access to it.
* Proxy puts a class in front of your real process that adds on a little additional functionality.

Functional Design

* Proxies act as a substitute or placeholder for another object.
* A proxy controls access to the original obj. acting as an intermediary, controlling access to real object.
* Subject(image interface), Real Subject(real image class), Proxy (proxy image class), Client Code

Why / When

* Lazy Loading, Access Control, Protection Proxy, Caching, Logging and Monitoring
* Skd

Questions and Answers

**Q**: Why not use child inheritance to add functionality instead of using a proxy?

**A**: In an ideal world, we’d want to put this code directly into our object’s class,

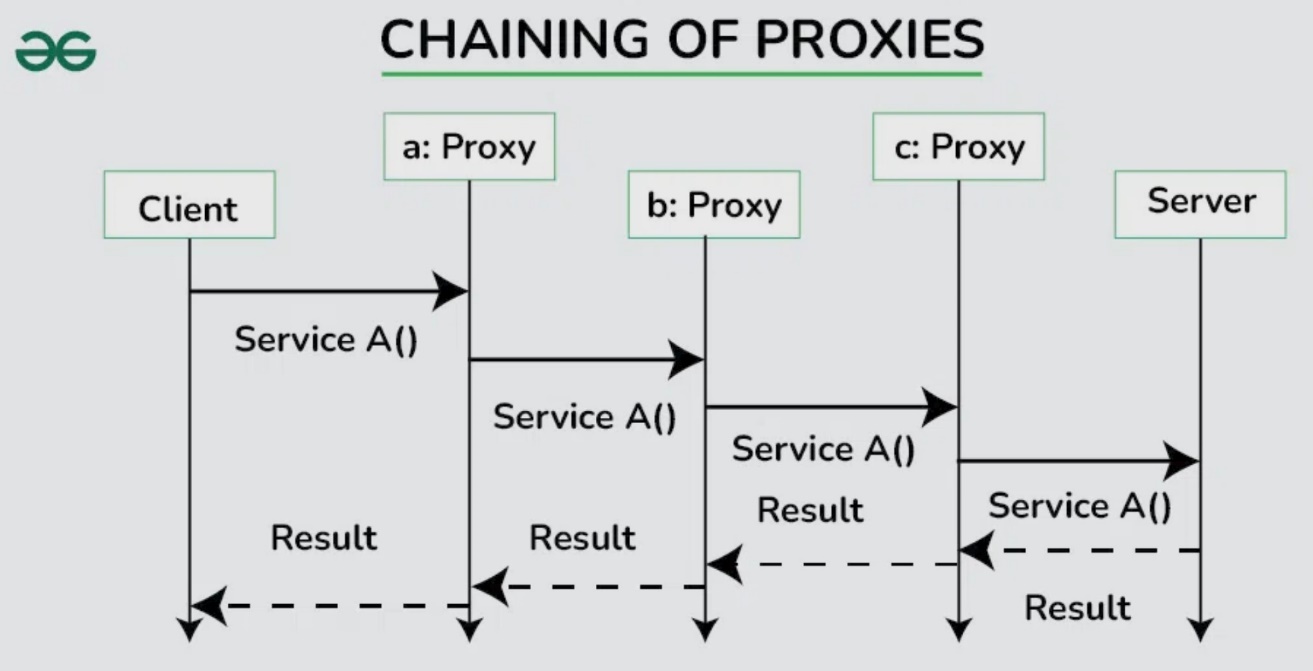
but that isn’t always possible. For instance, the class may be part of a CLOSED 3rd-party library.

**A:** If the superclass (RealSubject) is not under your control, i.e. in the same package,and specifically designed and documented for Extension, any changes to it's implementation from version to version may break your implementation of the subclass (EnhancedSubject).Depending directly on a concrete implementation leads to fragile code.

**A:** It is safe to use inheritance if the RealSubject and EnhancedSubject are under your control and released with the same life cycle, but

Depending directly on a concrete implementation leads to fragile code.

## Diagrams



## Code Example

**// proxy.h**

// Online store that tie services together.

// Hackers use proxies on stores to take some of transaction money, routing it to their account.

#include <vector>

#include <iostream>

enum PaymentType { CREDIT, BANK, WIRE, CRYPTO };

bool Charge(const PaymentType payment\_, int amount\_to\_charge) {

return true;

}

// ### ITEM ###

struct ItemForSale {

int sku;

int cost\_per;

int num\_in\_stock;

ItemForSale(int sku, int cost\_per, int num\_in\_stock)

: sku(sku), cost\_per(cost\_per), num\_in\_stock(num\_in\_stock){}

};

// ### Request Interface ###

class Request {

public:

virtual ~Request() = default;

virtual void AddInventory(ItemForSale itemForSale) = 0;

virtual int RequestItem(const int item\_sku, const PaymentType payment\_) = 0;

protected:

std::vector<ItemForSale> current\_inventory;

};

// ### Item Request ###

class ItemRequest : Request {

public:

virtual ~ItemRequest() {}

virtual void AddInventory(ItemForSale itemForSale) override {

// not allow duplicate skus, instead add to num in stock

for (ItemForSale item : current\_inventory) { // adds to number in stock

if (item.sku == itemForSale.sku) {

item.num\_in\_stock += itemForSale.num\_in\_stock;

return;

}

}

current\_inventory.push\_back(itemForSale); // if unique, add to current inventory vector.

}

// returns amount charged, returns 0 if item not found,returns -1 if payment error,

virtual int RequestItem(const int item\_sku, const PaymentType payment\_) override {

for (auto& item : current\_inventory) {

if (item.sku == item\_sku && item.num\_in\_stock > 0) {

if (Charge(payment\_, item.cost\_per)) {

item.num\_in\_stock--;

return item.cost\_per;

}

return -1;

}

}

return 0;

}

};

// ### PROXY - Check Request ###

// Whole point of the proxy is to do additional stuff

// Proxy\_CheckBeforeRequest puts a proxy on top of the ItemRequest

class Proxy\_CheckBeforeRequest : Request {

private:

// Proxy\_CheckBeforeRequest doesn't inherit from ItemRequest (just Request)

// but because of this member variable, it will instantiate ItemRequest's functionality

ItemRequest\* item\_request; // The constructor requires an ItemRequest be passed in.

void LogInventoryAdd(ItemForSale itemForSale) {

std::cout << "attempted to add sku: " << itemForSale.sku << '\n';

}

bool PaymentCheck(const PaymentType payment\_) {

// pre-check payment for legality, make sure account exits/card isn't expired, etc

return true; // THIS CODE Would be fleshed out, just for example sake, it'll always return true.

}

public:

Proxy\_CheckBeforeRequest(ItemRequest\* item\_req) : item\_request(new ItemRequest(\*item\_req)) {};

~Proxy\_CheckBeforeRequest() { delete item\_request; }

// Adds LogInventory to the AddInventory function, on top of the standard ItemRequest.Add()

virtual void AddInventory(ItemForSale itemForSale) override {

LogInventoryAdd(itemForSale); // PROXY EXAMPLE

item\_request->AddInventory(itemForSale);

}

virtual int RequestItem(const int item\_sku, const PaymentType payment\_) override {

// adds 'declined payment' functionality to the RequestItem

if (!PaymentCheck(payment\_)) { // PROXY EXAMPLE

return -1;

}

return item\_request->RequestItem(item\_sku, payment\_);

}

};

#include <iostream>

#include "proxy.h"

int main() {

// real store

ItemRequest our\_store\_front;

// this DOESN'T have Log inventory functionality

our\_store\_front.AddInventory(ItemForSale(123, 10, 1));

our\_store\_front.AddInventory(ItemForSale(456, 2, 2));

our\_store\_front.AddInventory(ItemForSale(543, 15, 1));

our\_store\_front.AddInventory(ItemForSale(233, 6, 1));

our\_store\_front.AddInventory(ItemForSale(434, 9, 1));

our\_store\_front.AddInventory(ItemForSale(665, 1, 1));

// proxy it

Proxy\_CheckBeforeRequest proxy\_store(&our\_store\_front);

// this DOES add to Log Inventory

proxy\_store.AddInventory(ItemForSale(277, 2, 2));

// client

PaymentType our\_payment = PaymentType::CREDIT;

// ultimately passes Request item to ItemRequest class

// since 234 doesn't exist in inventory, returns '0' because 234 isn't in inventory.

auto return\_code = proxy\_store.RequestItem(234, our\_payment);

std::cout << "return code for purchase 234: " << return\_code << '\n';

auto return\_code2 = proxy\_store.RequestItem(233, our\_payment);

std::cout << "return code for purchase 233: " << return\_code2 << '\n';

system("pause");}

# **Singleton**

## Overview

Singleton Pattern

* Singleton is a creational design pattern, which ensures that only one object of its kind exists and provides a single point of access to it for any other code.

Functional Design

* Make all the constructors of the class private.
* Delete the copy constructor of the class.
* Make a private static pointer that can point to the same class object (singleton class).
* Make a public static method that returns the pointer to the same class object (singleton class). ‘Lazy’

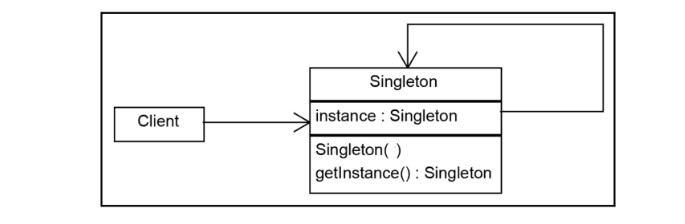
Proper:

* Instead of using a ptr and creating a new instance with the static public get() function, have the private member be a static object to the singleton class and the get function will return the object directly.
* Defining that static object in the file you are going to use it in automatically calls the class’s constructor, initializing all member variables including the private static Singleton object.

Why / When

* The Singleton Pattern guarantees that a class has only one instance throughout the application. This can be useful in scenarios in which multiple instances must be avoided, such as when managing shared resources or configurations.

## Diagrams



## Code Example

class Singleton

{

public:

// DON'T want a constructor, so people can't instantiate it outside of this class.

Singleton(const Singleton&) = delete;

// provides a way to access this class staticly.

// needs to return a reference or a pointer of this particular type.

static Singleton& Get()

{

return s\_Instance;

}

float Function() { return m\_Member; }

static float singletonFloat;

private:

float m\_Member = 0.0f;

// This member and the private constructor, allows this Singleton object/entity to exist.

Singleton() {}

// because it's static, it needs to be defined somewhere where's its being referenced, (the top of a cpp file).

// static members need to be defined in the file they are being applied to.

static Singleton s\_Instance; // this is NOT a Singleton member variable, because it's static

// it's a static variable that exists in the Singleton class namespace

// it's a static variable of Singleton type named s\_Instance.

// It is stored in a static variable because its scope has to be the class itself

// and not a particular instance

};

// Why doesn't this singleton pattern not require a definition of the statics in Singleton at top?

// STATIC MEMBER VARIABLES:

// If Static Member variables of a class/namespace need to be used outside of that class, they need to be

// defined (initialization not required), before the code block that needs to use them.

//

// To define a static member variable, you need their:

// variable type, Class/NameSpace, scope operator, static variable name.

// float Singleton::singletonFloat;

// It does not matter if the member variable is public or private, if it's a static member variable,

// it needs to be defined.

// Public Static variables of a namespace(.h included) can be accessed directly with scope operator,

// if defined before code block it's used. float Singleton::singletonFloat; Singleton::singletonFloat = 0.f;

// STATIC MEMBER FUNCTIONS:

// DO NOT need to be defined above code block they are used in, just need the header file included in file.

// Static member functions (public) can be accessed directly using scope operator. Singleton::Get();

// Private Static member functions don't need to be defined but are NOT available with scope operator.

// Singleton::Singleton() | Nope Singleton() constructor is a private member function.

// SINGLETONS, non lazy

// In Order to make a proper singleton:

// You need a private static Singleton member variable: m\_Instance;

// You need a private Singleton() constructor

// You need a public static function that returns a reference to m\_Instance. ::Get()

//

// MOST IMPORTANTLY, you need to define the m\_Instance at the top of the file you are using it in.

// Reason: the only way to use a single instance of a singleton pattern class, is to

// make a private static member variable of the same type and a public static Get() function to get it.

// Because, Get returns that private static member as a reference after being created inside the class,

// that private member variable needs to be defined before it can be accessed.

/\*

static Singleton& Get()

{

static Singleton \_instance;

return \_instance;

//return s\_Instance;

}\*/ // Works but breaks Singleton pattern.

// Singleton\_Pattern.cpp | https://www.youtube.com/watch?v=PPup1yeU45I

//

#include <iostream>

#include "Singleton.cpp"

#include "RandomNumber.cpp"

//#include "tea.h"

// s\_Instance is static, therefor, it needs to be defined somewhere where it's been used.

// v Singleton is the type of variable being defined!

// v this second Singleton is the namespace in which the s\_Instance static variable exists.

// this defines the static variable s\_Instance of a Singleton type that is part of the Singleton namespace.

Singleton Singleton::s\_Instance; // if we want to access static variable in a namespace, we need to tell the class using it, what type it is and what the namespace scope it's from.

// s\_Instance is static private in the Singleton:: scope and it's of type Singleton. The type itself forces the Singleton class to call it's default constructor.

// In calling the default Singleton constructor, all the member variable get initialized, which constructs the an instance of the singleton and assigns it to s\_Instance.

// s\_Instance is private, thus, not available, but Get() is public and is available to use

// Now that s\_Instance has been defined as a static member of type Singleton in the Singleton namespace, any public members or functions part of the Singleton namespace are also available.

float Singleton::singletonFloat; // public variable defined in singleton namespace.

//Tea Tea::m\_Tea; // even though m\_Tea is a private static member that already exists in tea.h,

// defining it in this file, declares/initializes the static member type automatically (as a global type variable).

// And, because m\_Tea is a class type variable (not a ptr), initializing m\_Tea means a constructor needs to be called for it

// to be a valid object in memory.

// Doesn't matter if this is private or public, or the constructor is private or public.

// Making a static member variable of a class type available in another file, means it's constructor is called to make it an object.

int main()

{

// this function call returns s\_Instance, which is why it needs to be defined up top.

// allowing this local reference of s\_Instance to be used indirectly as a static Singleton.

Singleton& instance = Singleton::Get();

Singleton::singletonFloat = 3.f;

//instance.Function();

system("pause");

}

# **Lazy Singleton**

## Overview

Notes from VS2022 sln file:

// SINGLETONS

// Single instance of a class. A single set of data. Don't need multiple instances.

// Singletons are a way to organize a set of global variables and static functions.

// Static functionality.

//

// WHEN to use Singleton?

// When we have functionality we want to use for a global set of data.

//

// Example: Random\_Number\_Generator() class{};

// Example2: Renderer() class{};

//

// Singletons can be used as NAMESPACES.

// In C++, Singletons do not need to be classes, they can be namespaces.

// In C++ (unlike Java), files do NOT need to be class files,

// C++ files can be just headers(.h) or source(.cpp) files with a set of data and/or functions not bound to a class.

//

// Wny do I need a class to make a Singleton?

// You don't. Singletons aren't classes, they are patterns.

// Singletons can be implemented as classes or

// Singletons can just be a .cpp or .h files with static functions and data global to all other classes who can see it.

// Singleton has it's own instance to be controlled by itself

// Single point of access to a manager class, no one else can make additional managers

// Singleton can very carefully control an inventory, the only way to add/remove is

// through this single manager instance, that call these specific functions.

// Making this static allows outside classes to get access to it,

// without needing to instantiate the Singleton externally.

// A static member function is independent of any object of the class.

// A static member function can be called even if no objects of the class exist.

// A static member function can also be accessed using the class name through

// the scope resolution operator.

// s\_Instance is static, therefor, it needs to be defined somewhere where it's been used.

// v Singleton is the type of variable being defined!

// v this second Singleton is the namespace in which the s\_Instance static variable exists.

// this defines the static variable s\_Instance of a Singleton type that is part of the Singleton namespace.

Singleton Singleton::s\_Instance; // if we want to access static variable in a namespace, we need to tell the class using it, what type it is and what the namespace scope it's from.

// s\_Instance is static private in the Singleton:: scope and it's of type Singleton. The type itself forces the Singleton class to call it's default constructor.

// In calling the default Singleton constructor, all the member variable get initialized, which constructs the an instance of the singleton and assigns it to s\_Instance.

// s\_Instance is private, thus, not available, but Get() is public and is available to use

// Now that s\_Instance has been defined as a static member of type Singleton in the Singleton namespace, any public members or functions part of the Singleton namespace are also available.

//Tea Tea::m\_Tea; // even though m\_Tea is a private static member that already exists in tea.h,

// defining it in this file, declares/initializes the static member type automatically (as a global type variable).

// And, because m\_Tea is a class type variable (not a ptr), initializing m\_Tea means a constructor needs to be called for it

// to be a valid object in memory.

// Doesn't matter if this is private or public, or the constructor is private or public.

// Making a static member variable of a class type available in another file, means it's constructor is called to make it an object.

// SINGLETONS

// Single instance of a class. A single set of data. Don't need multiple instances.

// Singletons are a way to organize a set of global variables and static functions.

// Static functionality.

//

// WHEN to use Singleton?

// When we have functionality we want to use for a global set of data.

//

// Example: Random\_Number\_Generator() class{};

// Example2: Renderer() class{};

//

// Singletons can be used as NAMESPACES.

// In C++, Singletons do not need to be classes, they can be namespaces.

// In C++ (unlike Java), files do NOT need to be class files,

// C++ files can be just headers(.h) or source(.cpp) files with a set of data and/or functions not bound to a class.

//

// Wny do I need a class to make a Singleton?

// You don't. Singletons aren't classes, they are patterns.

// Singletons can be implemented as classes or

// Singletons can just be a .cpp or .h files with static functions and data global to all other classes who can see it.

## Code Example

// tea.h

#include <string>

// Name goes after struct keyword.

struct Tea {

std::string name;

int cost;

int strength;

int quantity;

// default initializer typically good idea.

Tea(std::string n, int c, int s, int q) : name(n), cost(c), strength(s), quantity(q){}

};

#pragma once

#include <iostream>

#include <string>

#include <vector>

#include "tea.h"

// Singleton has it's own instance to be controlled by itself

// Single point of access to a manager class, no one else can make additional managers

// Singleton can very carefully control an inventory, the only way to add/remove is

// through this single manager instance, that call these specific functions.

// Making this static allows outside classes to get access to it,

// without needing to instantiate the Singleton externally.

class Singleton // When declaring a class, there are no '()' after the name, just '{}' brackets!!!

{

public:

static float singletonFloat;

// A static member function is independent of any object of the class.

// A static member function can be called even if no objects of the class exist.

// A static member function can also be accessed using the class name through

// the scope resolution operator.

static Singleton\* GetInstance() // 'Lazy' Initialization

{

// static makes it so, the Singleton instance isn't recreated everytime Init is called.

static Singleton\* instance = nullptr;

if (!instance)

instance = new Singleton();

return instance; // you could look at the memory and change the memory

}

void Add(Tea tea)

{

teas.push\_back(tea);

}

void RemoveTea(Tea tea)

{

std::vector<Tea>::iterator it = teas.begin();

for (it; it < teas.end(); it++)

{

if (tea.strength == 2)

{

teas.erase(it);

}

}

}

private:

Singleton() {}

std::vector<Tea> teas;

};

// Singleton\_CTT.cpp

// Single copy of your class, or a single instance'd object for a Singleton class.

// Master Manager, Renderer,

#include "singleton.h"

// Don't get access to Singleton::Get() function until we instantiate Singleton

// Singleton Singleton::m\_Instance;

float Singleton::singletonFloat;

int main()

{

Singleton::singletonFloat = 2.3f;

// You can hold references to Singleton, holding a reference to it doesn't make it a copy.

Singleton\* singleton = nullptr;

singleton = Singleton::GetInstance();

Singleton::singletonFloat = 2.1f;

Tea mytea("shit Tea", 1, 2, 3);

// even when mytea leaves scope, the Singleton will retain a copy of that Tea object.

// Singleton::GetInstance()->Add(mytea);

system("pause");

}

# **State**

## Overview

State Pattern

* This pattern encapsulates each state in a separate class, which makes it easier to add new states and modify existing states without altering the object’s code directly.
* This pattern is useful when an object transitions between different states, with each transition triggering specific actions.

Components

* **Context**: The class that has a state and assigns the state-specific behavior to different state objects.
* **State**: The abstract base class or interface that defines a common interface for all concrete states. Each concrete state implements specific behaviors associated with a particular state.
* **ConcreteState**: These are the classes that inherit from the State class and provide the specific implementations for the behaviors associated with a particular state.

Functional Design

* State Pattern - Base state - abstract class (at least one virtual member function).
* States do some sort of work. Behavioral pattern.
* The State's class it's interacting with (like the main game loop) will have a pointer to that class,

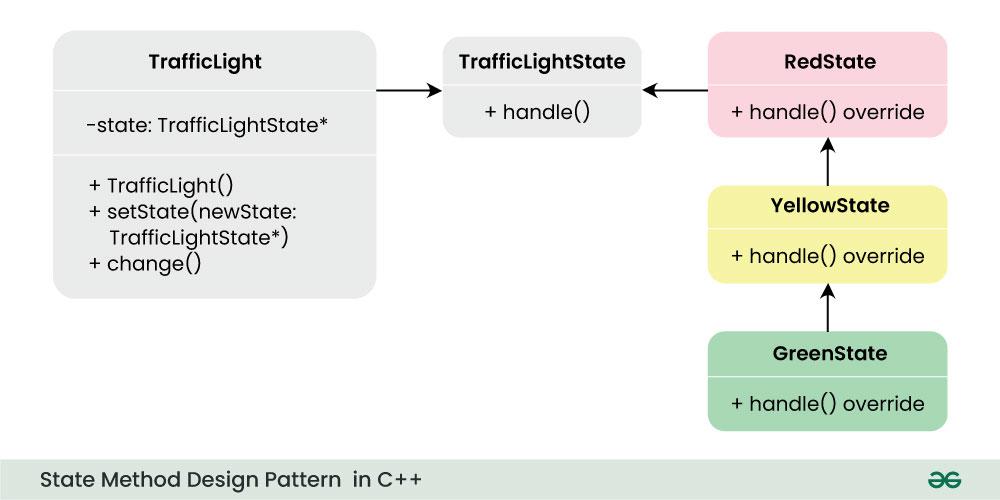
and that class will have a pointer to the state.

* During the State's partner class's (Engine) constructor is called, it runs a Transition to which takes in a state, the newly constructed Engine sets a member ptr to current state, and has the state call it's set engine. So, they both have pointers to each other.

Why / When

* Clean code (modular), flexibility, readability, scalable, testable (each state test independently).

## Diagrams



## Code Example

#define classname typeid(this).name()

class Engine;

class State {

protected:

// Not owned by the state, don’t ever delete it, in case we want to change state.

Engine\* \_engine;

public:

virtual ~State() {}

From the state, we want to call set\_engine, so that, it has a back reference to the engine. Because : when we run DoWorkA() or DoWorkB() later, sometimes they'll have a transition, that way, we'll be able to reference the engine, and call TransitionTo() to setup other States!

Set\_Engine is a method that's called in Engine transitionTo States function. After engine changes to a different state, it sets a reference back to engine.

void set\_engine(Engine\* engine) {

\_engine = engine;

}

virtual void DoWorkA() = 0;

virtual void DoWorkB() = 0;

};

class Engine {

private:

State\* \_state;

public:

Engine(State\* state) : \_state(nullptr) {

TransitionTo(state);

}

~Engine() { delete \_state; } // Engine has ownership of the states.

void TransitionTo(State\* state) {

if (\_state) delete \_state;

\_state = state;

\_state->set\_engine(this);

}

// Engine delegates some of it's function to the states here!

void RequestA() { \_state->DoWorkA(); }

void RequestB() { \_state->DoWorkB(); }

};

class ConcreteStateA : public State {

public:

void DoWorkA() override {

std::cout << classname << " called DoWorkA()\n"; }

void DoWorkB() override {

std::cout << classname << " called DoWorkB()\n";

}

};

class ConcreteStateB : public State {

public:

// change the context, ConcreteState B class DoWorkA(),

void DoWorkA() override {

std::cout << classname << " called DoWorkA()\n";

// transition to ConcreteState A,

\_engine->TransitionTo(new ConcreteStateA);

}

// states have protected member, engine pointer (defined in abstract base).

// flip all the states around that you need.

void DoWorkB() override

{

std::cout << classname << " called DoWorkB()\n";

}

};

// State.cpp | [**State Design Pattern YouTube**](https://www.youtube.com/watch?v=Y3FHNAYeamc&list=PLalVdRk2RC6otl3oBU2cn-P6DWi1y1PS3&index=16&t=6s)

// Designs some sort of situation, your program is in, and do some certain thing

// Initial engine (context), store a back reference in our states, so the states can call back to the engine to change states

// Need a class to handle states.

#include <iostream>

#include "state.h"

int main() {

Engine engine1(new ConcreteStateB);

engine1.RequestA();

engine1.RequestA();

system("pause");

}

// ; semicolons - create a class, call a function, prototype a function (but NOT define a function).

// why use .cpp files instead of just .h files?

# **Strategy**

## Overview

Strategy Pattern

* A behavioral design pattern that enables an algorithm to be selected at runtime.

Functional Design

* Defines a family of interchangeable algorithms and allows them to be used interchangeably.
* **Context**: The class that contains a reference to the strategy interface and is responsible for executing the algorithm.
* **Strategy Interface**: An interface or abstract class that declares the method(s) for the algorithm. Different strategies implement this interface.
* **Concrete Strategies**: The different algorithms that implement the strategy interface.

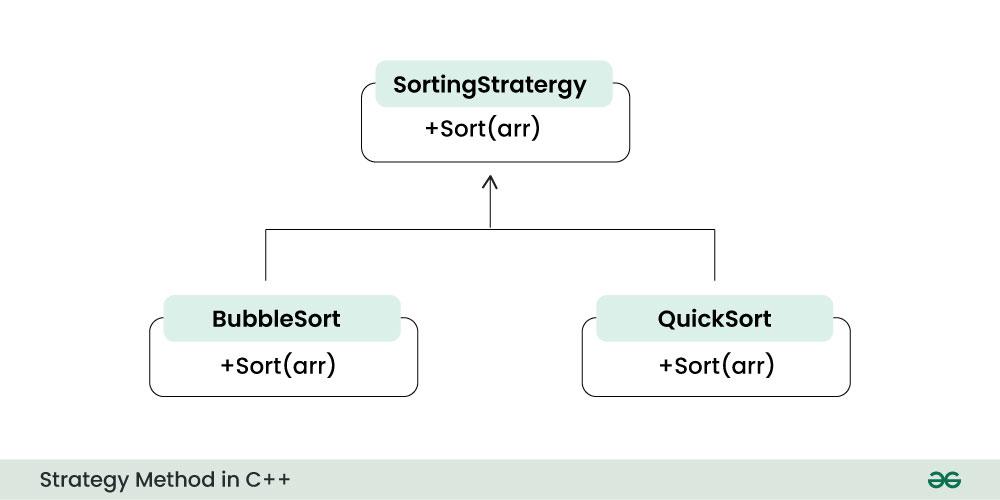
Why / When

* Flexibility: Easily switch between different algorithms at runtime.
* Code Reusability: Strategies can be reused across different contexts.
* Promotes Single Responsibility: Each strategy focuses on a specific algorithm.

Why not

* Increased number of classes.
* Inefficient use of exposing interface of all behaviors to classes not needing them.
* Application already configures the context with required strategy object, so app maintains two objects.

## Diagrams



## Code Example

// player.h

#include "strats.h"

// Player Class implements the strategy interface set the strategy to any concrete one we've implemented, calls execute on whatever the current strategy is set to.

// Player() can assignn strategy to a different strategy, and the next time execute is called, it applies the different strategy function.

class Player {

public:

Player(Strategy\* strat) {

level\_strat = strat;

}

void SetStrat(Strategy\* strat) {

level\_strat = strat;

}

int MeleeAttack(int base\_dmg, int stat\_bonus) {

return level\_strat->meleeattack(base\_dmg, stat\_bonus);

}

int RangedAttack(int base\_dmg, int stat\_bonus) {

return level\_strat->rangedattack(base\_dmg, stat\_bonus);

}

private:

Strategy\* level\_strat;

};

class Strategy {

public:

virtual ~Strategy() = default;

virtual int meleeattack(int base, int bonus) = 0;

virtual int rangedattack(int base, int bonus) = 0;

};

class Level1Strategy : public Strategy {

public:

int meleeattack(int base, int bonus) override {

return base + bonus;

}

int rangedattack(int base, int bonus) override {

return base + bonus;

}

};

class Level2Strategy : public Strategy {

public:

int meleeattack(int base, int bonus) override {

return base + bonus + 2;

}

int rangedattack(int base, int bonus) override {

return base + bonus + 2;

}

};

class Level3Strategy : public Strategy {

public:

int meleeattack(int base, int bonus) override {

return base + bonus + 4;

}

int rangedattack(int base, int bonus) override {

return base + bonus + 4;

}

};

// Strategy.cpp | [**Strategy Design Pattern YouTube**](https://www.youtube.com/watch?v=3I-C-Bu8pX4&list=PLalVdRk2RC6otl3oBU2cn-P6DWi1y1PS3&index=22)

#include "strats.h"

#include "player.h"

#include <iostream>

int main()

{

char a;

Level1Strategy l1;

Level2Strategy l2;

Level3Strategy l3;

Player player1(&l1);

std::cout << "Level 1 Melee Attack Damage: " << player1.MeleeAttack(1, 1);

std::cout << "Level 1 Ranged Attack Damage: " << player1.RangedAttack(1, 1);

std::cin >> a;

std::cout << "level up!" << '\n';

player1.SetStrat(&l2);

std::cout << "Level 2 Melee Attack Damage: " << player1.MeleeAttack(1, 1);

std::cout << "Level 2 Ranged Attack Damage: " << player1.RangedAttack(1, 1);

std::cin >> a;

std::cout << "level up!" << '\n';

player1.SetStrat(&l3);

std::cout << "Level 3 Melee Attack Damage: " << player1.MeleeAttack(1, 1);

std::cout << "Level 3 Ranged Attack Damage: " << player1.RangedAttack(1, 1);

system("pause");

}

# **Template**

## Overview

Template Pattern

* The Template Method design pattern is a behavioral design pattern that defines the skeleton of an algorithm in a superclass but allows subclasses to override specific steps of the algorithm without changing its structure.

Functional Design

* The overall structure and sequence of the algorithm are preserved by the parent class.

Components

* Abstract Class (or interface) – Superclass defines the template method.
* Template Method - the method within the abstract class that defines the overall algorithm structure by calling various steps in a specific order.
* Abstract (hook) Method – Placeholders steps in the algorithm to be implemented by subclasses
* Concrete Subclasses – classes that provide concrete implementation but don’t change overall structure.

Why / When

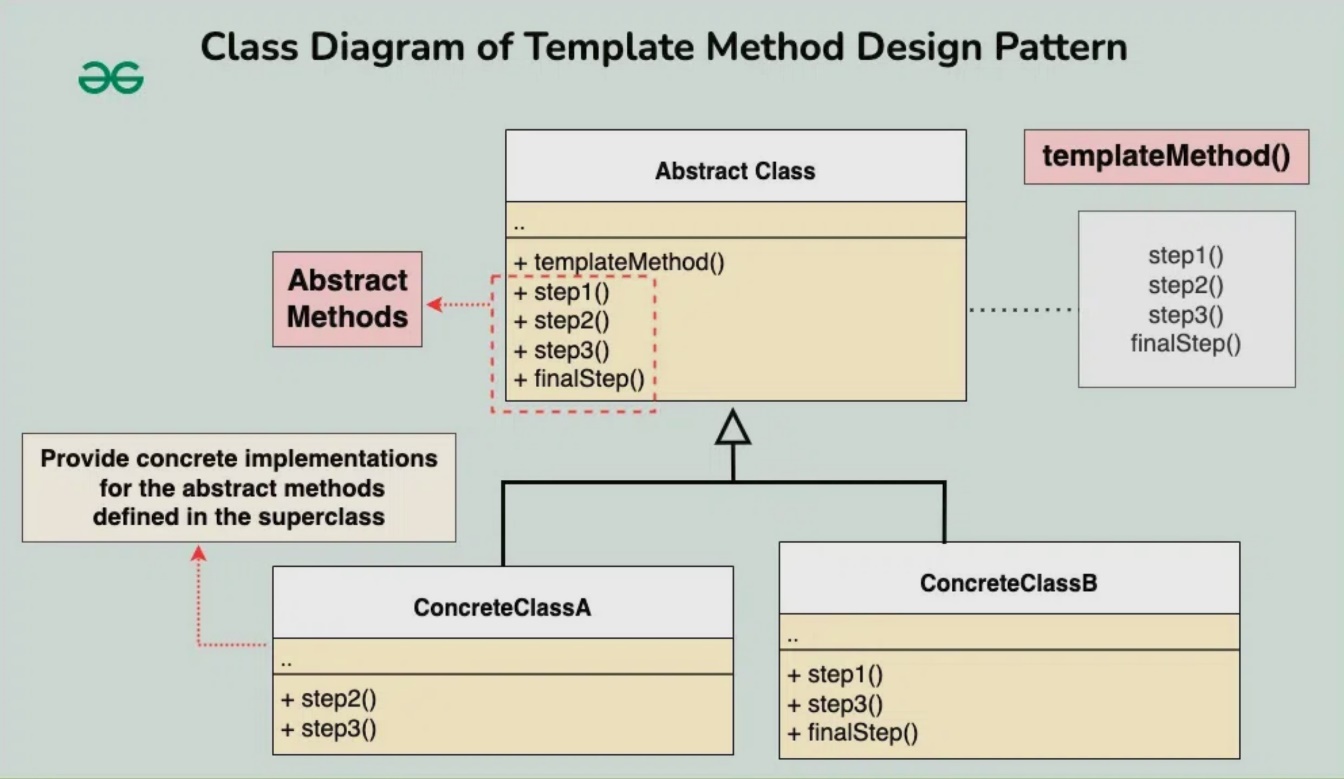
* Code reuse, reduces duplication, and provides a flexible way to accommodate variations.

Why not

* Highly variable algorithms, tight coupling for algorithms you will be using, inflexible runtime changes, and overhead cost for abstraction / inheritance.

## Diagrams





## Code Example

// The abstract class defines a template method that contains a skeleton of some

// algorithm, composed of calls to abstract primitive operations.

// Template pattern uses a interface to define functionality, a base algorithm

class IAbstractAI {

public:

void BaseOperatrion() { // publicly run all your code, all your templates.

ScanMap();

MoveUnit();

Attack();

}

protected:

// operations you have to go through as part of construction.

void ScanMap() {

//std::cout << "base ai - scan map\n";

Scanner();

}

void MoveUnit() {

//std::cout << "base ai - move unit\n";

Mover();

}

void Attack() {

//std::cout << "base ai - attack\n";

SpecialAttack();

}

virtual void Scanner() = 0;

virtual void Mover() = 0;

virtual void SpecialAttack() = 0;

virtual void Special1() {}

virtual void Special2() {}

};

// Concrete subclasses should implement these operations, but leave the template method intact.

class RangedAlly : public IAbstractAI {

public:

protected:

// Inherited via IAbstractAI

virtual void Scanner() override {

std::cout << "RangedAlly SCANNER\n";

EnemyInSight = (bool)ZOOR();

if (EnemyInSight) {

std::cout << " - Finds enemy in sight...\n";

}

else {

std::cout << " - no enemy in sight...\n";

}

}

virtual void Mover() override {

if (EnemyInSight) {

std::cout << " - moves to attack range\n";

EnemyInRange = true;

}

}

virtual void SpecialAttack() override {

if (EnemyInSight && EnemyInRange) {

std::cout << " - fires weapon\n";

Special1();

Special2();

}

}

// attack with possible crit chance special

virtual void Special1() override {

if (amountcount < 1) {

Reload();

}

else {

auto roll = NTKR(1, 100);

if (roll > 67) {

std::cout << " - crits!\n";

}

else if (roll > 33) {

std::cout << " - hits!\n";

}

else {

std::cout << " - misses!\n";

}

}

}

private:

bool EnemyInSight = false;

bool EnemyInRange = false;

int amountcount = 4;

void Reload() { amountcount = 4; }

};

// Number generator class

float ZTORf()

{

// initialize the random number generator with time-dependent seed

static uint64\_t timeSeed = std::chrono::high\_resolution\_clock::now().time\_since\_epoch().count();

static std::seed\_seq ss{ uint32\_t(timeSeed & 0xffffffff), uint32\_t(timeSeed >> 32) };

static std::mt19937 mgen(ss);

// initialize a uniform distribution between 0 and 1

static std::uniform\_real\_distribution<float> unif(0, 1);

return unif(mgen);

}

double ZTOR()

{

// initialize the random number generator with time-dependent seed

static uint64\_t timeSeed = std::chrono::high\_resolution\_clock::now().time\_since\_epoch().count();

static std::seed\_seq ss{ uint32\_t(timeSeed & 0xffffffff), uint32\_t(timeSeed >> 32) };

static std::mt19937 mgen(ss);

// initialize a uniform distribution between 0 and 1

static std::uniform\_real\_distribution<double> unif(0, 1);

return unif(mgen);

}

int ZOOR()

{

static uint64\_t timeSeed = std::chrono::high\_resolution\_clock::now().time\_since\_epoch().count();

static std::seed\_seq ss{ uint32\_t(timeSeed & 0xffffffff), uint32\_t(timeSeed >> 32) };

static std::mt19937 mgen(ss);

// initialize a distribution of 0 or 1 randomer

static std::uniform\_int\_distribution<std::mt19937::result\_type> dist2(0, 1);

return dist2(mgen);

}

int NTKR(int n, int k)

{

static uint64\_t timeSeed = std::chrono::high\_resolution\_clock::now().time\_since\_epoch().count();

static std::seed\_seq ss{ uint32\_t(timeSeed & 0xffffffff), uint32\_t(timeSeed >> 32) };

static std::mt19937 mgen(ss);

// check if user put n <= k to process bounds

if (n <= k) {

std::uniform\_int\_distribution<std::mt19937::result\_type> ntkd(n, k);

return ntkd(mgen);

}

else // else n > k, still functions as bounds k to n

{

std::uniform\_int\_distribution<std::mt19937::result\_type> ntkd(k, n);

return ntkd(mgen);

}

}

float NTKR(float n, float k)

{

static uint64\_t timeSeed = std::chrono::high\_resolution\_clock::now().time\_since\_epoch().count();

static std::seed\_seq ss{ uint32\_t(timeSeed & 0xffffffff), uint32\_t(timeSeed >> 32) };

static std::mt19937 mgen(ss);

// check if user put n <= k to process bounds

if (n <= k)

{

std::uniform\_real\_distribution<float> ntkd(n, k);

return ntkd(mgen);

}

else // else n > k, still functions as bounds k to n

{

std::uniform\_real\_distribution<float> ntkd(k, n);

return ntkd(mgen);

}

}

double NTKR(double n, double k)

{

static uint64\_t timeSeed = std::chrono::high\_resolution\_clock::now().time\_since\_epoch().count();

static std::seed\_seq ss{ uint32\_t(timeSeed & 0xffffffff), uint32\_t(timeSeed >> 32) };

static std::mt19937 mgen(ss);

// check if user put n <= k to process bounds

if (n <= k)

{

std::uniform\_real\_distribution<double> ntkd(n, k);

return ntkd(mgen);

}

else // else n > k, still functions as bounds k to n

{

std::uniform\_real\_distribution<double> ntkd(k, n);

return ntkd(mgen);

}

}

// Template.cpp | [**Template Pattern example**](https://www.youtube.com/watch?v=2Cu86MnAFlc&list=PLalVdRk2RC6otl3oBU2cn-P6DWi1y1PS3&index=18)

// Method pattern that defines base stuff and some required stuff that any other class might want to use it has to implement it.

// Framework extending behavior using inheritance.

// This example uses AI theory to build something in a game. It's common in game code.

#include <iostream>

#include "rand.h"

#include "AI.h"

// testing and client down here

int main()

{

RangedAlly ranged\_ally;

ranged\_ally.BaseOperatrion();

}

# **Visitor**

## Overview

Visitor Pattern

* A behavioral design pattern that perform an operation on a group of similar kind of Objects.

Functional Design

* The point is, we add functionality to the visitors instead of the components.
* Components call functions that pass in the visitor which in turn calls visitor functions that pass in itself giving the visitor access to the component to perform behaviors on them.
* Thanks to polymorphism the component function that passes in the visitor pointer doesn't have to specify which concrete visitor class it is.

Components

* **Client** : The Client class is a consumer of the classes of the visitor design pattern. It has access to the data structure objects and can instruct them to accept a Visitor to perform the appropriate processing.
* **Visitor** : This is an interface or an abstract class used to declare the visit operations for all the types of visitable classes.
* **ConcreteVisitor** : For each type of visitor all the visit methods, declared in abstract visitor, must be implemented. Each Visitor will be responsible for different operations.
* **Visitable** : This is an interface which declares the accept operation. This is the entry point which enables an object to be “visited” by the visitor object.
* **ConcreteVisitable** : These classes implement the Visitable interface or class and defines the accept operation. The visitor object is passed to this object using the accept operation.

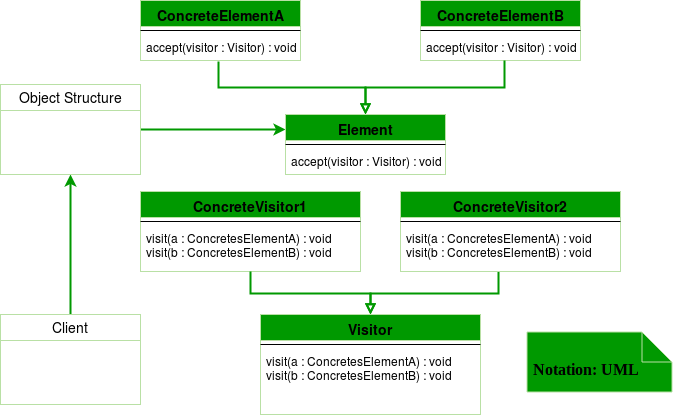
Why / When

* If the logic of operation changes, then we need to make change only in the visitor implementation rather than doing it in all the item classes.
* Adding a new item to the system is easy, it will require change only in visitor interface and implementation and existing item classes will not be affected.

Why not

* We should know the return type of visit() methods at the time of designing otherwise we will have to change the interface and all of its implementations.
* If there are too many implementations of visitor interface, it makes it hard to extend.

## Diagrams



## Code Example

// Visitor pattern is a behavioral pattern decouples functionality from your components.

#include <string>

#include <iostream>

class ComponentA;

class ComponentB;

class ComponentC;

// things to operate on, visitor needs to know about them.

// This is the visitor class to apply pattern.

class Visitor {

public:

// these will called something more in the line of what they do.

// they are const because they aren't going to change anything, they are only going to visit things and run functions.

// only public

// Visitor interface needs to define public functions passing the type of components it has access to.

virtual void VisitComponentA(const ComponentA\* comp) const = 0;

virtual void VisitComponentB(const ComponentB\* comp) const = 0;

virtual void VisitComponentC(const ComponentC\* comp) const = 0;

};

// Base component class that's an interface

class Component {

public:

virtual ~Component() {}

// accept a visitor ptr. Needs to be pointer because the visitor is a pure virtual class.

virtual void Accept(Visitor\* visitor) const = 0;

};

// Don't need to edit components, hence the visitor pattern.e

class ComponentA : public Component {

public:

virtual void Accept(Visitor\* visitor) const override {

visitor->VisitComponentA(this); // passes this component to the visitor.

}

// the visitor now has access to these component functions.

std::string ComponentAFunc() const {

return "ComponentA";

}

};

class ComponentB : public Component {

public:

virtual void Accept(Visitor\* visitor) const override {

visitor->VisitComponentB(this);

}

std::string ComponentBFunc() const {

return "ComponentB";

}

};

class ComponentC : public Component {

public:

virtual void Accept(Visitor\* visitor) const override {

visitor->VisitComponentC(this);

}

std::string ComponentCFunc() const {

return "ComponentC";

}

};

// Visitor concrete classes would have specific functionality and would be called something

// analogous to what that functionality is. Instead of 'Visitor1' it would be 'VisitorPrint' for example.

#include <iostream>

class Visitor1 : public Visitor {

public:

void VisitComponentA(const ComponentA\* comp) const {

std::cout << "visited" << comp->ComponentAFunc() << "from Visitor1\n";

}

void VisitComponentB(const ComponentB\* comp) const {

std::cout << "visited" << comp->ComponentBFunc() << "from Visitor1\n";

}

void VisitComponentC(const ComponentC\* comp) const {

std::cout << "visited" << comp->ComponentCFunc() << "from Visitor1\n";

}

};

class Visitor2 : public Visitor {

public:

void VisitComponentA(const ComponentA\* comp) const {

std::cout << "visited" << comp->ComponentAFunc() << "from Visitor2\n";

}

void VisitComponentB(const ComponentB\* comp) const {

std::cout << "visited" << comp->ComponentBFunc() << "from Visitor2\n";

}

void VisitComponentC(const ComponentC\* comp) const {

std::cout << "visited" << comp->ComponentCFunc() << "from Visitor2\n";

}

};

// Visitor.cpp | [**Visistor Design Pattern YouTube**](https://www.youtube.com/watch?v=IobXgpgqnYc&list=PLalVdRk2RC6otl3oBU2cn-P6DWi1y1PS3&index=12)

#include "visitor.h"

#include <array>

int main() {

std::array<const Component\*, 3> components = { new ComponentA, new ComponentB, new ComponentC };

Visitor1\* v1 = new Visitor1;

Visitor2\* v2 = new Visitor2;

for (const Component\* comp : components) {

comp->Accept(v1);

}

for (const Component\* comp : components) {

comp->Accept(v2);

}

// cleanup

for (const Component\* comp : components) {

delete comp;

}

delete v1;

delete v2;

system("pause");

}

**MISC:**

Design Patterns

3 Types: Creational, Behavioral, and Structural

Singleton - gatekeeper of shared resources, created by invoking a static method exposed on the class. Sometimes considered an anti-pattern because they simulate the use of global variables. Better than static methods because of Singletons are objects (inheritance/interface can be used), possible multiplicity later if you decide to change your code, dynamic binding at runtime. Singletons can be slow, holds onto resources too long, and must be synchronized in multithreaded environments.

Strategy pattern is a behavioral software design pattern that enables selecting an algorithm at runtime. Allows the code to be more flexible and reusable. A strategy pattern might select the type of validation algorithm depending on the type of data, source of data, or user choice. The separate algorithms or strategies are encapsulated separately from the validating object. This allows better decoupling between the behavior and the class that uses the behavior. The behavior can be changed without breaking the classes that use it, and the classes can switch between behaviors by changing the specific implementation used without requiring any significant code changes.

Builders - software pattern that creates objects in a stepwise manner without caring how those objects are constructed. Instead of constructing object directly, instantiate a builder object and let it create multiple objects on your behalf. This is particularly useful when objects have multiple constructor parameters.

Iterator - software pattern that enables you to traverse through all elements of a data structure.

Observer - Observers listen to state changes from its subject. Observers registers with the subject using a common interface for update notifications. A listener.

Decorator - software pattern that modifies the behavior of an object by wrapping with another class that is derived from the same base class and thus has same set of methods as the original object.

Decorator is generally made up of four classes: Component, Concrete Component, Decorator, and Concrete Decorator. Component is the base abstract class or interface that defines all the public methods needed for the underlying object and decorators that wrap it. Decorator is another abstract class that provides functionality shared by all decorators. The decorator itself wraps the Concrete Component and forwards the method calls to the Compoment. The Concrete Component is a specific child of the Component and the object by which the whole decorator pattern is used through. The Concrete Decorators (usually several), modify the behavior of the wrapped Concrete Component by overriding the methods of their parent Decorator class.

Bibliography

<https://www.geeksforgeeks.org/>