

ACCU
2023

DESIGN PATTERNS:

EXAMPLES IN C++

CHRIS RYAN



Design Patterns



ACCU 2023 - Bristol, England

Chris Ryan

[linkedin.com/in/chrisr98008](https://www.linkedin.com/in/chrisr98008)

Design Patterns

SECRET CODE FOR OFFICIAL CLUB MEMBERS ONLY!

A	∟	G	∞	M	∟	S	//	Y	▽
B	∕	H	∟	N	+	T	>	Z	++
C	△	I	→	O	•	U	↑	SUPERMAN	••
D	∨	J	7	P	↓	V	∪	TIM	◁•
E	•	K	∩	Q	S	W	ε	•	⊙
F	⊕	L	□	R	=	X	⊥		

Design Patterns

What are Design Patterns?

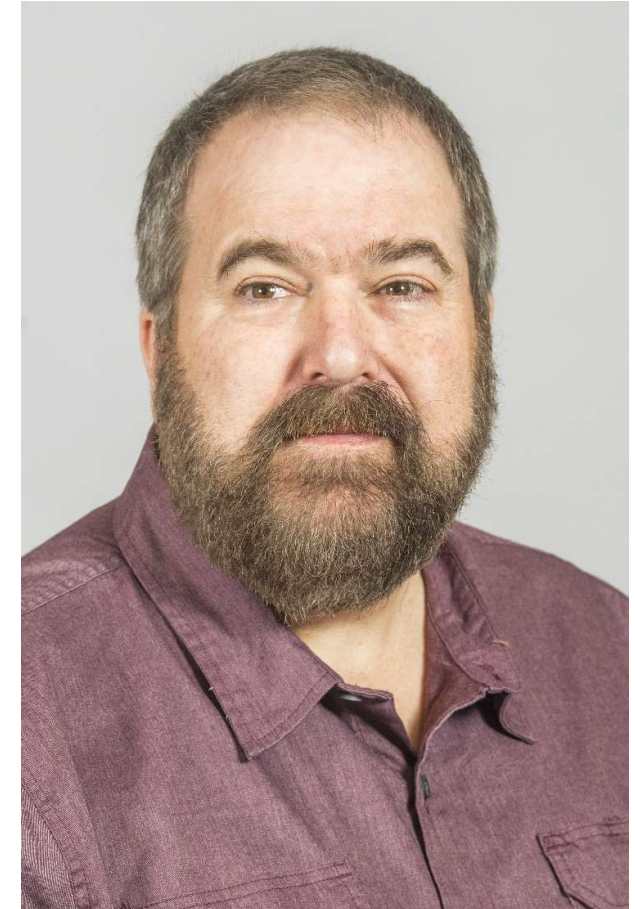
- They are a repeatable solution to commonly occurring problems,
- ...not architecture, a design, and not code,
- ...general descriptions of general solutions to general scenarios,
- ...common vocabulary for quickly explaining ideas or design,
- ...quickly recognizable in code,

- Design Patterns are not only for C++ or OOP.
 - They work in many languages and in many paradigms.

Design Patterns

Who am I?

- Chris Ryan, Seattle Metro area, Washington State, U.S.A.
- Classically trained in Software and Hardware engineering,
- Specialize in Modern C++,
- Worked in complex problem spaces,
 - Believe in simplification and reducing complexity,
- Projects of many scales, extremely large and small,
 - Including Firmware & Embedded,
- Recently joined the ISO C++ Standards Committee,
 - Work with the Evolution Working Group (EWG),
- Have no interest in C#/.Net, Java, js or web-ish tech.



Design Patterns

Practice: Design patterns can speed up the development process by providing tested, proven development paradigms.

Criticism: Misuse and Abuse leads to Anti-Patterns.

Design patterns may just be a sign that some features are missing in a given programming language.

Design Patterns

History

- **Personal Toolbox**
- **A Pattern Language:** (1977) Towns, Buildings, Construction.
 - Christopher Alexander (Architect)
- **Design Patterns:** (1994) Elements of Reusable Object-Oriented Software,
 - Gamma, Vlissides, Johnson, and Helm (aka “The Gang of Four” / “GoF”)
- **Code Complete:** (1993 / 2004)
 - Steve McConnell
- **Pattern-Oriented Software Architecture (POSA)**
 - 5 Volumes: (1996, 2000, 2004, 2007, 2007)
 - Multiple authors

Architecture, Patterns & Idioms

Design Level

Software Architecture

Architecture patterns
(client-server, microservices, frameworks, ...)

Software Design

Design patterns
(Visitor, Strategy, Observer, Polymorphic, ...)

Idioms

Design patterns (Pimpl, non-virtual, ...)
Implementation patterns (RAII, temp-swap,...)

Code Level

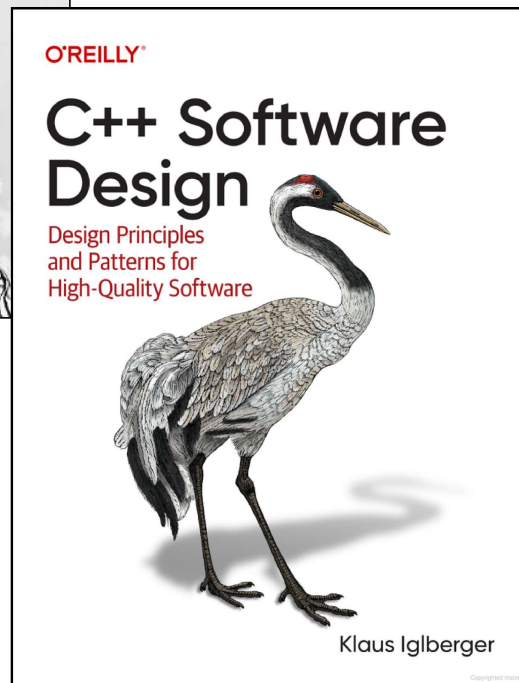
Implementation Details

Implementation patterns
(std::..., make_shared<>, enable_if, class,...)

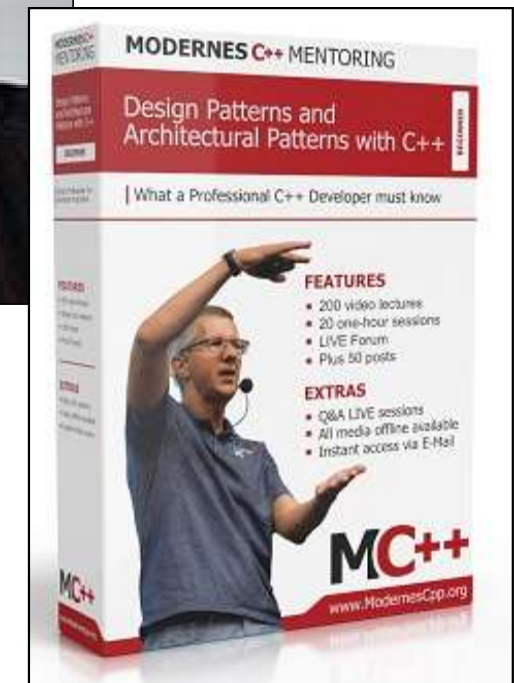
Image "borrowed" from "C++ Software Design" Klaus Iglberger

Design Patterns – Training, Books, Speakers

Klaus Iglberger



Rainer Grimm



Design Patterns

Pseudo Code / Slideware

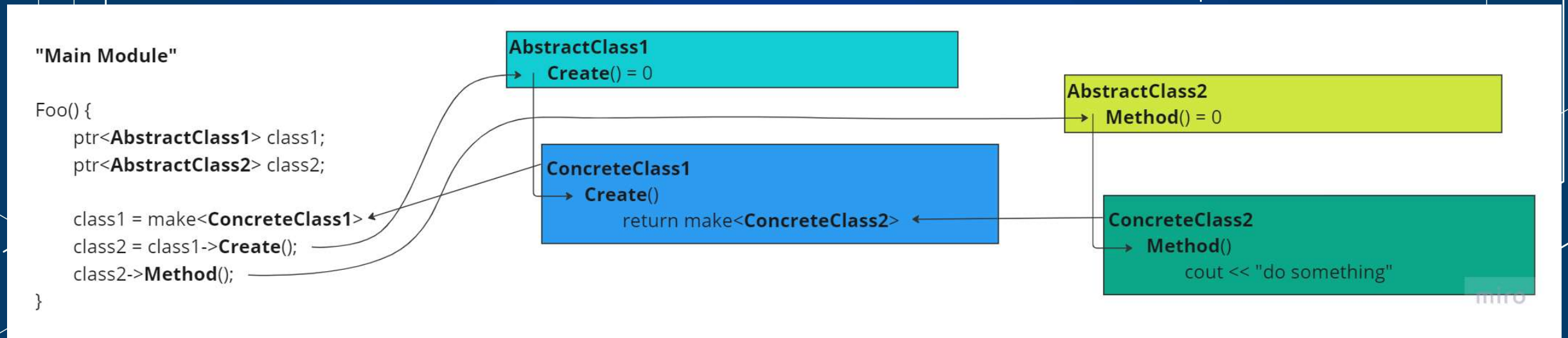
```
ptr<Factory> factory;           // smart pointer
factory = make<Factory>();      // constructs object & smart pointer
factory->Method(...);
```

Godbolt sample links are over-simplified, missing best use of const, override, reference params &, and move params &&

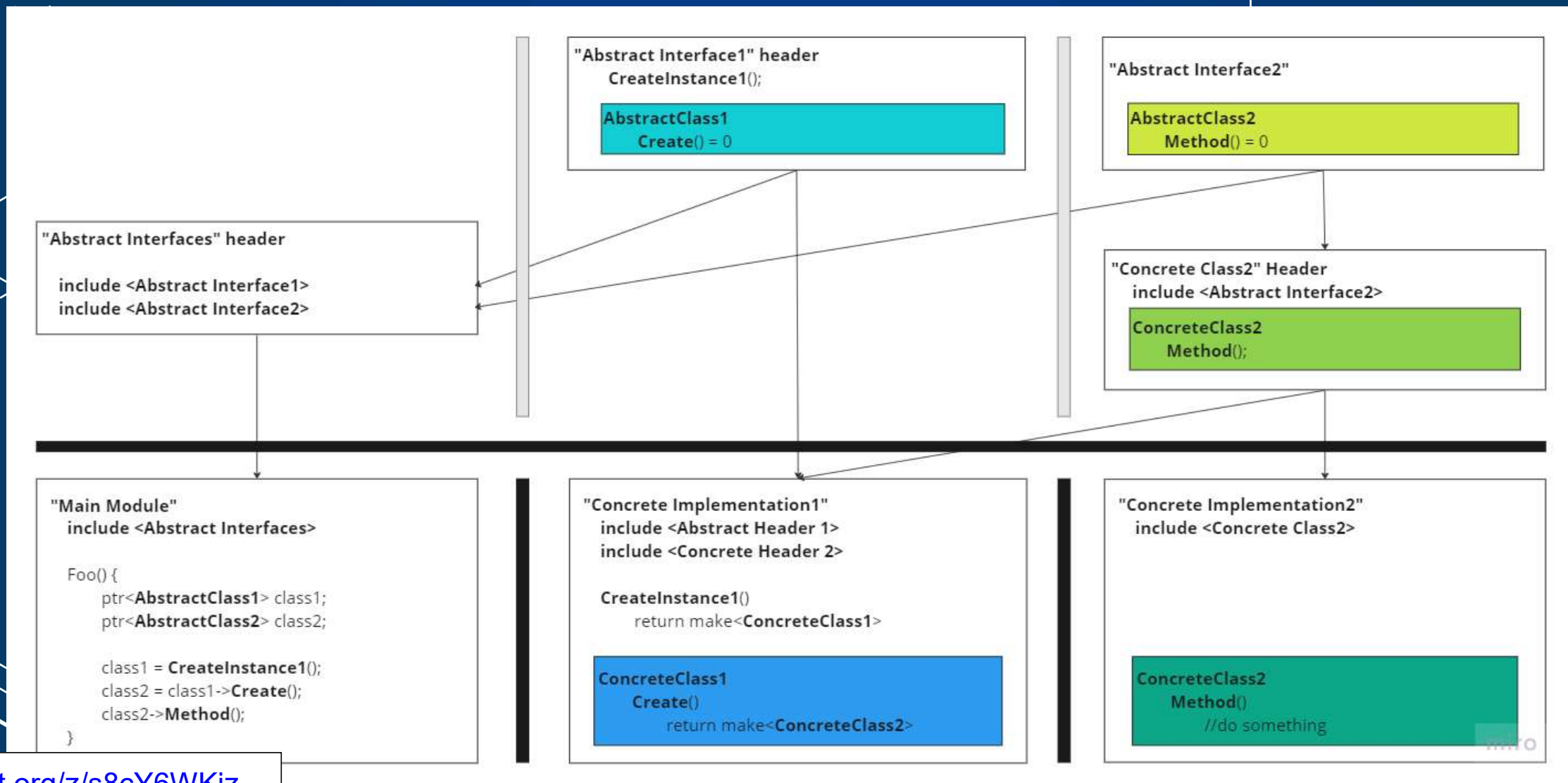
You Don't need the latest language features and gadgets.

Most samples are C++11 Modern compliant.

Design Patterns



Design Patterns



Design Patterns

```
#include <iostream>
#include <memory>

// Abstract Classes
struct AbstractClass2 { virtual void Method() = 0; };
struct AbstractClass1 { virtual std::shared_ptr<AbstractClass2> Create() = 0; };

// Concrete Classes
struct ConcreteClass2 : AbstractClass2 {
    void Method() { std::cout << "do something.\n"; };
};
struct ConcreteClass1 : AbstractClass1 {
    std::shared_ptr<AbstractClass2> Create() { return std::make_shared<ConcreteClass2>(); };
};

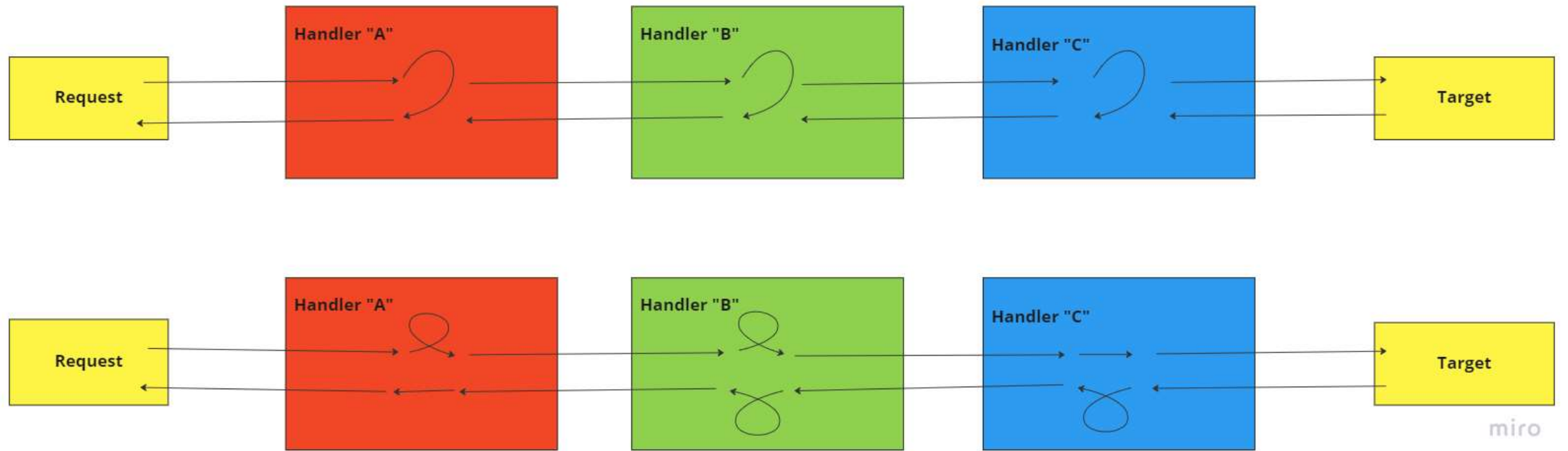
int main() {
    std::shared_ptr<AbstractClass1> class1 = std::make_shared<ConcreteClass1>();
    std::shared_ptr<AbstractClass2> class2 = class1->Create();

    class2->Method();
}
```

Design Patterns

Daisy Chain Filtering: Intercepting vs. Modifying

Daisy Chain Filtering: Intercepting vs. Modifying



Design Patterns-Dependency Injection

Polymorphic

```
int CopyAll(Source& src, Source& dest) {  
    int count{};  
    byte buffer[255];  
    while (int read = src.Read(buffer, sizeof(buffer)))  
    {  
        dest.Write(buffer, read);  
        count += read;  
    }  
    return count;  
}
```

```
int main() {  
    File file;  
    Open(file);  
  
    Stream stream;  
    Open(stream);  
  
    CopyAll(file, stream);  
    file.Seek(777);  
    stream.Seek(222);  
    CopyAll(stream, file);  
  
    file.Close();  
    stream.Close();  
}
```

```
struct Source  
{  
    virtual int Read(byte* buffer, int count) = 0;  
    virtual int Write(byte* buffer, int count) = 0;  
    virtual int Seek(int pos) = 0;  
    virtual void Close() = 0;  
};
```

```
struct File  
{  
    File() { ... };  
    friend Open(File&) { ... }  
    int Read(byte* buffer, int count) { ... }  
    int Write(byte* buffer, int count) { ... }  
    int Seek(int pos) { ... }  
    void Close() { ... }  
};
```

```
struct Stream  
{  
    File() { ... }  
    friend Open(File&) { ... }  
    int Read(byte* buffer, int count) { ... }  
    int Write(byte* buffer, int count) { ... }  
    int Seek(int pos) { ... }  
    void Close() { ... }  
};
```

Templatized

```
template<typename Src, typename Dest>  
int CopyAll(Src& src, Dest& dest) {  
    int count{};  
    byte buffer[255];  
    while (int read = src.Read(buffer, sizeof(buffer)))  
    {  
        dest.Write(buffer, read);  
        count += read;  
    }  
    return count;  
}
```

```
int main() {  
    File file;  
    Open(file);  
  
    Stream stream;  
    Open(stream);  
  
    CopyAll(file, stream);  
    file.Seek(777);  
    stream.Seek(222);  
    CopyAll(stream, file);  
  
    file.Close();  
    stream.Close();  
}
```

```
struct File  
{  
    File() { ... };  
    friend Open(File&) { ... };  
    int Read(byte* buffer, int count) { ... };  
    int Write(byte* buffer, int count) { ... };  
    int Seek(int pos) { ... };  
    void Close() { ... };  
};
```

```
struct Stream  
{  
    File() { ... };  
    friend Open(File&) { ... };  
    int Read(byte* buffer, int count) { ... };  
    int Write(byte* buffer, int count) { ... };  
    int Seek(int pos) { ... };  
    void Close() { ... };  
};
```

miro

Design Patterns

Classification of Software Design Patterns

- Creational Patterns
- Structural Patterns
- Behavioral Patterns
- **Concurrency Patterns**

Design Patterns

Concurrency Patterns

- Sharing
- Mutation
- Concurrent Architectures

Concurrency

- Processes & Threading (Simultaneous / Procedural)
- Synchronization (Data Protection)
- Inter-Process Communication

Design Patterns

Creational

- Abstract
- Builder
- Factory
- Prototype
- Singleton

Structural

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Behavioral

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

Design Patterns-Creational

- Abstract - Groups object factories that have a common theme.
- Builder - Constructs objects by separating construction and representation.
- Factory - Method creates objects without specifying the exact class to create.
- Prototype - Creates objects by cloning an existing object.
- Singleton - Restricts object creation to only one instance.

Design Patterns-Structural

- Adapter - An intermediary translating interface.
- Bridge - Decouples an abstraction from its implementation.
- Composite - Dynamic collection of objects manipulated as one object.
- Decorator - Daisy chain of filters extending interface behavior.
- Façade - Provides a simplified interface to a larger body of code.
- Flyweight - Reduces the cost of creating and manipulating of similar objects.
- Proxy - A placeholder for another object to control access, reduce complexity

Design Patterns-Behavioral

- Chain of Responsibility - Delegates commands through cascade processing.
- Command - Batching of multiple actions and parameters until executed.
- Interpreter - Implements a specialized language.
- Iterator - Access to the elements of an object hiding its implementation.
- Mediator - Like the Observer pattern but excludes the notifier sender.
- Memento - Provides the ability to restore an object state (undo).
- Observer - A publish/subscribe, all observer objects see event notifications.
- State - An object to alter its behavior when its internal state changes.
- Strategy - Allows switchable algorithms on-the-fly at runtime.
- Template Method - Extends skeleton functionality by providing concrete behavior.
- Visitor - Separation of responsibility applied to an object hierarchy.

Design Patterns-Creational



Design Patterns-Creational

- Abstract - Groups object factories that have a common theme.
- Builder - Constructs objects by separating construction and representation.
- Factory - Method creates objects without specifying the exact class to create.
- Prototype - Creates objects by cloning an existing object.
- Singleton - Restricts object creation to only one instance.

Design Patterns-Creational

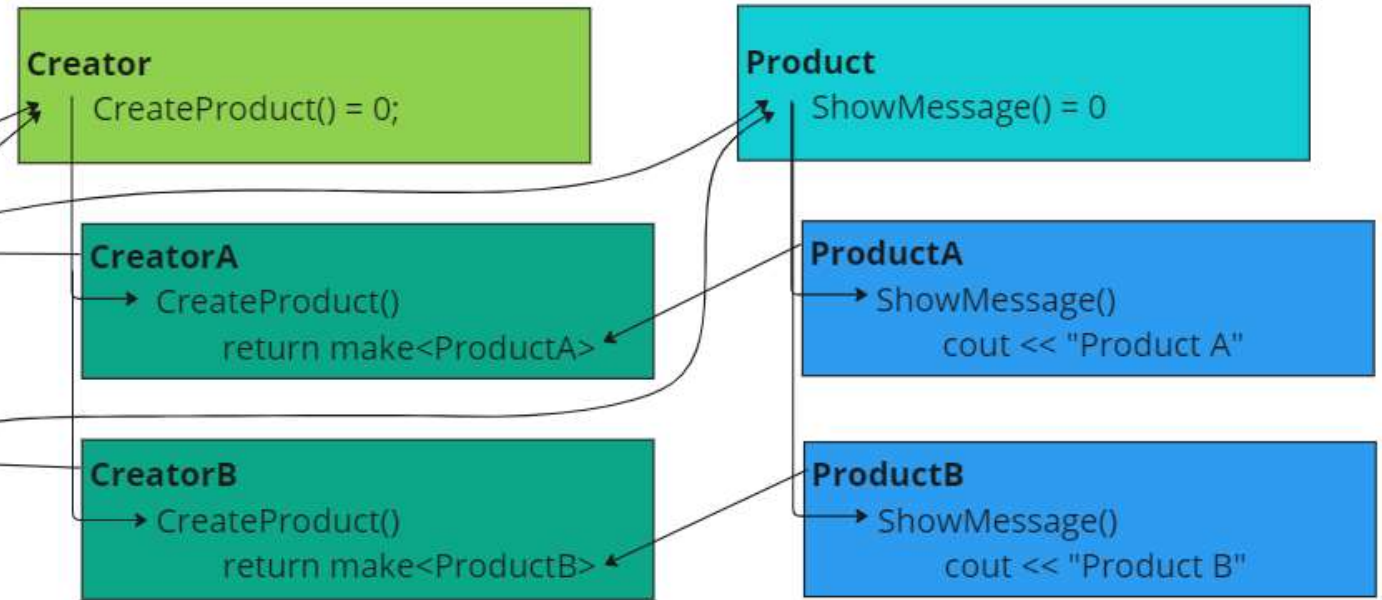
Focused on the process of object creation

- Flexibility
- Maintainability
- Reusability
- Scalability
- Reduce code duplication
- Performance

Design Patterns-Creational-Factory

Method creates objects without specifying the exact class to create

```
Foo() {  
  ptr<Creator> creator;  
  ptr<Product> product;  
  
  creator = make<ConcreteCreatorA>();  
  product = creator->CreateProduct();  
  product->ShowMessage();  
  
  creator = make<ConcreteCreatorB>();  
  product = creator->CreateProduct();  
  product->ShowMessage();  
}
```



```
output:  
Product A  
Product B
```

Design Pattern – Creational - Factory

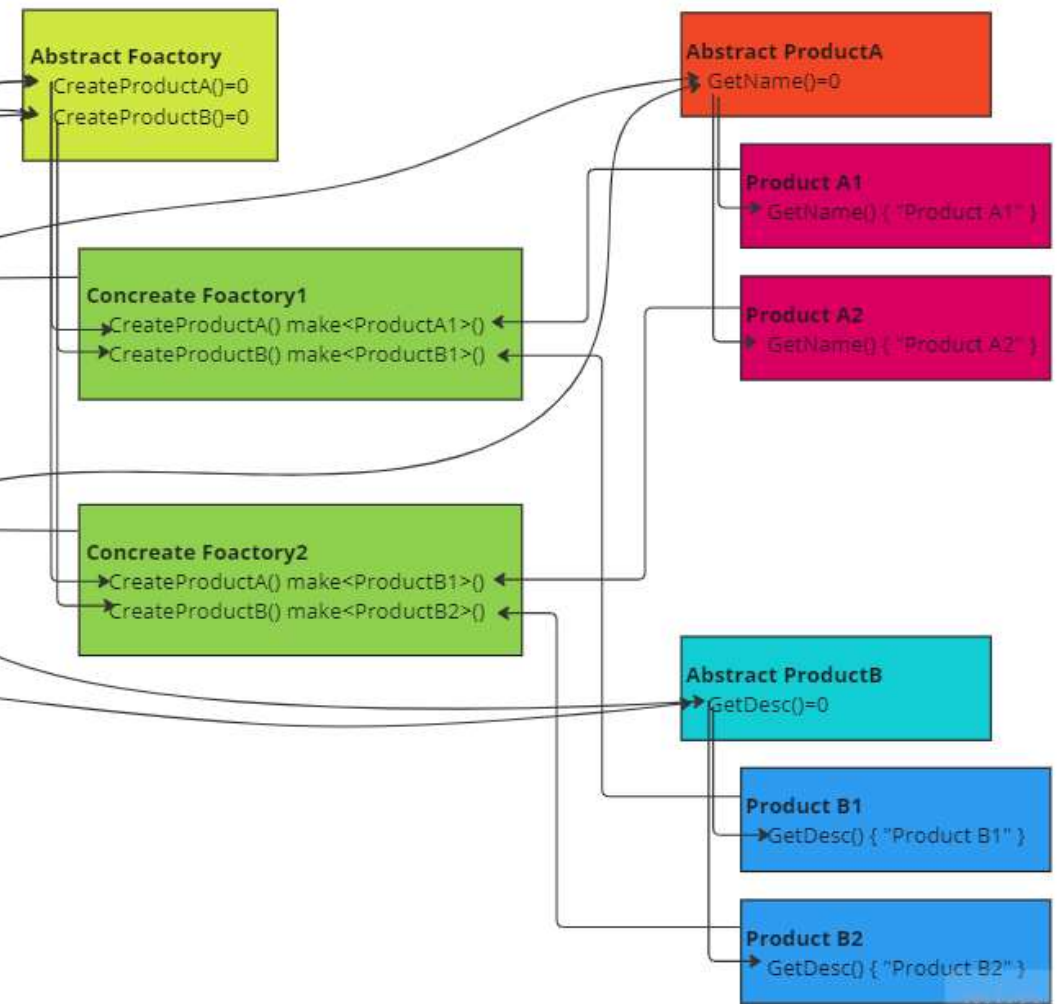
miro

Design Patterns-Creational-Abstract Factory

Groups object factories that have a common theme

```
Foo() {  
    ptr<AbstractFactory> factory;  
    ptr<AbstractProductA> productA;  
    ptr<AbstractProductB> productB;  
  
    factory = make<Factory1>();  
    productA = factory->CreateProductA();  
    productB = factory->CreateProductB();  
    cout << productA->GetName() << " / " << productB->GetDesc();  
  
    factory = make<Factory2>();  
    productA = factory->CreateProductA();  
    productB = factory->CreateProductB();  
    cout << productA->GetName() << " / " << productB->GetDesc();  
}
```

```
output:  
Product A1 / Product B2  
Product A2 / Product B2
```



Design Pattern – Creational - Abstract Factory

Design Patterns-Creational-Builder

Constructs objects by separating construction and representation

```
Foo()  
{  
    ptr<PizzaBuilder> builder;  
    auto cook = make<Cook>();  
  
    builder = make<HawaiianBuilder>();  
    Pizza hawaiian = cook->MakePizza(builder);  
    hawaiian.Display();  
  
    builder = make<SpicyBuilder>();  
    Pizza spicy = cook->MakePizza(builder);  
    spicy.Display();  
}
```

Output:
Pizza with pan baked dough, mild sauce and ham+pineapple topping
Pizza with thin crust dough, hot sauce and pepperoni+salami topping

```
Cook // Director  
MakePizza(ptr<PizzaBuilder> builder)  
    auto pizza = builder->NewPizza();  
    builder->BuildDough(pizza);  
    builder->BuildSauce(pizza);  
    builder->BuildTopping(pizza);  
    return *pizza
```

```
PizzaBuilder  
NewPizza() make<Pizza>();  
BuildDough(pizza) = 0  
BuildSauce(pizza) = 0  
BuildTopping(pizza) = 0
```

```
HawaiianBuilder  
BuildDough(pizza) pizza.SetDough("pan baked")  
BuildSauce(pizza) pizza.SetSauce("mild")  
BuildTopping(pizza) pizza.SetTopping("ham+pineapple")
```

```
SpicyBuilder  
BuildDough(pizza) pizza.SetDough("thin crust")  
BuildSauce(pizza) pizza.SetSauce("hot")  
BuildTopping(pizza) pizza.SetTopping("pepperoni & salami")
```

```
Pizza  
SetDough(dough) { ... }  
SetSauce(sauce) { ... }  
SetTopping(topping) { ... }  
Display()  
    cout<< dough<< sauce<< topping  
private  
    string dough  
    string sauce  
    string topping
```

Design Pattern – Creational - Builder

miro

Design Patterns-Creational-Prototype(Clone)

Creates objects by cloning an existing object

```
Foo() {  
    std::shared_ptr<Prototype> prototype;  
    std::shared_ptr<Prototype> clone;  
  
    prototype = std::make_shared<ConcretePrototypeA>();  
    clone = prototype->Clone();  
    clone->Display();  
  
    prototype = std::make_shared<ConcretePrototypeB>();  
    clone = prototype->Clone();  
    clone->Display();  
}
```

```
output:  
ctor: Prototype A  
clone: Prototype A  
    ctor: Prototype A+copy  
Display: Prototype A+copy  
ctor: Prototype B  
clone: Prototype B  
    ctor: Prototype B+copy  
Display: Prototype B+copy
```

```
Prototype  
-Clone() = 0  
-Display() = 0
```

```
PrototypeA  
PrototypeA() : name("Prototype A")  
    cout << "ctor" << name  
PrototypeA(PrototypeA&) name(+copy)  
    cout << "ctor" << name  
Clone()  
    cout << "clone" << name  
    return make<PrototypeA>(this)  
void Display()  
    cout << name  
private  
    string name;
```

```
PrototypeB  
PrototypeB() : name("Prototype B")  
    cout << "ctor" << name  
PrototypeB(PrototypeB&) name(+copy)  
    cout << "ctor" << name  
Clone()  
    cout << "clone" << name  
    return make<PrototypeB>(this)  
void Display()  
    cout << name  
private  
    string name;
```

Design Pattern – Creational - Prototype

Design Patterns-Creational-Singleton (old)

Restricts object creation to only one instance

```
foo() {  
    auto data = Singleton::GetInstance();  
    data->ShowMessage();  
    data->Set...(...);  
    data->Get...();  
}
```

Old School Style - Potential Race Condition

Design Pattern - Creational - Singleton

```
struct Singleton {  
    static Singleton* getInstance() {  
        if (instance == nullptr) {  
            instance = new Singleton();  
        }  
        return instance;  
    }  
    void ShowMessage() { std::cout << "Hello, World!\n"; }  
    //add data accessors  
private:  
    Singleton() {}  
    static Singleton* instance;  
    //add data members  
};
```

Singleton* Singleton::instance = nullptr;

miro

Design Patterns-Creational-Singleton

Restricts object creation to only one instance

```
foo() {  
    auto data = Singleton::GetInstance();  
    data->ShowMessage();  
    data->Set...(...);  
    data->Get...();  
}
```

Design Pattern – Creational - Singleton

Singleton

```
static GetInstance()  
static Data data  
return data.GetPtr()
```

Data

```
GetPtr() { return ptr(this) }  
ShowMessage() { cout << "Hello, World!"; }  
Set...(...) {...}  
Get...() {...}  
private:  
    // Data Members
```

Design Patterns-Creational-Monostate

Restricts object creation to only one instance (Variation of Singleton)

```
foo() {  
    Monostate::SetName(...);  
    Monostate::SetAge(...);  
    string = Monostate::GetName();  
    age = Monostate::GetAge();  
}
```

Monostate

```
GetName()    { return m_name; }  
SetName(name) { m_name = name; }  
GetAge()     { return m_age; }  
SetAge(age)  { m_age = age; }  
private:  
    static string m_name;  
    static int    m_age;
```

```
string Monostate::m_name;  
int    Monostate::m_age;
```

Design Pattern - Creational - Singleton(Monostate)

miro

Design Patterns-Tempetiezed Factory<>

```
template<typename Base, typename Key=std::string>
struct Factory {
    template<typename Derived, typename... Args>
    static void Register(Key key, Args ...args) {
        Map()[key] = [args...]() { return std::make_shared<Derived>(args...); };
    }
    template<typename... Args>
    static std::shared_ptr<Base> Create(Key key, Args ...args) {
        if(Map().find(key)==Map().end())
            return nullptr;
        return Map()[key](args...);
    }
private:
    using MMap = std::map<Key, std::function<std::shared_ptr<Base>()>>;
    static MMap& Map() {
        static MMap map;
        return map;
    }
};
```

Design Patterns-Tempetiezed Factory<>

```
struct Service { virtual void Do() = 0; };

struct ServiceA : Service {
    void Do() { std::cout << "ServiceA: doing something!\n"; }
};

struct ServiceB : Service {
    ServiceB(std::string param): param(param){}
    void Do() { std::cout << "ServiceB: using " << param << "!\n"; }
private:
    std::string param;
};

int mainA() {
    Factory<Service>::Register<ServiceA>("Foo");
    Factory<Service>::Register<ServiceB>("Bar1", "Extra");
    Factory<Service>::Register<ServiceB>("Bar2", "Data");

    Factory<Service>::Create("Foo")->Do();
    Factory<Service>::Create("Bar1")->Do();
    Factory<Service>::Create("Bar2")->Do();
    return 0;
}
```

Output:

```
ServiceA: doing something!
ServiceB: using Extra!
ServiceB: using Data!
```


Design Patterns-Tempetiezed Factory<>

```
struct Shape { virtual std::string Type() = 0; };

struct Square : Shape { std::string Type() { return "Square"; } };
struct Circle : Shape { std::string Type() { return "Circle"; } };
struct Triangle : Shape { std::string Type() { return "Triangle"; } };

int main() {
    using ShapeFactory = Factory<Shape, int>;
    ShapeFactory::Register<Square>(1);
    ShapeFactory::Register<Circle>(2);
    ShapeFactory::Register<Triangle>(3);

    Factory<Shape, int> shapeFactory;
    auto shape1 = shapeFactory.Create(1);
    auto shape2 = shapeFactory.Create(2);
    auto shape3 = shapeFactory.Create(3);

    std::cout << "shape1:" << shape1->Type() << "\n";
    std::cout << "shape2:" << shape2->Type() << "\n";
    std::cout << "shape3:" << shape3->Type() << "\n";
}
```

Output:

```
shape1:Square
shape2:Circle
shape3:Triangle
```


Design Patterns-Tempetiezed Factory<>

```
template<typename Base, typename Key=std::string>
struct Factory {
    template<typename Derived, typename... Args>
    static void Register(Key key, Args ...args) {
        Map()[key] = [args...]() { return std::make_shared<Derived>(args...); };
    }
    template<typename... Args>
    static std::shared_ptr<Base> Create(Key key, Args ...args) {
        if(Map().find(key)==Map().end())
            return nullptr;
        return Map()[key](args...);
    }
private:
    using MMap = std::map<Key, std::function<std::shared_ptr<Base>()>>;
    static MMap& Map() {
        static MMap map;
        return map;
    }
};
```


Design Patterns-Structural



Design Patterns-Structural

- Adapter - An intermediary translating interface.
- Bridge - Decouples an abstraction from its implementation.
- Composite - Dynamic collection of objects manipulated as one object.
- Decorator - Daisy chain of filters extending interface behavior.
- Façade - Provides a simplified interface to a larger body of code.
- Flyweight - Reduces the cost of creating and manipulating of similar objects.
- Proxy - A placeholder for another object to control access, reduce complexity

Design Patterns-Structural

Greater flexibility and modularity in for better:

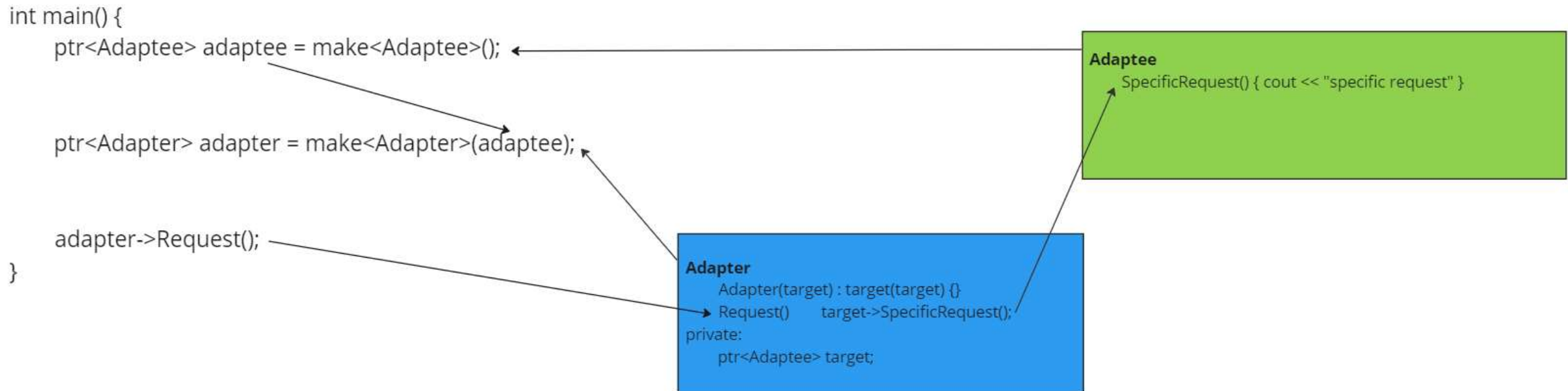
- Abstraction
- Flexibility
- Reduced coupling
- Separation of algorithms from the structure

Used for Improved:

- Extensibility
- Flexibility
- Performance
- Security

Design Patterns-Structural-Adapter(Basic)

An intermediary translating interface



Design Pattern – Structural - Adapter (Basic)

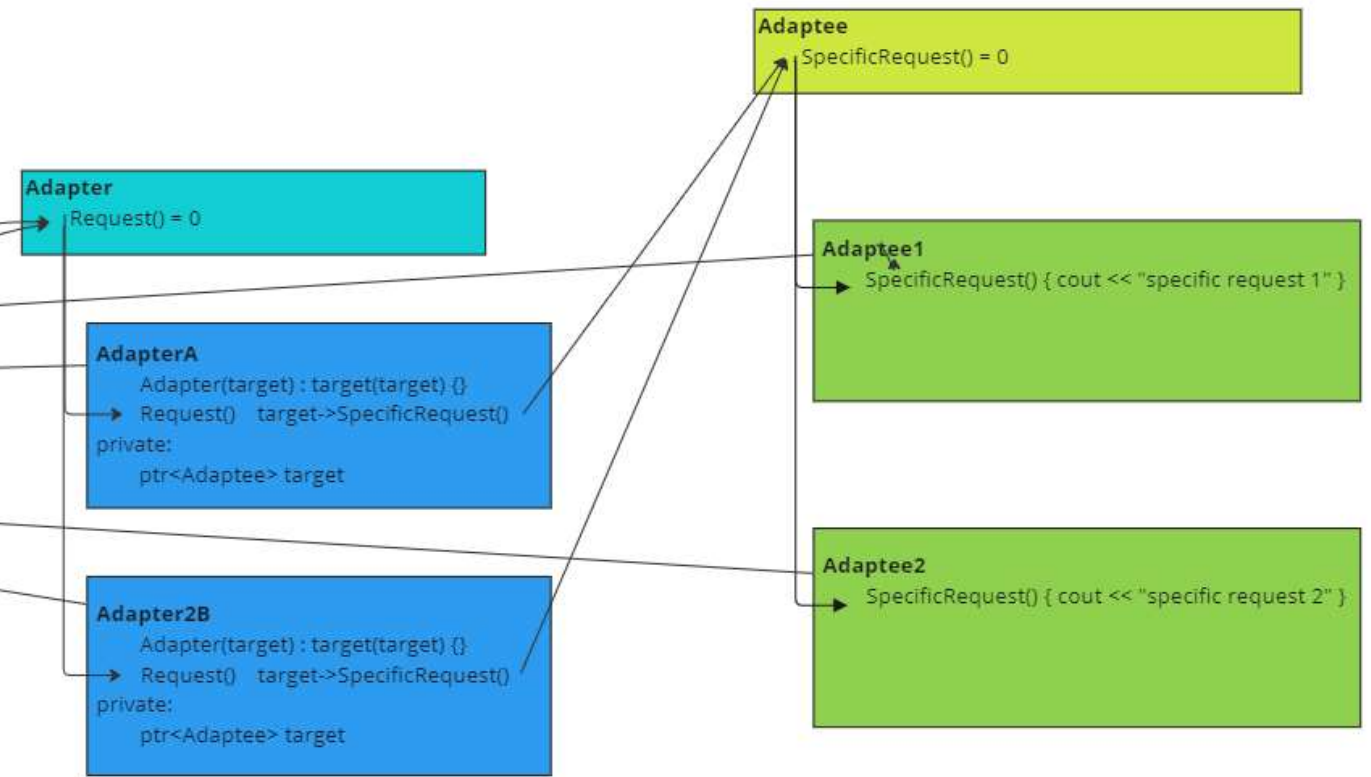
miro

Design Patterns-Structural-Adapter(Abstract)

An intermediary translating interface

```
int main() {  
    ptr<Adaptee> adaptee;  
    ptr<Adapter> adapter;  
  
    adaptee = make<Adaptee1>();  
    adapter = make<AdapterA>(adaptee);  
    adapter->Request();  
  
    adaptee = make<Adaptee2>();  
    adapter = make<AdapterB>(adaptee);  
    adapter->Request();  
}
```

output:
Adaptee 1's specific request.
Adaptee 2's specific request.



Design Pattern – Structural - Adapter (Abstract)

miro

Design Patterns-Structural-Decorator

Daisy chain of filters extending interface behavior

```
Foo() {  
  
    ptr<Component> component = make<ConcreteComponent>();  
  
    ptr<Component> decoratorA = make<ConcreteDecoratorA>(component);  
  
    ptr<Component> decoratorB = make<ConcreteDecoratorB>(decoratorA);  
  
    decoratorB->Operation();  
}
```

Output

```
ConcreteDecoratorA Operation  
ConcreteComponent Operation  
ConcreteDecoratorB Operation
```

Component
operation() = 0;

ConcreteComponent
operation() { cout << "Component operation"; }

Decorator
Decorator(component) : component(component) {}
Operation() { component->Operation() }
protected:
ptr<Component> component

ConcreteDecoratorA
Decorator(component) : Decorator(component) {}
Operation() {
 cout << "Concrete Decorator A Operation"
 Decorator::Operation()
}

ConcreteDecoratorB
Decorator(component) : Decorator(component) {}
Operation() {
 Decorator::Operation()
 cout << "Concrete Decorator B Operation"
}

Design Pattern - Structural - Decorator

Design Patterns-Structural-Decorator

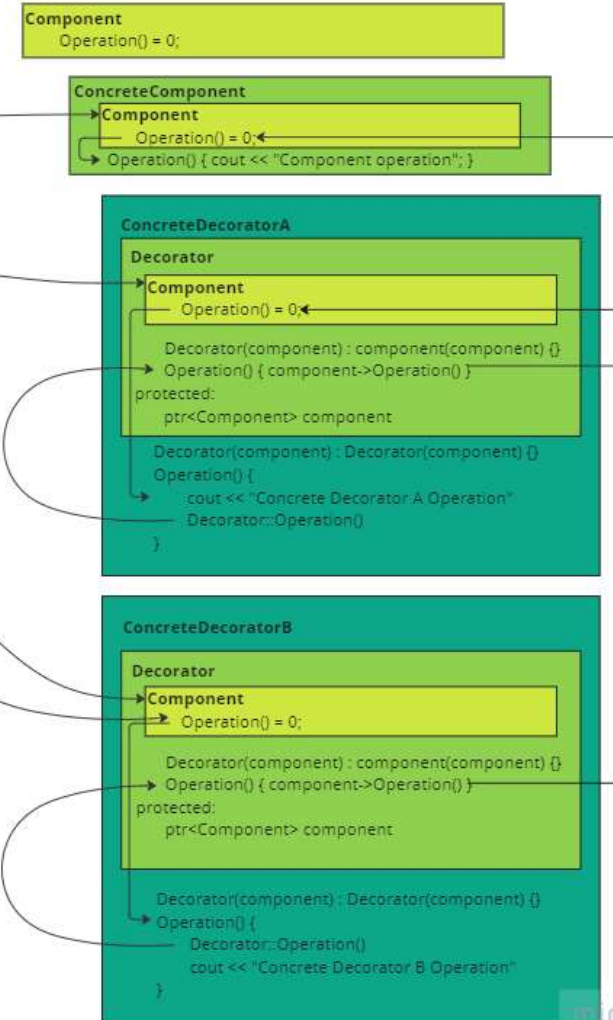
Daisy chain of filters extending interface behavior

```
Foo() {  
    ptr<Component> component = make<ConcreteComponent>();  
    ptr<Component> decoratorA = make<ConcreteDecoratorA>( component );  
    ptr<Component> decoratorB = make<ConcreteDecoratorB>( decoratorA );  
    decoratorB->Operation();  
}
```

Output

```
ConcreteDecoratorA Operation  
ConcreteComponent Operation  
ConcreteDecoratorB Operation
```

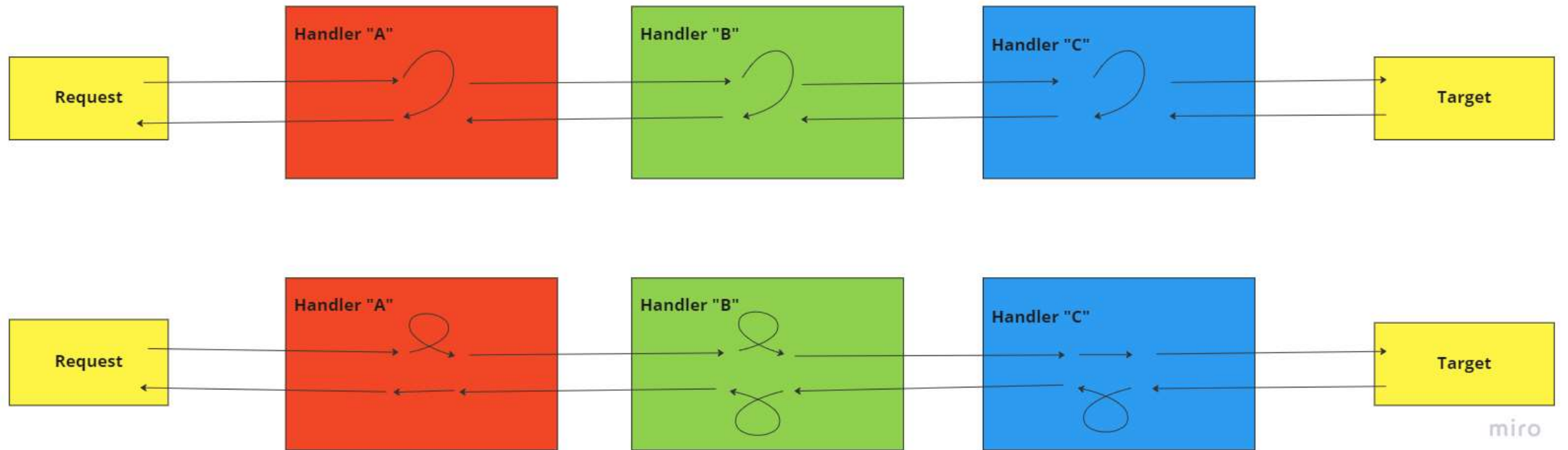
Design Pattern - Structural - Decorator



Design Patterns

Daisy Chain Filtering: Intercepting vs. Modifying

Daisy Chain Filtering: Intercepting vs. Modifying

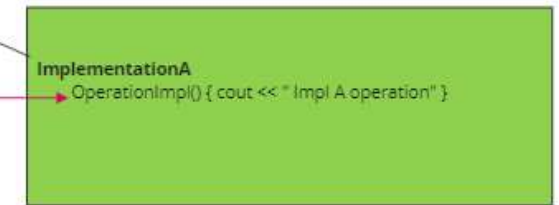
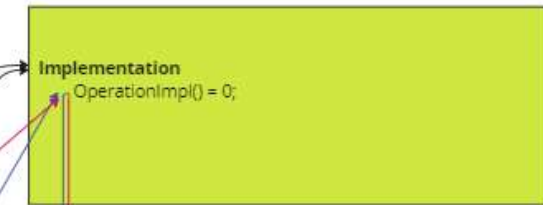
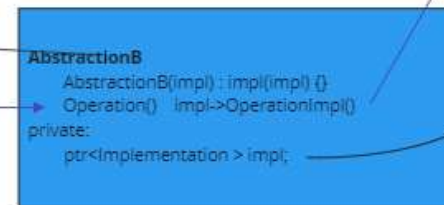
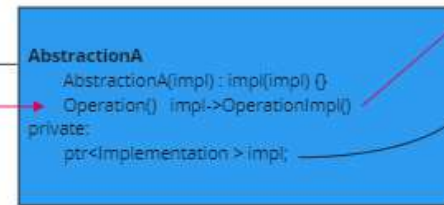
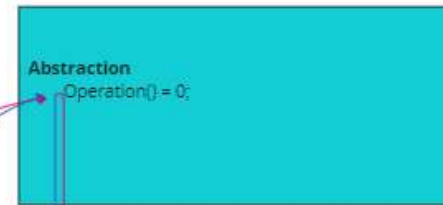


Design Patterns-Structural-Bridge

Decouples an abstraction from its implementation

```
int main() {  
  ptr<Implementation> implementation;  
  ptr<Abstraction> abstraction;  
  
  implementation = make<ImplementationA>();  
  abstraction = make<RefinedAbstraction>(implementation);  
  abstraction->Operation();  
  
  implementation = make<ImplementationB>();  
  abstraction = make<RefinedAbstraction>(implementation);  
  abstraction->Operation();  
}
```

```
output:  
Impl A operation  
Impl B operation
```

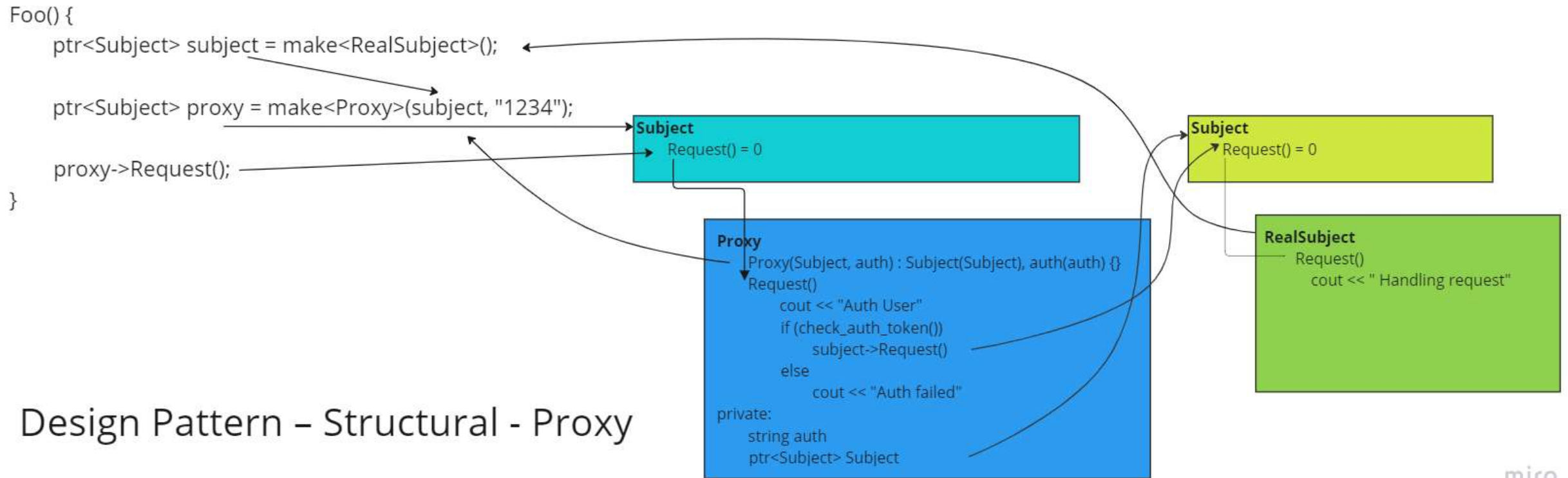


Design Pattern – Structural - Bridge

miro

Design Patterns-Structural-Proxy

A placeholder for another object to control access, reduce complexity



Design Patterns-Structural-Flyweight

Reduces the cost of creating and manipulating of similar objects

```
Foo() {
```

```
    FlyweightFactory factory;
```

```
    ptr<Flyweight> flyweight1 = factory.GetFlyweight(1);  
    ptr<Flyweight> flyweight2 = factory.GetFlyweight(2);  
    ptr<Flyweight> flyweight3 = factory.GetFlyweight(1);
```

```
    flyweight1->Operation(10);  
    flyweight2->Operation(20);  
    flyweight3->Operation(30);
```

```
}
```

Output:

```
Intr state: 1, Extr state: 10  
Intr state: 2, Extr state: 20  
Intr state: 1, Extr state: 30
```

Design Pattern – Structural - Flyweight

Flyweight

```
Operation(extrState) = 0;
```

ConcreteFlyweight

```
ConcreteFlyweight(int intrState) : intrState(intrState) {}
```

```
void Operation(int extrState)  
    cout << "Intr: " << intrState << ", Extr: " << extrState;
```

```
private:  
    int intrState;
```

FlyweightFactory

```
GetFlyweight(int key)  
    if ( !flyweights.find(key) )  
        flyweights[key] = make<ConcreteFlyweight>(key)  
    return flyweights[key]
```

```
private:  
    map<int, ptr<Flyweight>> flyweights;
```

miro

Design Patterns-Structural-Facade

Provides a simplified interface to a larger body of code

```
Foo() {  
  Facade facade;  
  
  facade.OperationABC();  
  
  facade.OperationAC();  
}
```

```
Facade  
  OperationABC()  
    subsystemA.Operation1()  
    subsystemB.Operation2()  
    subsystemC.Operation3()  
  OperationAC()  
    subsystemA.Operation1()  
    subsystemC.Operation1()  
private:  
  SubsystemA subsystemA;  
  SubsystemB subsystemB;  
  SubsystemC subsystemC;
```

```
SubsystemA  
  Operation1() {cout << "SubsystemA Operation1" }
```

```
SubsystemB  
  Operation2() { cout << "SubsystemB Operation2" }
```

```
SubsystemC  
  Operation1() { cout << "SubsystemC Operation1" }  
  Operation3() {:cout << "SubsystemC Operation3" }
```

Design Pattern – Structural - Facade

miro

Design Patterns-Structural-Composite

Dynamic collection of objects manipulated as one object

```
ptr<Component> composite = make<Composite>()
```

```
composite->Add(make<Leaf1>)
```

```
composite->Add(make<Leaf2>)
```

```
composite->Add(make<Leaf2>)
```

```
composite->Operation(...)
```

Component

```
Operation() = 0;  
Add(component) {}  
Remove(component) {}  
getChild(int index) { nullptr; }
```

Composite

```
Operation() { for_each(children) child->Operation() }  
Add(component) { children.Add(component) }  
Remove(component) { children.Remove(component) }  
GetChild(index) { children[index]; }  
private  
vector<ptr<Component>> children
```

Leaf1

```
Operation() { cout << "Leaf 1 operation"; }
```

Leaf2

```
Operation() { cout << "Leaf 2 operation"; }
```

Leaf3

```
Operation() { cout << "Leaf 3 operation"; }
```

Design Pattern – Structural - Composite

miro

Design Patterns-Behavioral



Design Patterns-Behavioral

- Chain of Responsibility - Delegates commands through cascade processing.
- Command - Batching of multiple actions and parameters until executed.
- Interpreter - Implements a specialized language.
- Iterator - Access to the elements of an object hiding its implementation.
- Mediator - Like the Observer pattern but excludes the notifier sender.
- Memento - Provides the ability to restore an object state (undo).
- Observer - A publish/subscribe, all observer objects see event notifications.
- State - An object to alter its behavior when its internal state changes.
- Strategy - Allows switchable algorithms on-the-fly at runtime.
- Template Method - Extends skeleton functionality by providing concrete behavior.
- Visitor - Separation of responsibility applied to an object hierarchy.

Design Patterns-Behavioral

Allows for greater flexibility and modularity in the design of the system to improve:

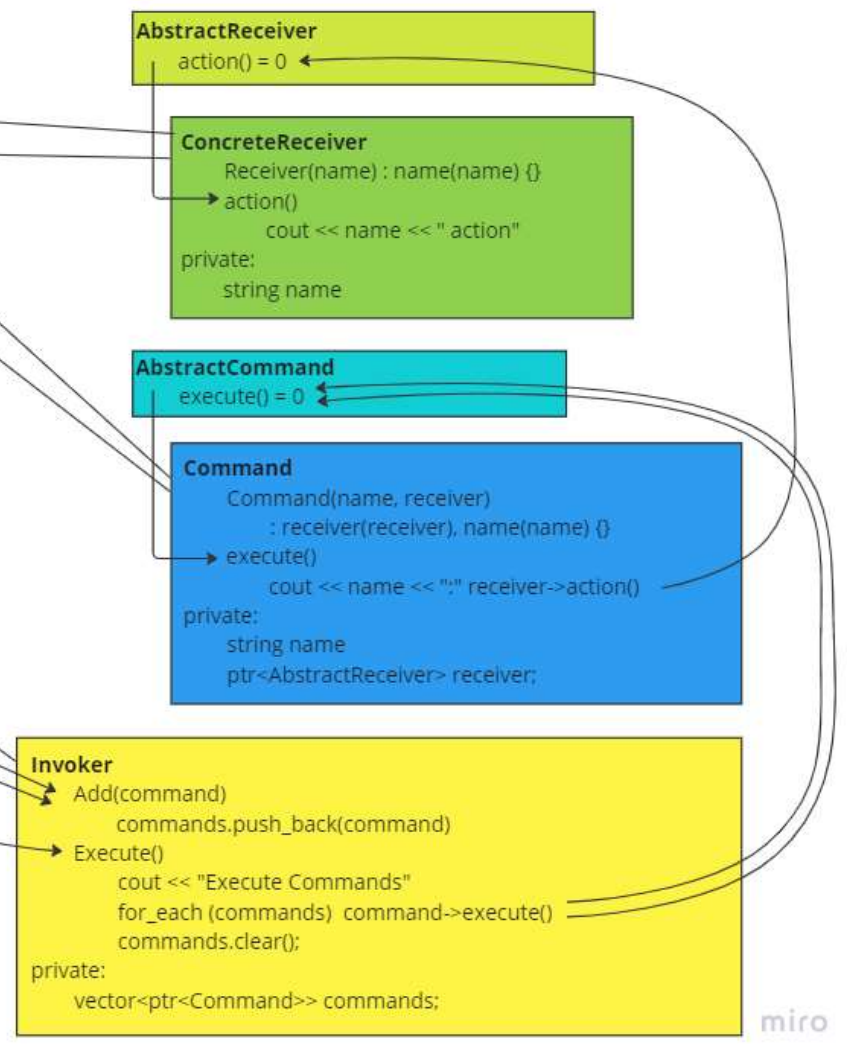
- Extensibility
- Decoupling
- Flexibility
- Maintainability
- Modularity
- Performance
- Reliability
- Scalability
- Security
- Testability
- Usability

Design Patterns-Behavioral-Command

Batching of multiple actions and parameters until executed (Transaction Caching)

```
Foo() {  
    ptr<AbstractReceiver> receiverA = make<Receiver>("A");  
    ptr<AbstractReceiver> receiverB = make<Receiver>("B");  
  
    ptr<AbstractCommand> command1 = make<Command>("1", receiverA);  
    ptr<AbstractCommand> command2 = make<Command>("2", receiverB);  
  
    ptr<Invoker> invoker = make<Invoker>();  
  
    invoker->Add(command1);  
    invoker->Add(command2);  
  
    invoker->Execute();  
}
```

Output:
Execute Commands
1:A action
2:B action



Design Pattern – Behavioral - Command

miro

Design Patterns-Behavioral-Chain of Responsibility

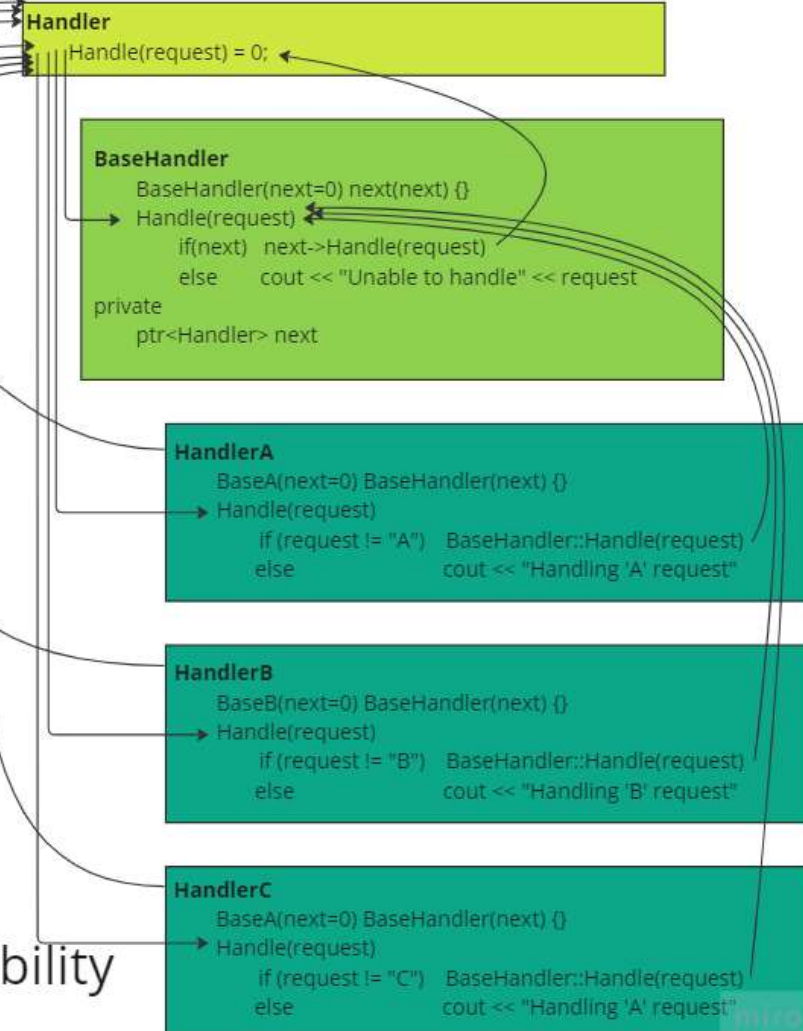
Delegates commands through cascade processing

```
Foo {  
  ptr<Handler> handlerC = make<HandlerC>();  
  ptr<Handler> handlerB = make<HandlerB>(handlerC);  
  ptr<Handler> handlerA = make<HandlerA>(handlerB);  
  handlerA->Handle("C");  
  handlerA->Handle("B");  
  handlerA->Handle("D");  
  handlerA->Handle("A");  
}
```

Output:

```
Handling 'C' request  
Handling 'B' request  
Unable to handle request 'D'  
Handling 'A' request
```

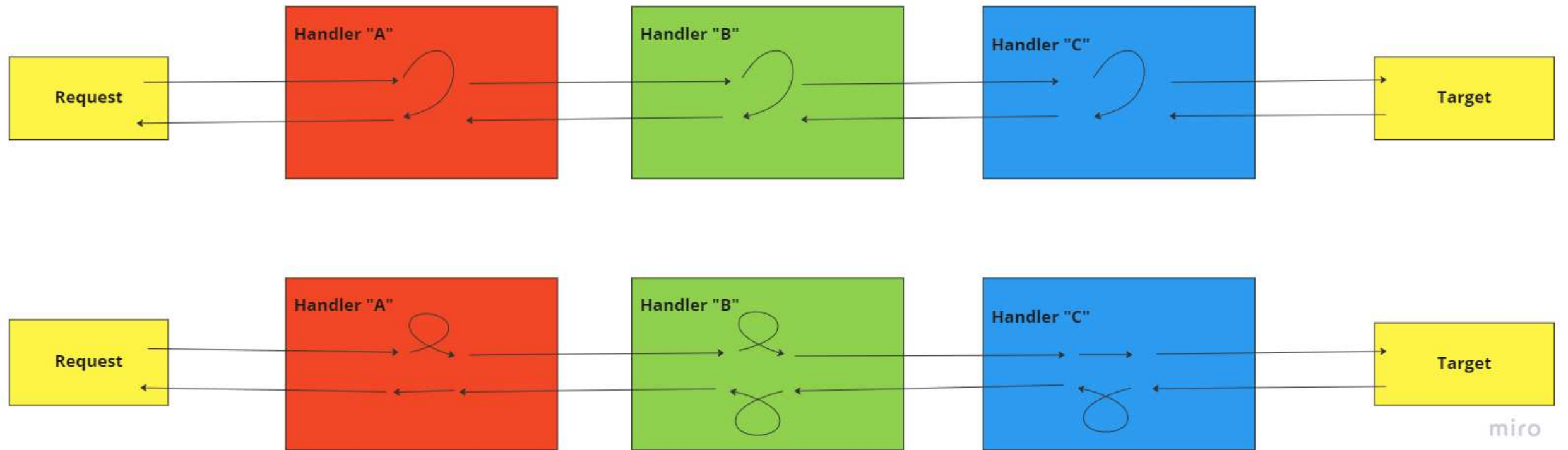
Design Pattern - Behavioral - Chain of Responsibility



Design Patterns

Daisy Chain Filtering: Intercepting vs. Modifying

Daisy Chain Filtering: Intercepting vs. Modifying



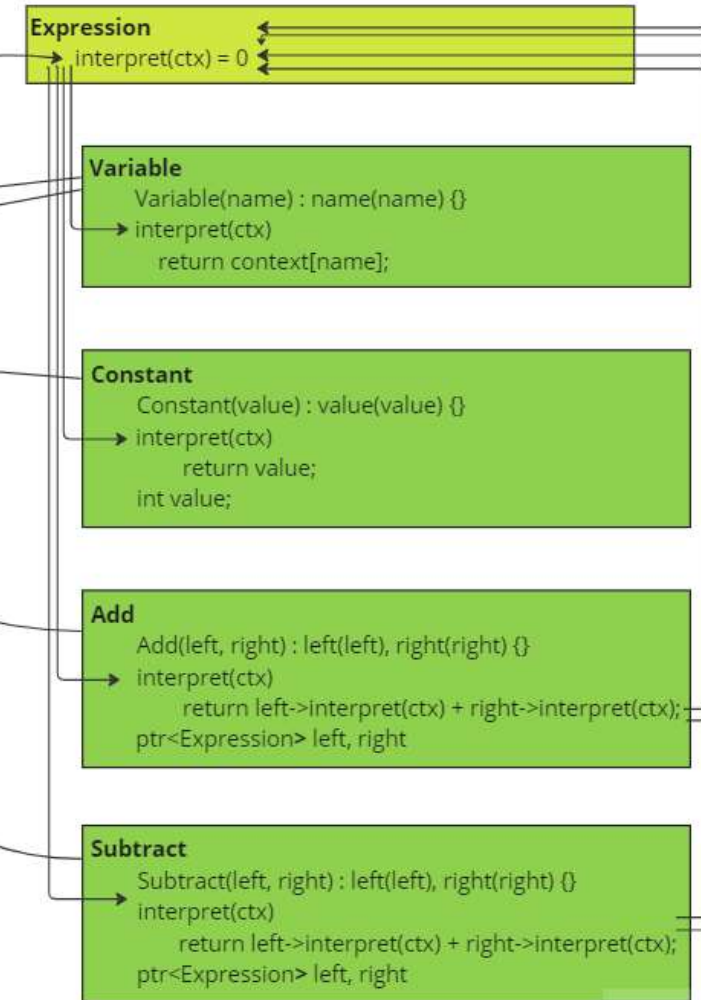
Design Patterns-Behavioral-Interpreter

Implements a specialized language

```
Foo() {  
    // = ((x+2)-y)  
    ptr<Expression> expr =  
        make<Subtract>( ←  
            make<Add>( ←  
                make<Variable>("x"), ←  
                make<Constant>(2), ←  
                make<Variable>("y") ←  
            );  
  
    Context context;  
    context["x"] = 10;  
    context["y"] = 5;  
  
    cout << "Result: " << expr->interpret(context)  
}
```

Output
Result: 7

Design Pattern – Behavioral - Interpreter



miro

Design Patterns-Behavioral-Iterator

Access to the elements of an object hiding its implementation

```
Foo() {  
    VectorAggregate aggregate; ←  
    aggregate.add(1); ←  
    aggregate.add(5); ←  
    aggregate.add(7); ←  
  
    ptr<Iterator> iterator = aggregate.create_iterator(); ←  
  
    while (iterator->has_next()) ←  
        cout << iterator->next() << " "; ←  
}
```

Output:
1 5 7

Aggregate

Create_iterator() = 0

VectorAggregate

```
add(value) { vector.push_back(value); }  
create_iterator() { return make<VectorIterator>(vector); }  
private:  
vector<> vector;
```

Iterator

```
next() = 0  
has_next() = 0
```

VectorIterator

```
VectorIterator(vector<int>& vector) : vector(vector), index(0) {}  
next() { return vector[index++]; }  
has_next() { return index < vector.size(); }  
private  
vector<>& vector;  
int index;
```

Design Pattern – Behavioral - Iterator

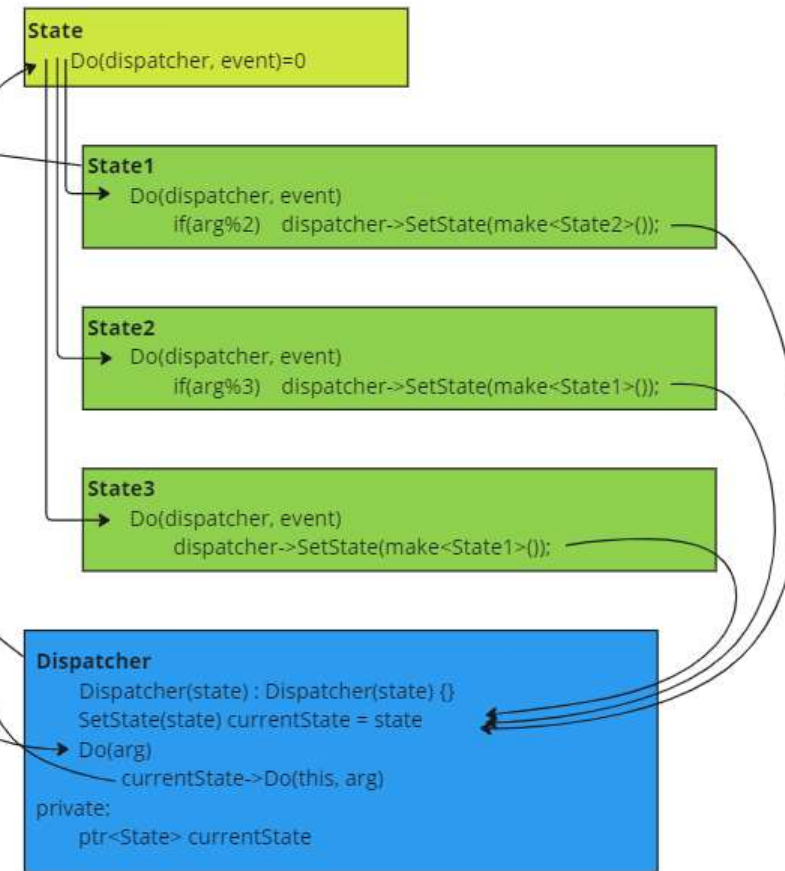
Design Patterns-Behavioral-State

An object to alter its behavior when its internal state changes

```
Foo() {  
    auto start = make<State1>();  
  
    auto dispatcher = make<Dispatcher>(start);  
  
    vector events{1,2,3,4,5,6,7,8,9,0};  
    for(auto event : events)  
    {  
        if(!event) break;  
        dispatcher->Do(event);  
    }  
}
```

Output:

```
Current state: 1 event:1 Resulting state: 2  
Current state: 2 event:2 Resulting state: 3  
Current state: 3 event:3 Resulting state: 1  
Current state: 1 event:4 Resulting state: 1  
Current state: 1 event:5 Resulting state: 2  
Current state: 2 event:6 Resulting state: 2
```



Design Patterns – Behavioral - State

miro

Design Patterns-Behavioral-Observer

A publish/subscribe, all observer objects see event notifications

```
Foo() {
```

```
    auto subject = make<Subject>();
```

```
    auto observer1 = make<Observer>("Observer 1");
```

```
    auto observer2 = make<Observer>("Observer 2");
```

```
    subject->Attach(observer1);
```

```
    subject->Attach(observer2);
```

```
    subject->SetData(10);
```

```
    auto observer3 = make<Observer>("Observer 3");
```

```
    subject->Attach(observer3);
```

```
    subject->SetData(20);
```

```
    subject->Detach(observer2);
```

```
    subject->SetData(30);
```

```
}
```

Output:

```
Observer 1 received update: 10  
Observer 2 received update: 10  
Observer 1 received update: 20  
Observer 2 received update: 20  
Observer 3 received update: 20  
Observer 1 received update: 30  
Observer 3 received update: 30
```

BaseSubject

```
Attach(observer)  bservers.push_back(observer)  
Detach(observer) bservers.erase(observer)  
Notify(data)      for_each(observers) observer->Update(data)  
private:  
vector<ptr<Observer>> observers
```

Subject

```
SetData(data)  
    this->data = data;  
    Notify(data);  
private:  
int data;
```

AbstractObserver

```
Update(data) = 0
```

Observer

```
ConcreteObserver(name) : name(name) {}  
Update(data)  cout << name << << data  
private:  
string name
```

Design Pattern – Behavioral - Observer

miro

Design Patterns-Behavioral-Mediator

Like the Observer pattern but excludes the notifying the sender

```
Foo() {  
    auto mediator = make<ConcreteMediator>();  
  
    auto colleagueA = make<ConcreteColleague>("A", mediator);  
    auto colleagueB = make<ConcreteColleague>("B", mediator);  
    auto colleagueC = make<ConcreteColleague>("C", mediator);  
  
    mediator->AddColleague(colleagueA);  
    mediator->AddColleague(colleagueB);  
    mediator->AddColleague(colleagueC);  
  
    colleagueB->Send("Hello World!");  
}
```

Output:
B sent message: Hello World!
A received message: Hello World!
C received message: Hello World!

Design Pattern - Behavioral - Mediator

Colleague
Send(message) = 0;
Receive(message) = 0;

ConcreteColleague
ConcreteColleague(mediator) : Colleague(mediator) {}
Send(stringmessage)
mediator->SendMessage(this, message);
Receive(stringmessage)
cout << message
private
ptr<Mediator> mediator;

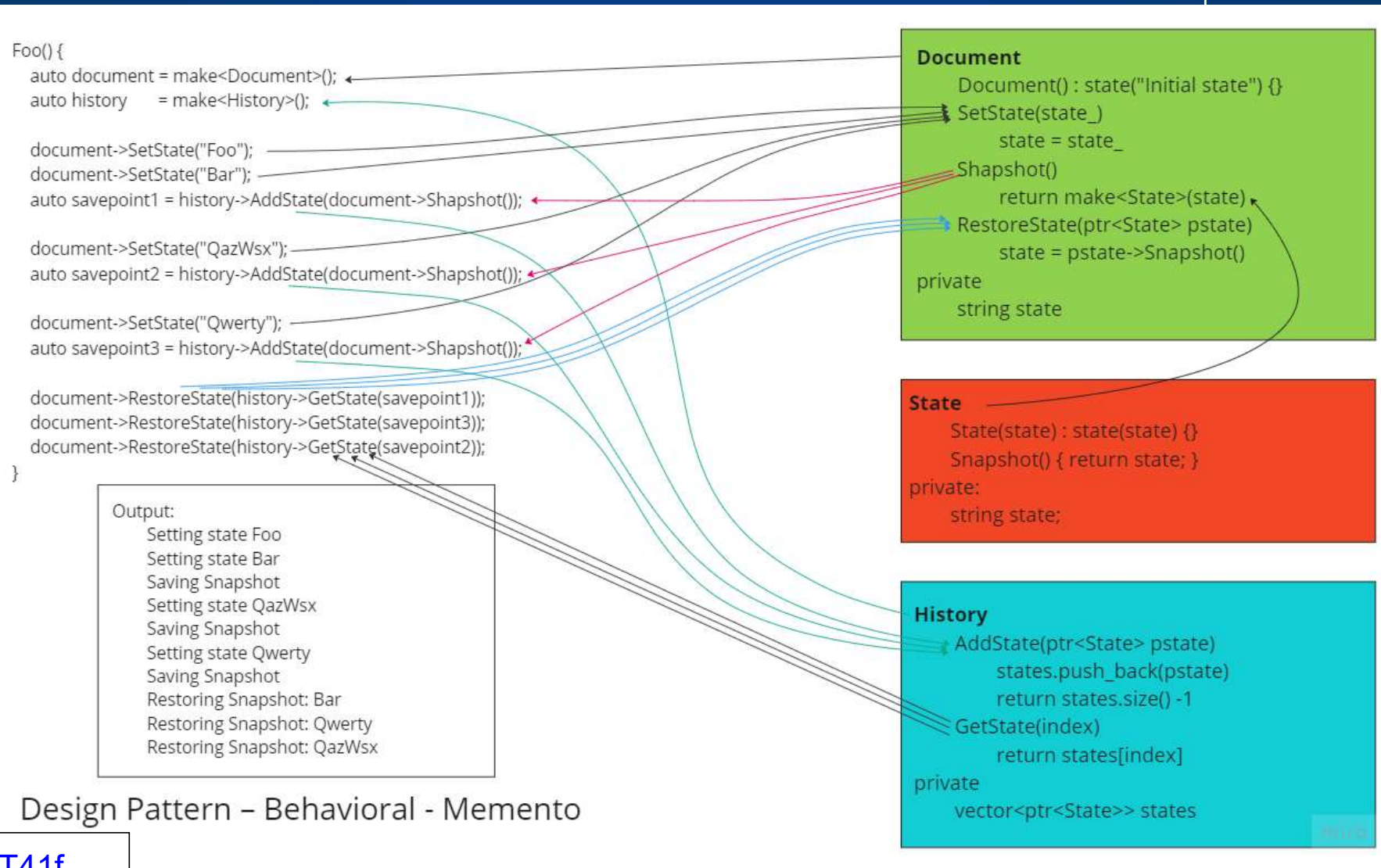
Mediator
Send(sender, message) = 0;

ConcreteMediator
AddColleague(colleague)
colleagues.push_back(colleague);
Send(sender, message)
for_each(colleagues)
if (colleague != sender)
colleague->Receive(message);
private
vector<ptr<Colleague>> colleagues;

miro

Design Patterns-Behavioral-Memento

Provides the ability to restore an object state (undo)



Design Patterns-Behavioral-Strategy

Allows switchable algorithms on-the-fly at runtime

```
Foo() {
```

```
    auto context = make<Context>();  
    auto strategyA = make<StrategyA>();  
    auto strategyB = make<StrategyB>();
```

```
    context->SetStrategy(strategyA);  
    context->ExecuteStrategy();  
    context->ExecuteStrategy();
```

```
    context->SetStrategy(strategyB);  
    context->ExecuteStrategy();  
    context->ExecuteStrategy();  
}
```

Output:

```
Executing strategy A  
Executing strategy A  
Executing strategy A  
Executing strategy B  
Executing strategy B
```

Strategy

```
Execute() = 0;
```

StrategyA

```
Execute()
```

```
cout << "Executing strategy A"
```

StrategyB

```
Execute()
```

```
cout << "Executing strategy B"
```

Context

```
SetStrategy(strategy)
```

```
currentStrategy = strategy;
```

```
ExecuteStrategy()
```

```
if(currentStrategy)
```

```
currentStrategy->Execute(); }
```

```
private:
```

```
ptr<Strategy> currentStrategy;
```

Design Pattern – Behavioral - Strategy

miro

Design Patterns-Behavioral-Template Method

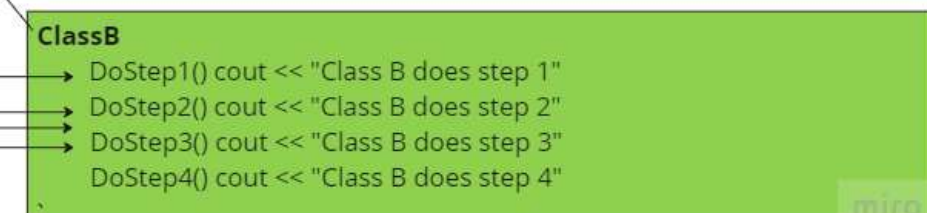
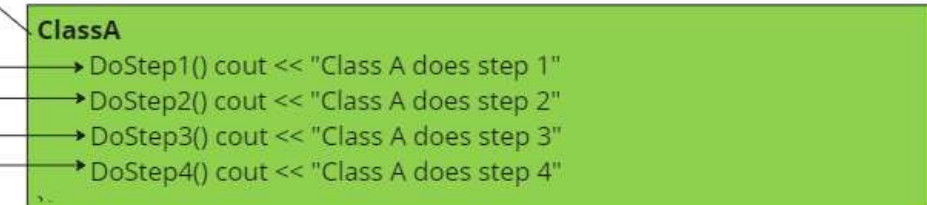
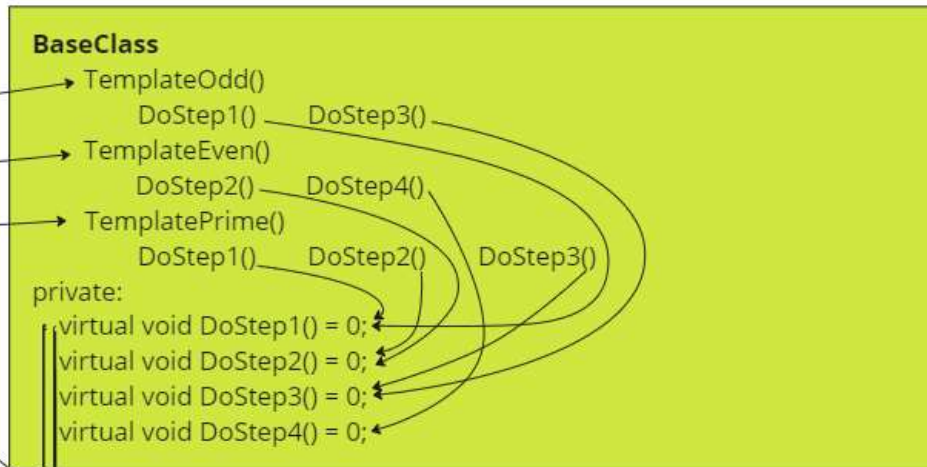
Extends skeleton functionality by providing concrete behavior

```
Foo() {  
    ptr<BaseClass> classA = make<ClassA>();  
    ptr<BaseClass> classB = make<ClassB>();  
  
    classB->TemplateOdd();  
    classA->TemplateEven();  
    classB->TemplatePrime();  
}
```

Output

```
Class B does step 1  
Class B does step 3  
Class A does step 2  
Class A does step 4  
Class B does step 1  
Class B does step 2  
Class B does step 3
```

Design Pattern - Behavioral - Template Method



Design Patterns-Behavioral-Visitor

Separation of responsibility applied to an object hierarchy

```
Foo() {  
    vector<ptr<Element>> elements {  
        make<ConcreteElement>(),  
        make<ConcreteElement>()  
    };  
  
    ConcreteVisitor visitor;  
    for(auto& element : elements)  
        element->Accept(visitor);  
}
```

Output:
ConcreteElement: ConcreteVisitor
ConcreteElement: ConcreteVisitor

Element
→ accept(& visitor) = 0

ConcreteElement
→ Accept(& visitor)
cout << "ConcreteElement: "
visitor.Visit(*this)

Visitor
→ Visit(Element&) = 0

ConcreteVisitor
→ Visit(& element)
cout << "ConcreteVisitor"

Design Pattern – Behavioral - Visitor

miro

Design Patterns-Behavioral-Visitor

Separation of responsibility applied to an object hierarchy

```
struct Node {
    friend ostream& operator<<(ostream& os, const Node& node)
    { return os << data; }

    ostream& PreOrder(ostream& os) {
        os << *this << ", ";
        if (left) left->PreOrder(os);
        if (right) right->PreOrder(os);
        return os;
    }
    //...
    ptr<class Node> left, right;
    type data;
}

struct PreOrder {
    ptr<class Node> node;
    PreOrder(ptr<class Node> node) : node(node) {}
    friend ostream& operator<<(ostream& os, const PreOrder& myself)
    { return myself.node->PreOrder(os); }
};

int main() {
    cout << using();
    int inOrder[] = { 3, 5, 6, 7, 10, 12, 13, 15, 16, 18, 20, 23, }; //lmr
    int postOrder[] = { 3, 7, 6, 10, 13, 12, 5, 18, 23, 20, 16, 15, }; //lrm
    auto tree = RebuildTree(postOrder, inOrder);
    cout << "PreOrder(mlr):\t" << PreOrder(tree) << "\n";
}
```

output:

Given post-order(lrm) & in-order(lmr)
Rebuild tree
List nodes in pre-order(mlr)

```
      15
     /  \
    5    16
   / \   \
  3  12  20
     / \  / \
    10 13 18 23
     /
    6
     \
     7
```

PreOrder(mlr): 15, 5, 3, 12, 10, 6, 7, 13, 16, 20, 18, 23,
InOrder(lmr): 3, 5, 6, 7, 10, 12, 13, 15, 16, 18, 20, 23,
PostOrder(lrm): 3, 7, 6, 10, 13, 12, 5, 18, 23, 20, 16, 15,

Design Patterns



Design Patterns

