accu 2023

DESIGN PATTERNS:

EXAMPLES IN C++

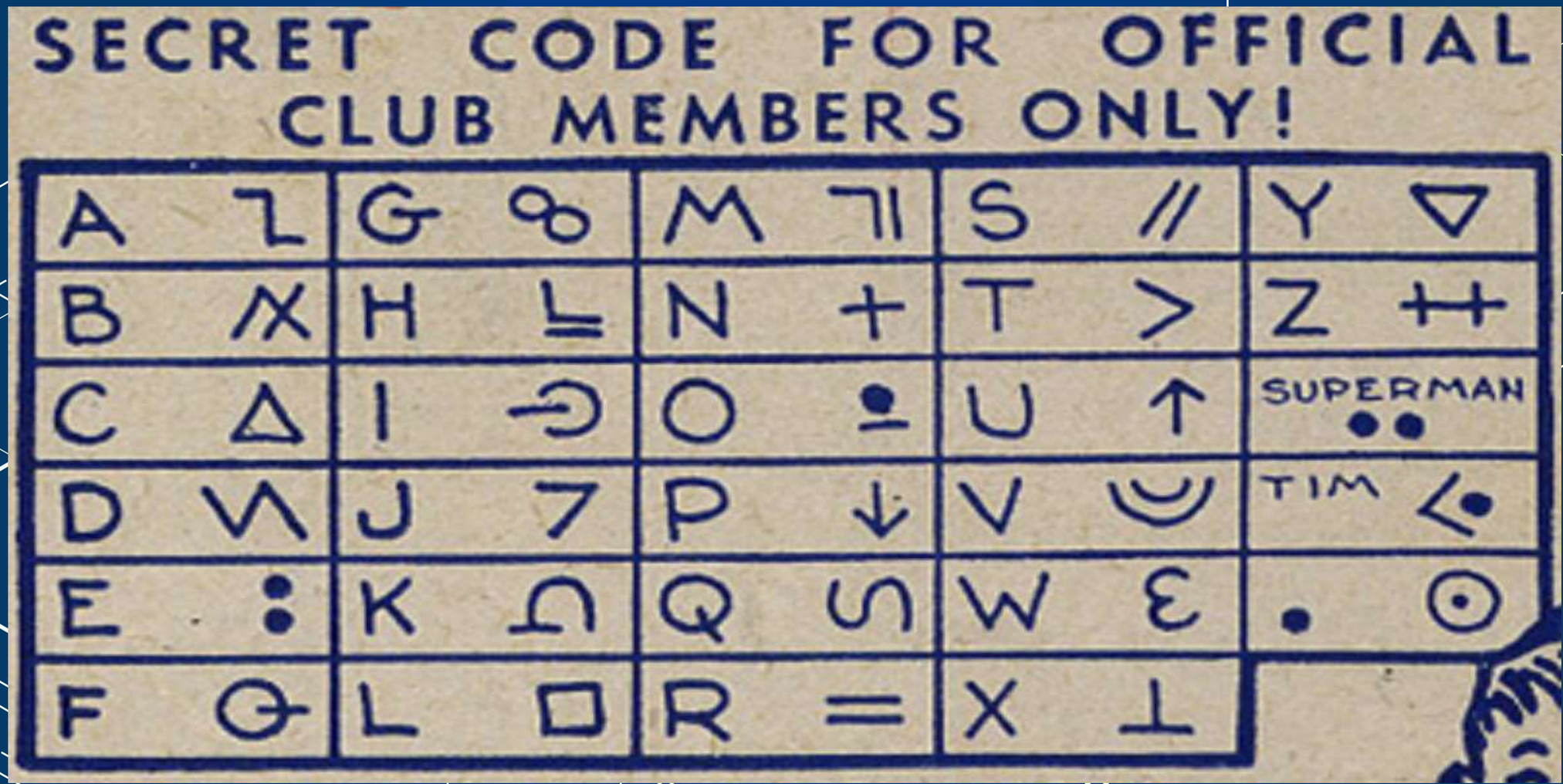
CHRIS RYAN

Design Patterns

ACCU 2023 - Bristol, England

Chris Ryan linkedin.com/in/chrisr98008

Design Patterns



Design Patterns

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

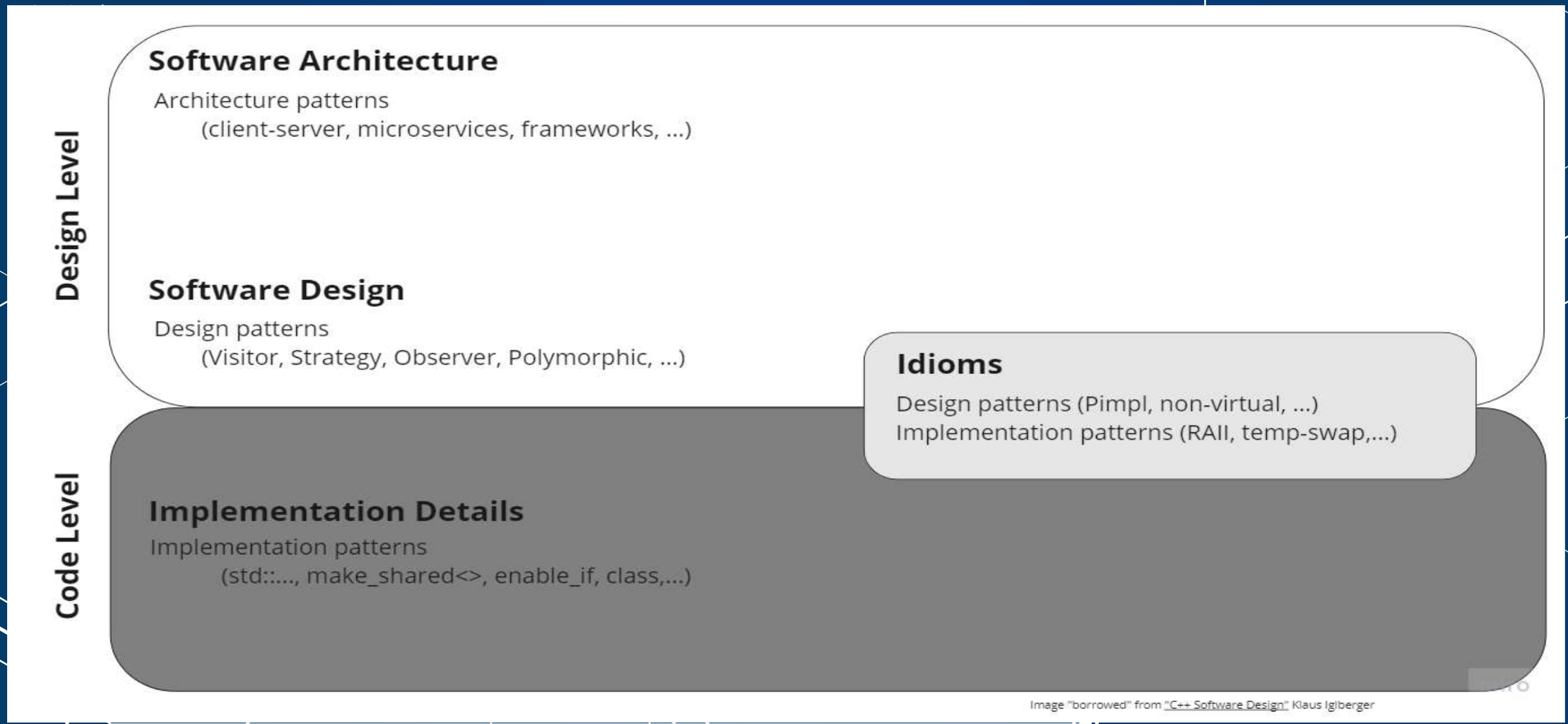
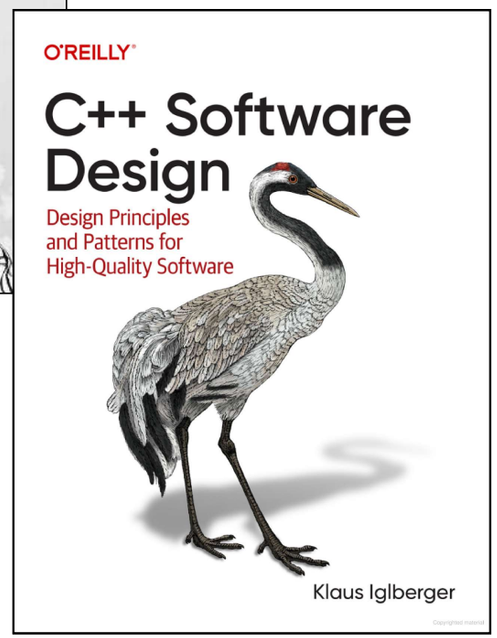


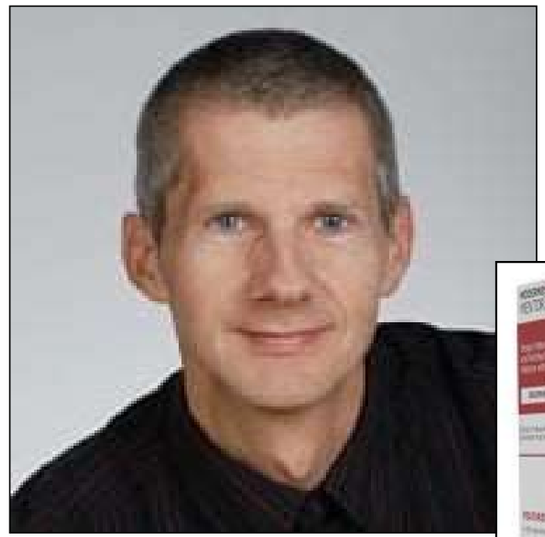
Image "borrowed" from "C++ Software Design" Klaus Iglberger

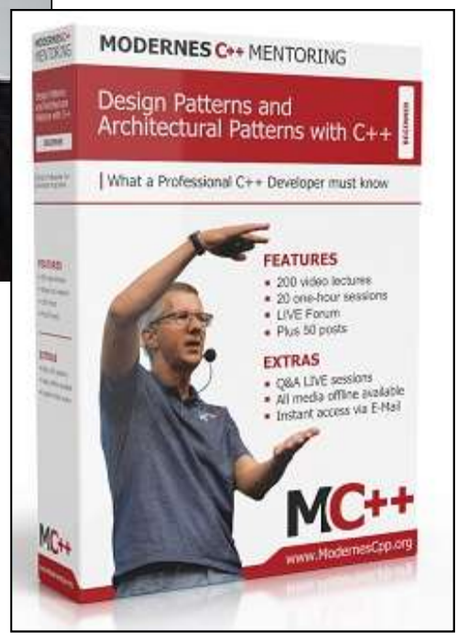
Design Patterns – Training, Books, Speakers

Klaus Iglberger



Rainer Grimm





Design Patterns

Pseudo Code / Slideware

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

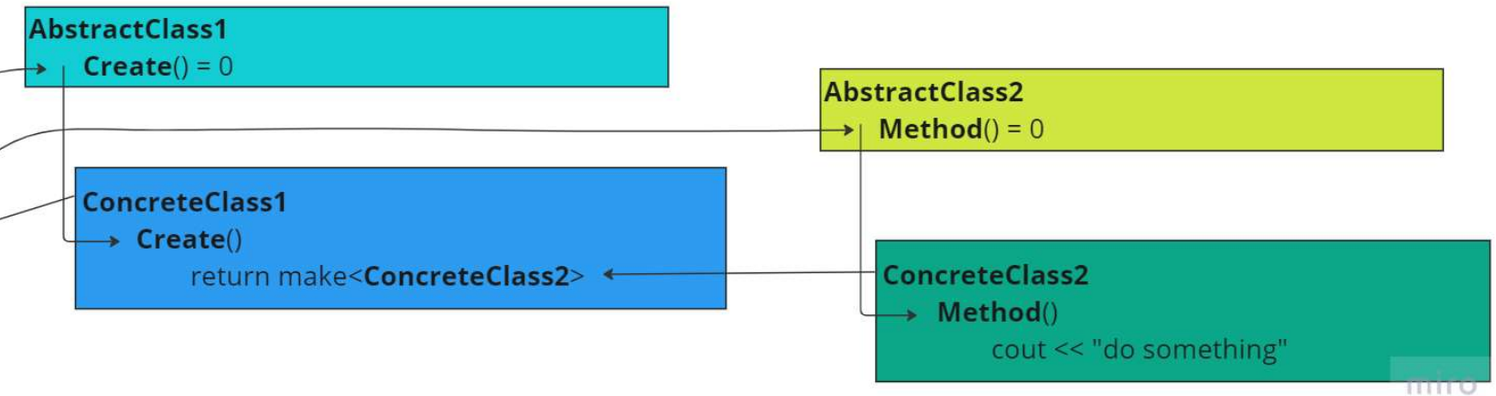
Godbolt sample links are over-simplified, missing best use of const, override, reference params &, and move params &&

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

Most samples are C++11 Modern compliant.

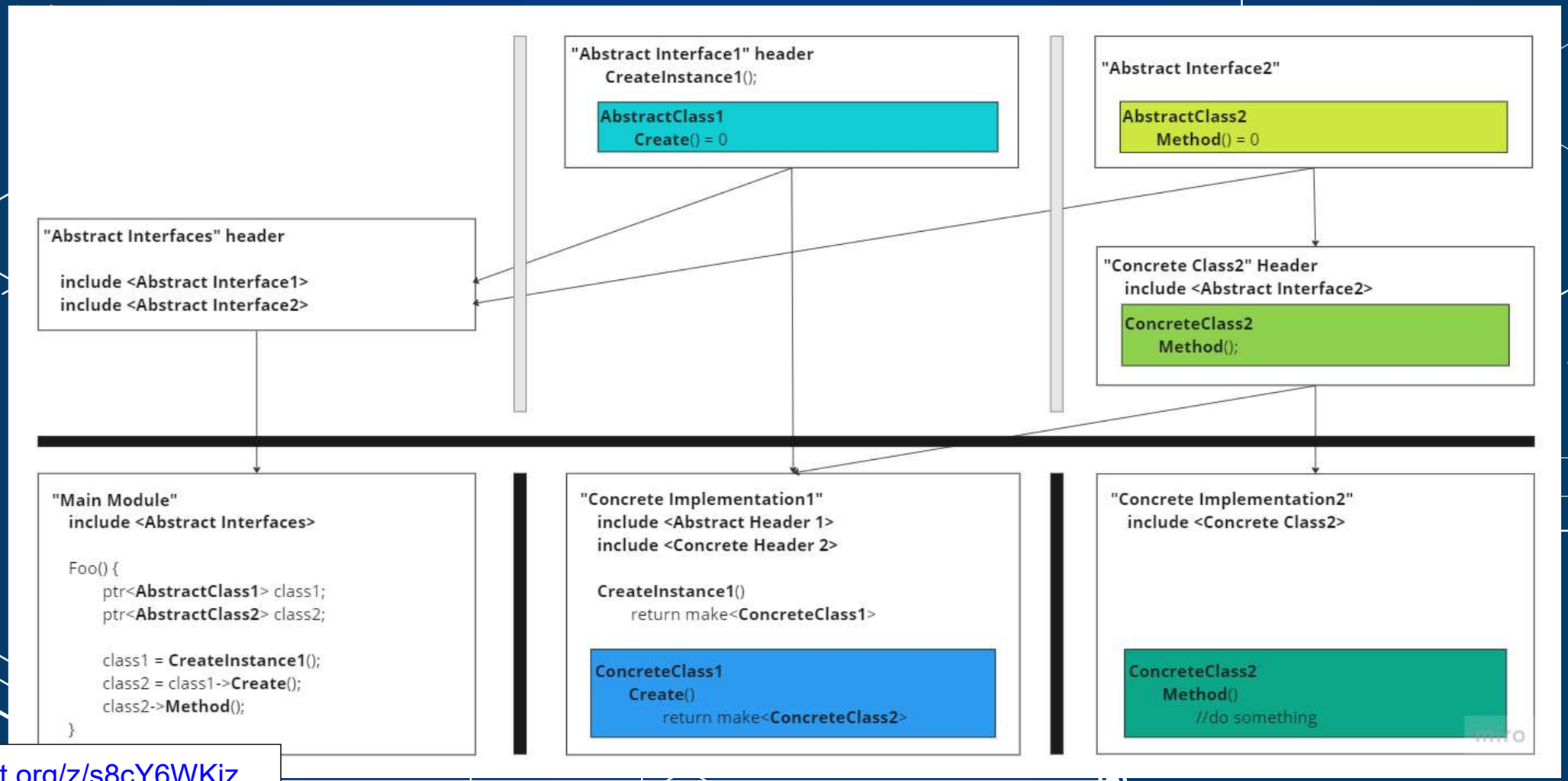
Design Patterns

```
"Main Module"

Foo() {
    ptr<AbstractClass1> class1;
    ptr<AbstractClass2> class2;

    class1 = make<ConcreteClass1>
    class2 = class1->Create();
    class2->Method();
}
```



godbolt.org/z/s8cY6WKjz

Design Patterns



godbolt.org/z/s8cY6WKjz

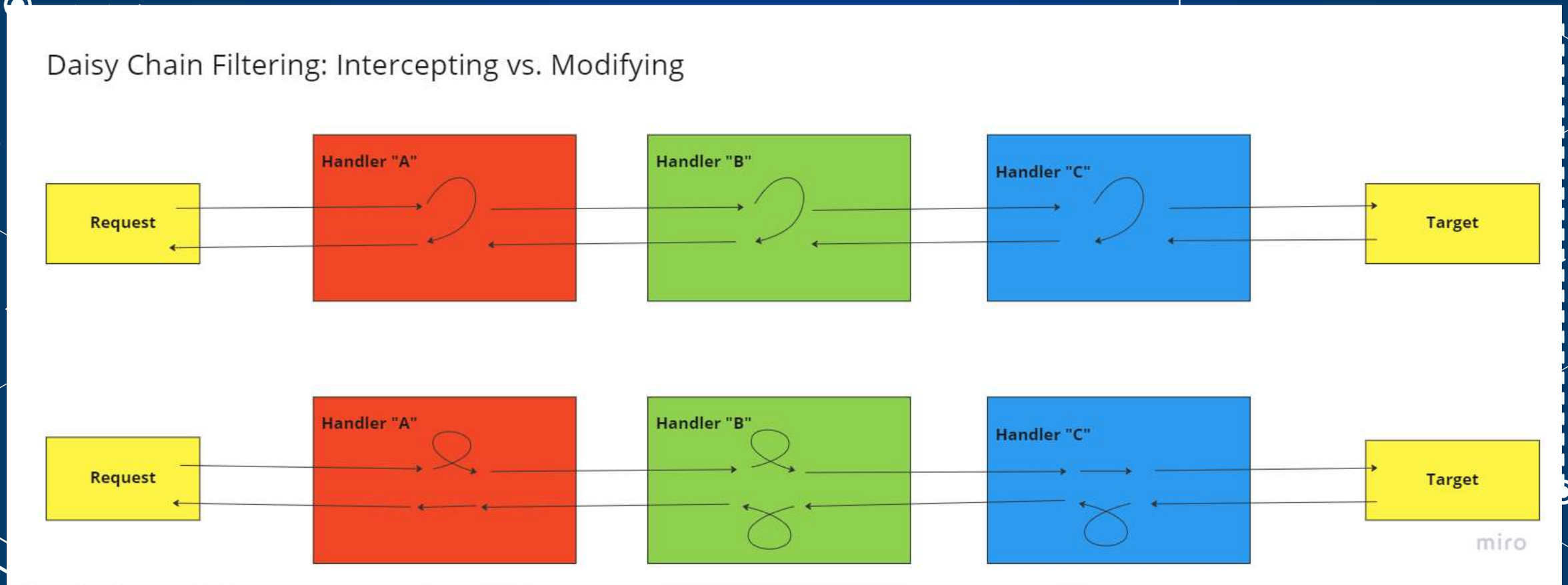
Design Patterns

```cpp
#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();
}
```

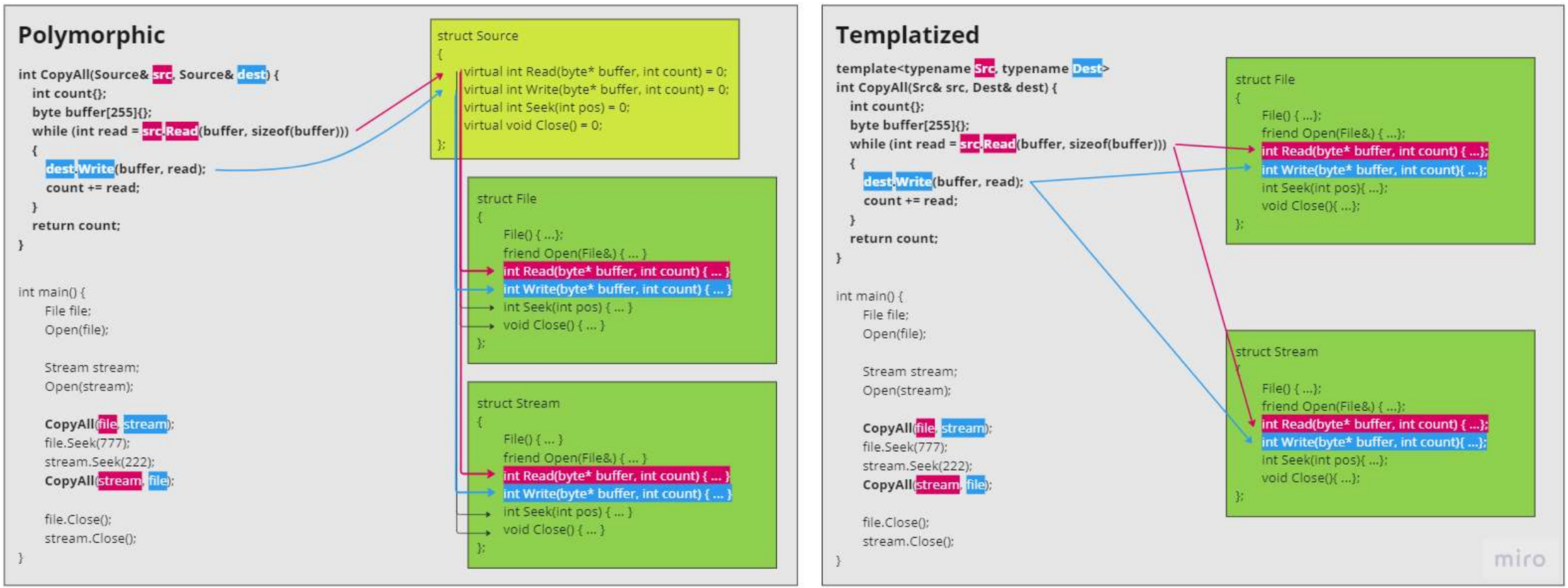
godbolt.org/z/s8cY6WKjz

Design Patterns

Daisy Chain Filtering: Intercepting vs. Modifying



Design Patterns-Dependency Injection



godbolt.org/z/5qE7Was48

godbolt.org/z/KTffoMv56

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

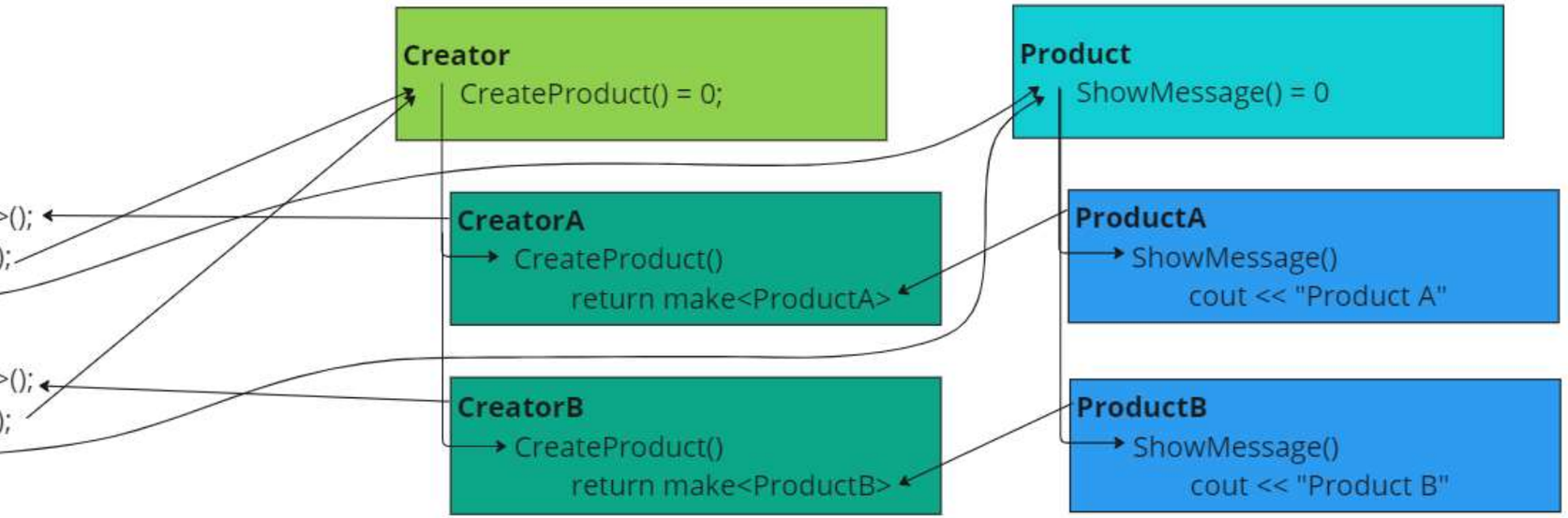
Focused on the process of object creation

- 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

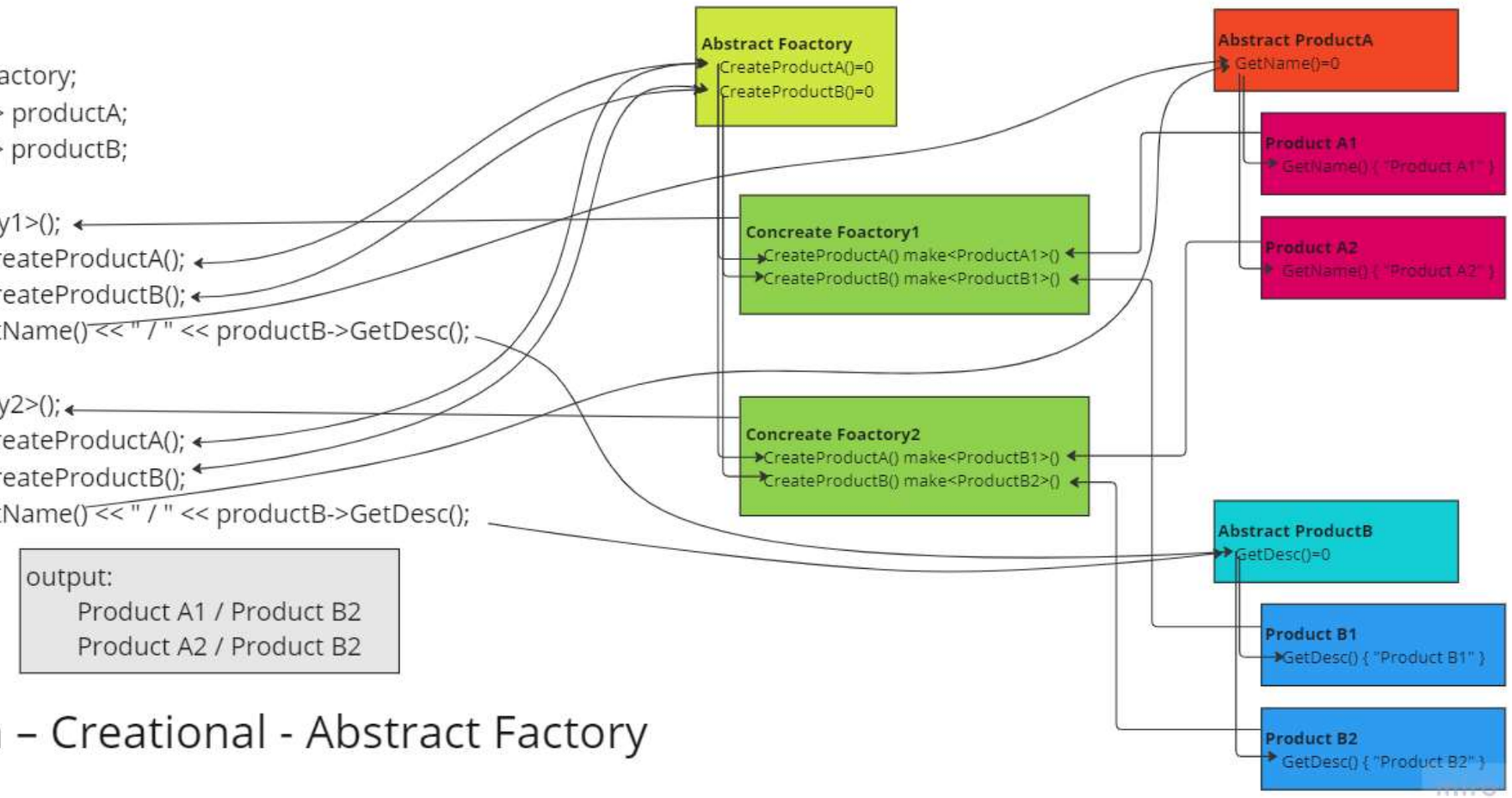
godbolt.org/z/bece3fEnj

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

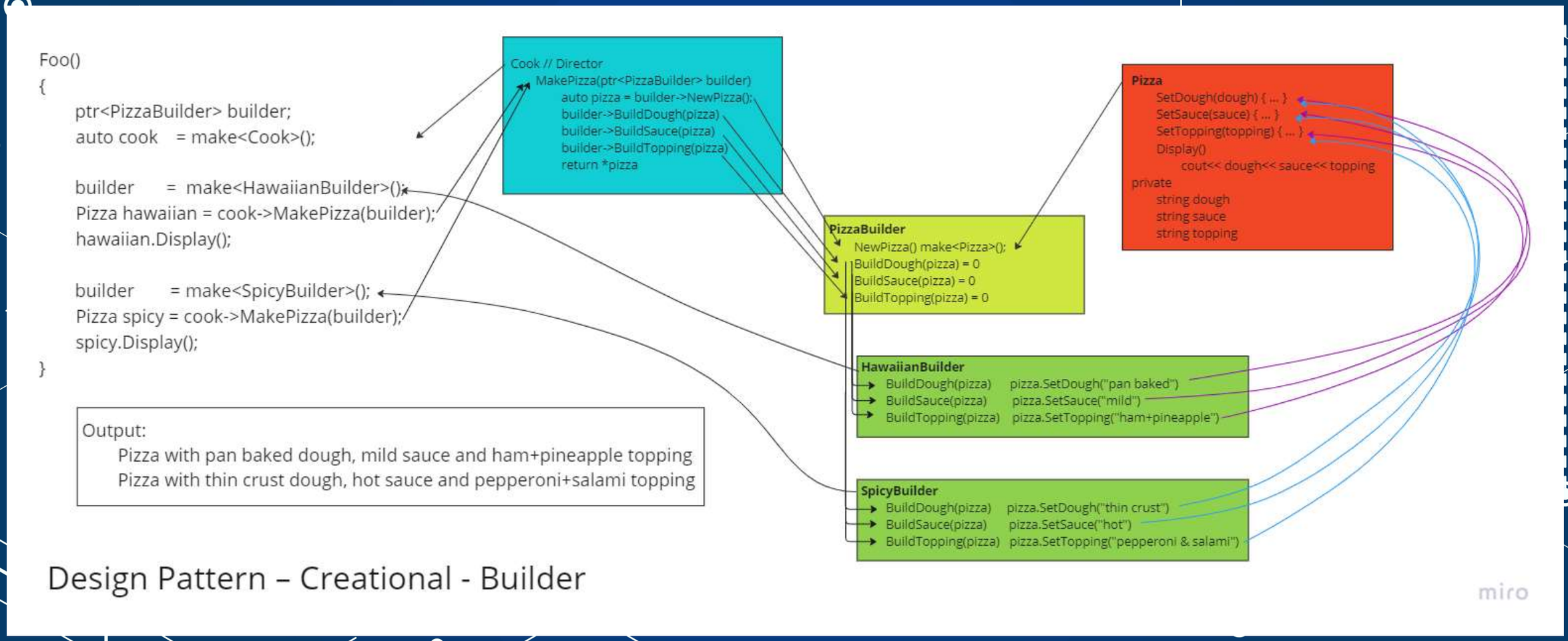


Design Pattern – Creational - Abstract Factory

godbolt.org/z/8WKzjTn6q

Design Patterns-Creational-Builder

Constructs objects by separating construction and representation



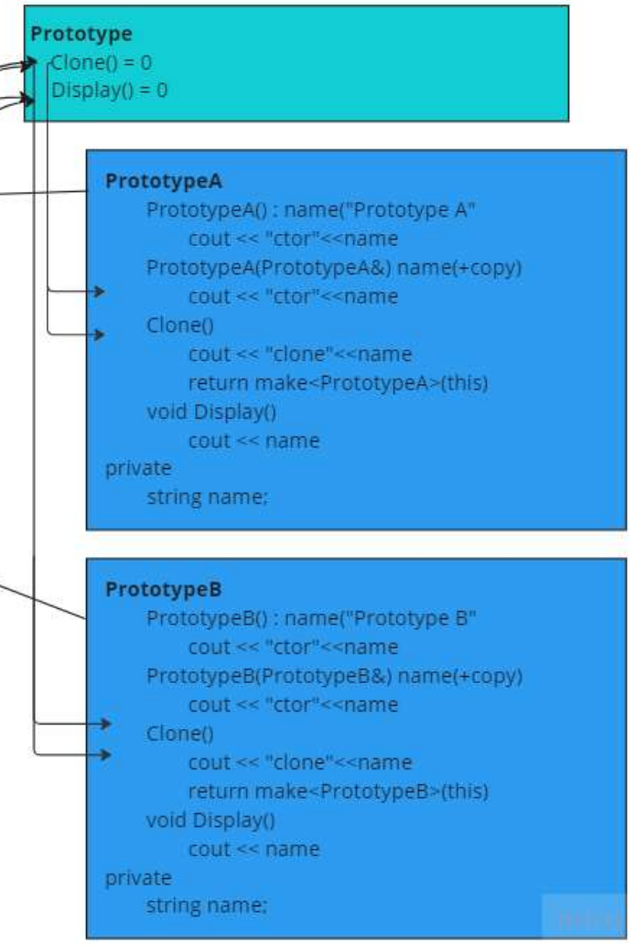
godbolt.org/z/frWrvenTj

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
```



Design Pattern – Creational - Prototype

godbolt.org/z/oqxbaWjb5

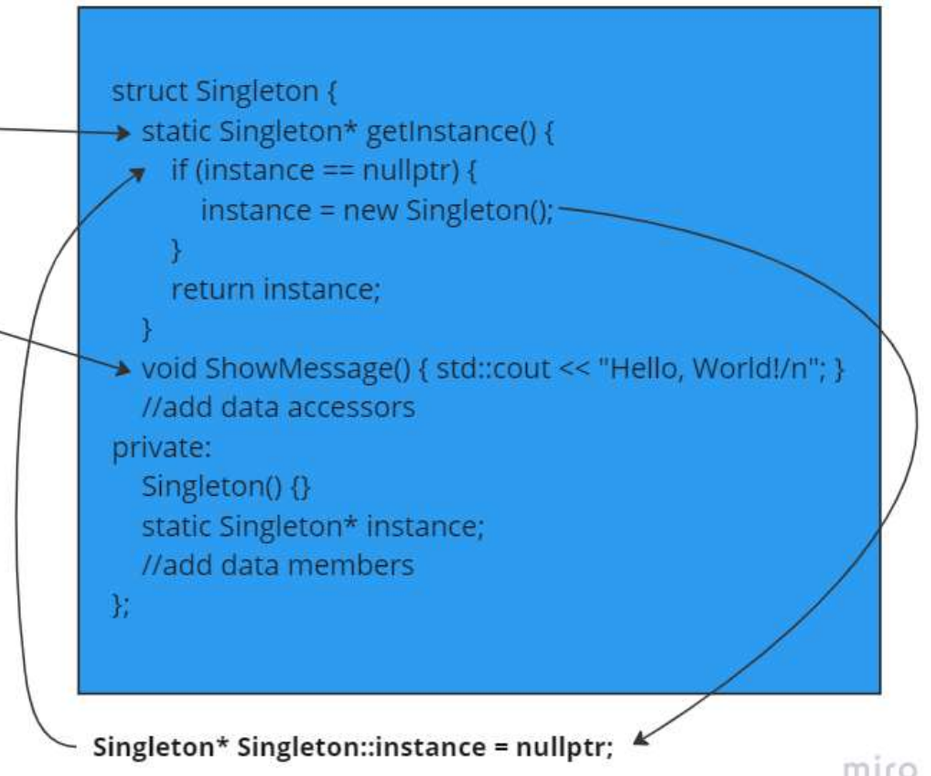
Design Patterns-Creational-Singleton (old)

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

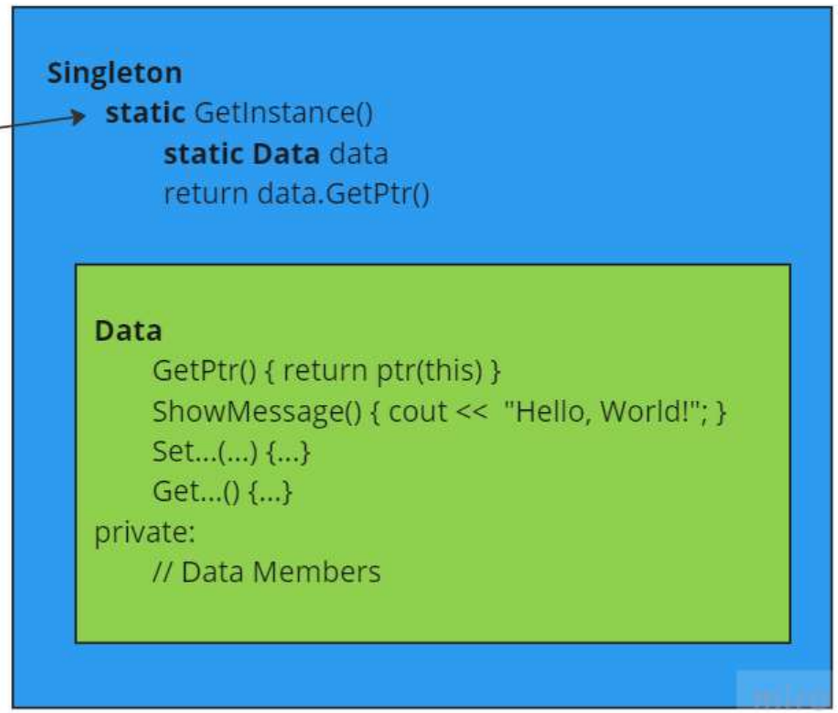


godbolt.org/z/3G85EzzT9

Design Patterns-Creational-Singleton

Restricts object creation to only one instance

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

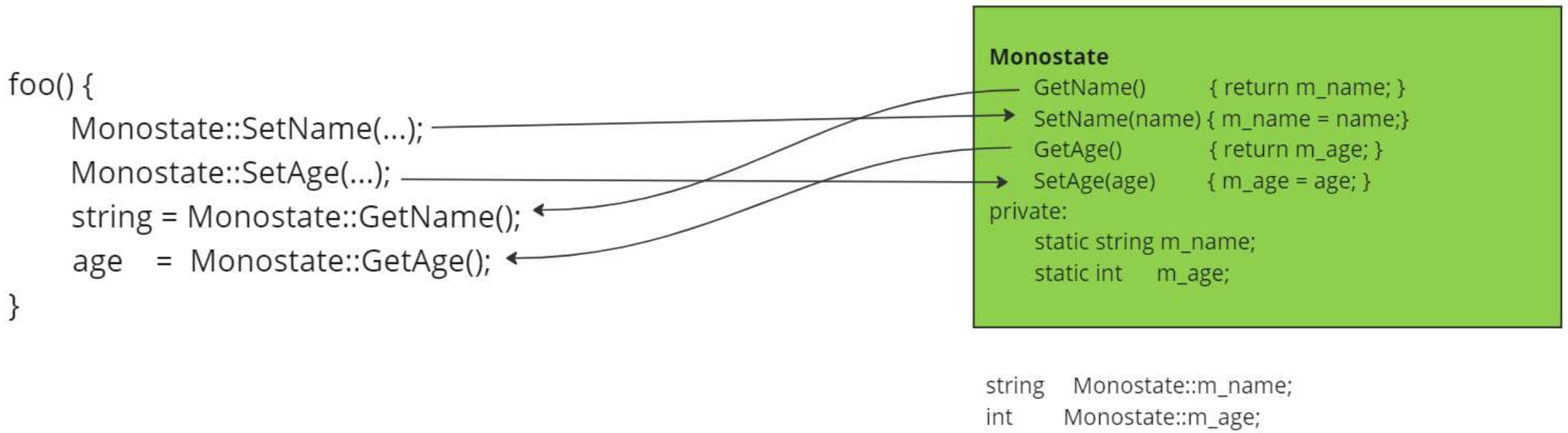


Design Pattern – Creational - Singleton

godbolt.org/z/76fnsj7Pz

Design Patterns-Creational-Monostate

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



godbolt.org/z/1TMj13z8x

Design Patterns-Tempetiezed Factory<>

```cpp
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;
    }
};
```

godbolt.org/z/WzoMYcMEY

Design Patterns-Tempetiezed Factory<>

```cpp
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!

godbolt.org/z/WzoMYcMEY

Design Patterns-Tempetiezed Factory<>

```cpp
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

godbolt.org/z/WzoMYcMEY

Design Patterns-Tempetiezed Factory<>

```cpp
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;
    }
};
```

godbolt.org/z/WzoMYcMEY

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:

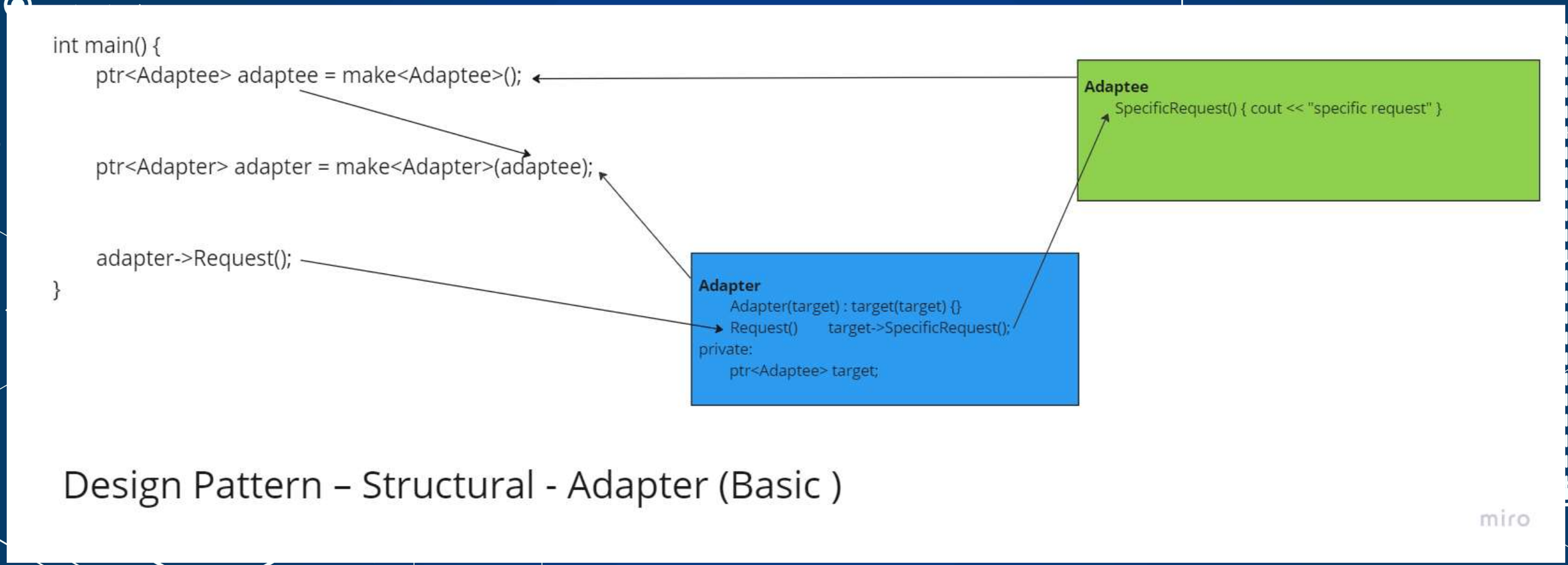
- Abstraction
- Flexibility
- Reduced coupling
- Separation of algorithms from the structure

Used for Improved:

- Extensibility
- Flexibility
- Performance
- Security

Design Patterns-Structural-Adapter(Basic)

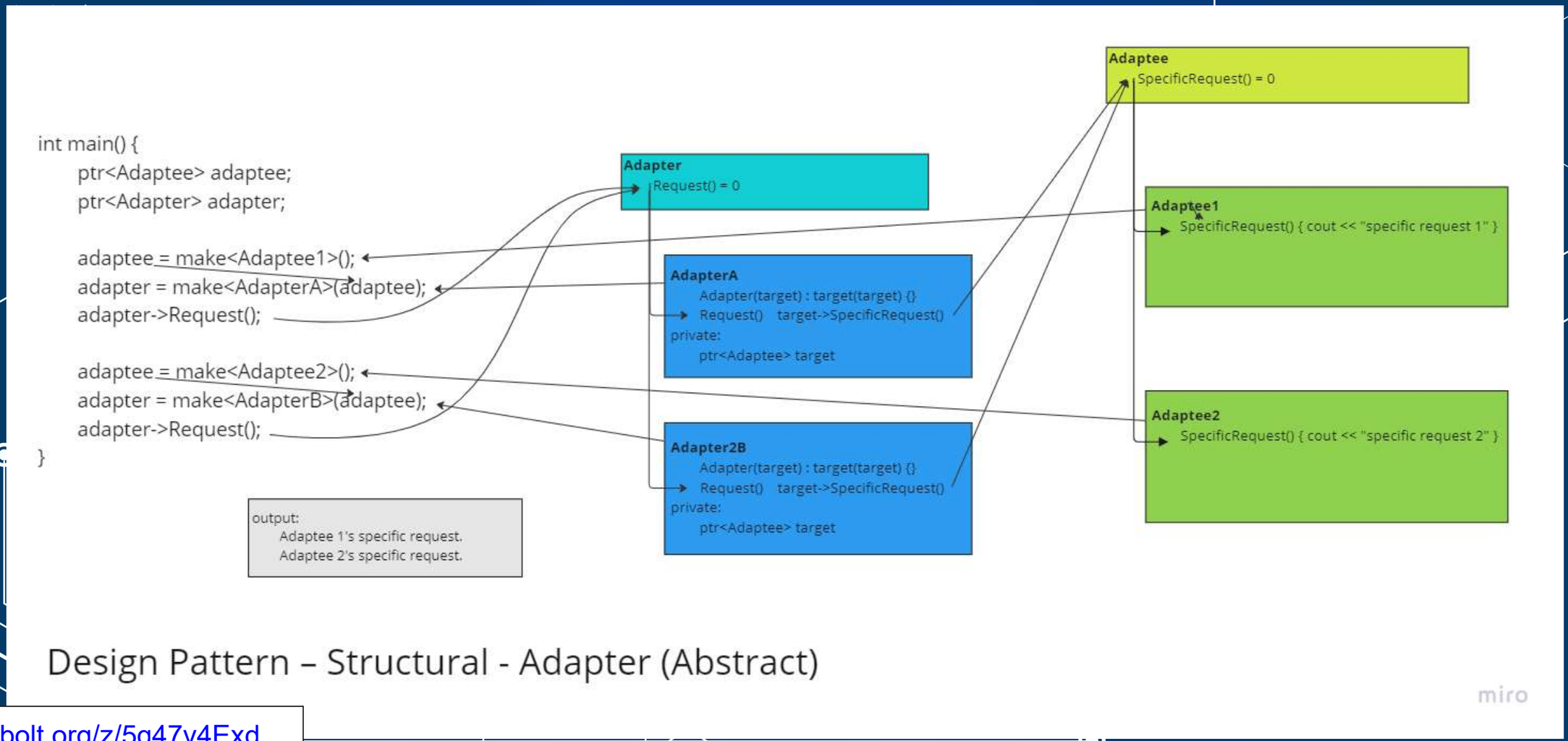
An intermediary translating interface



godbolt.org/z/Geccqhf19

Design Patterns-Structural-Adapter(Abstract)

An intermediary translating interface



godbolt.org/z/5q47v4Exd

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

godbolt.org/z/fEhroGsPx

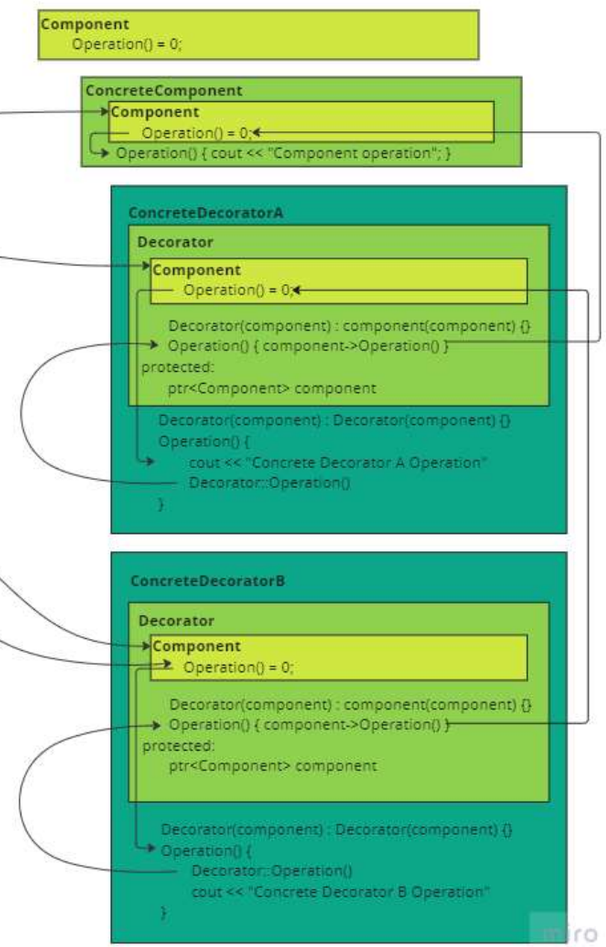
Design Patterns-Structural-Decorator

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

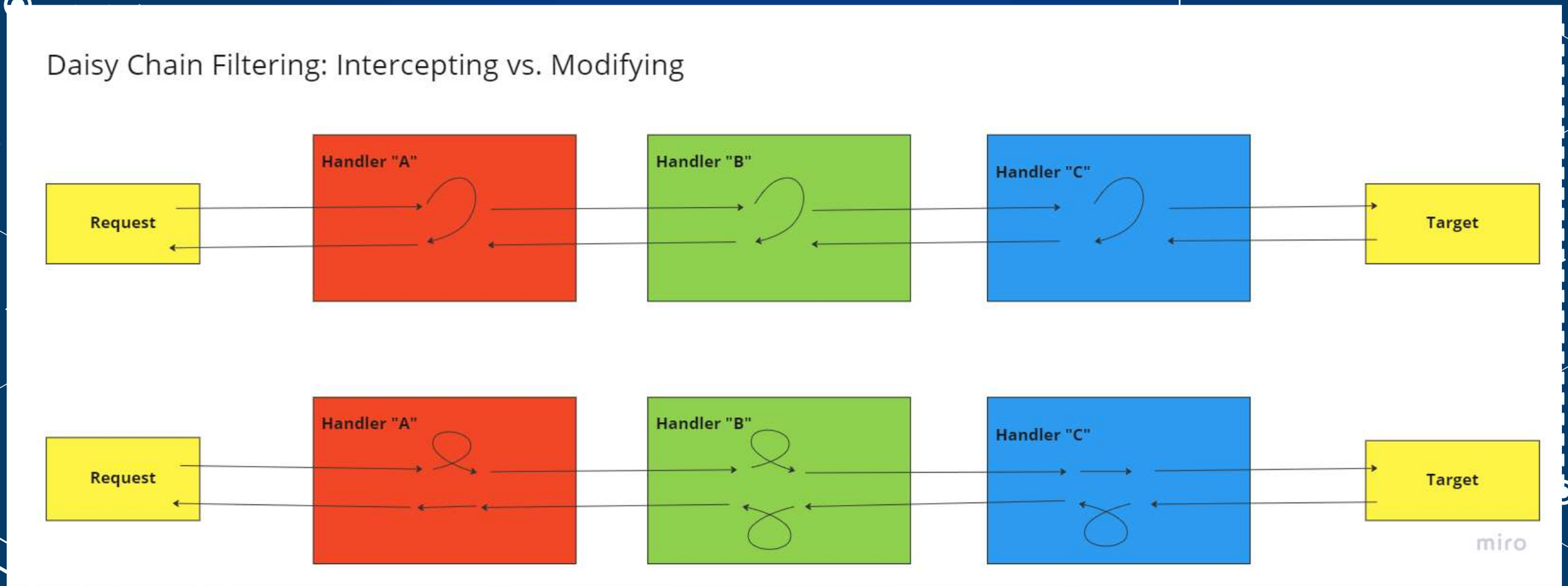


Design Pattern – Structural - Decorator

godbolt.org/z/fEhroGsPx

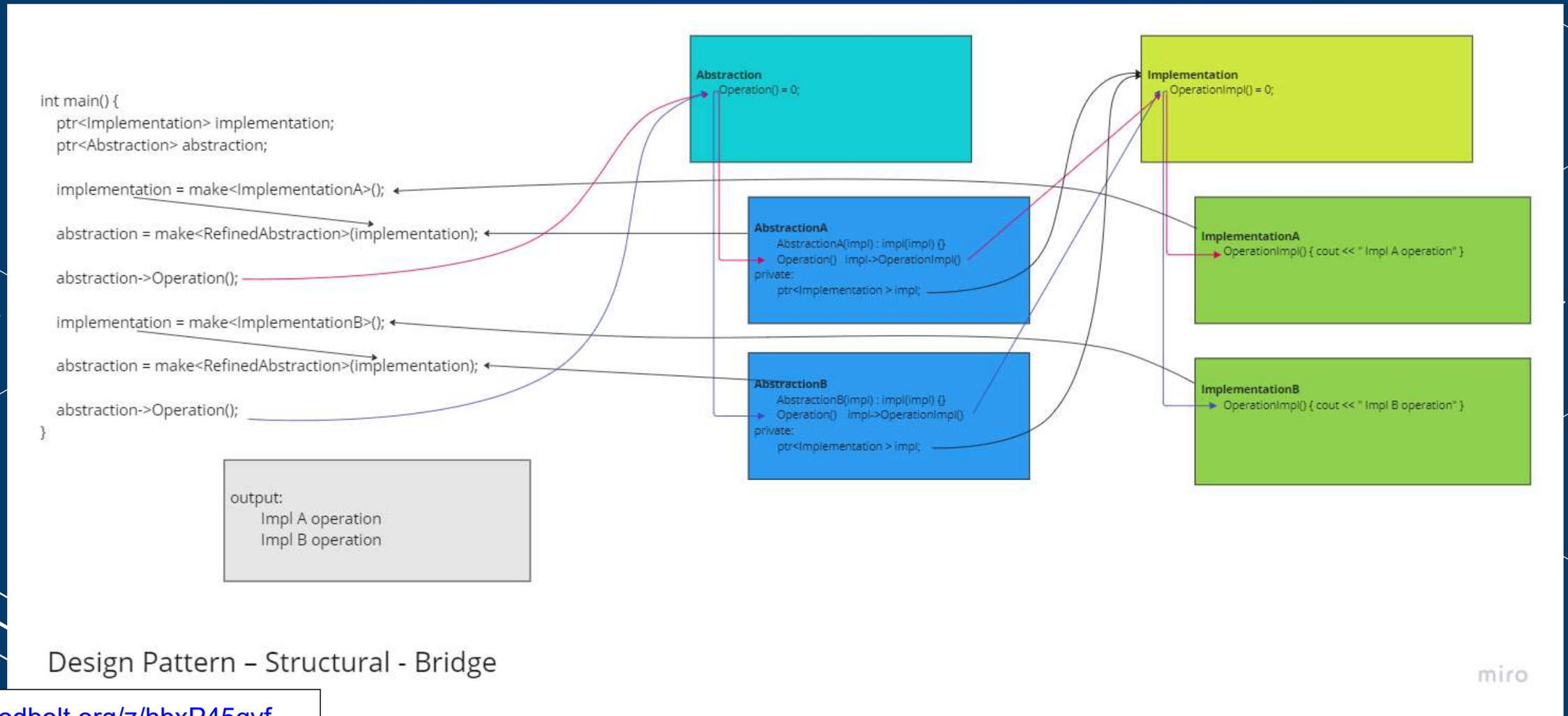
Design Patterns

Daisy Chain Filtering: Intercepting vs. Modifying



Design Patterns-Structural-Bridge

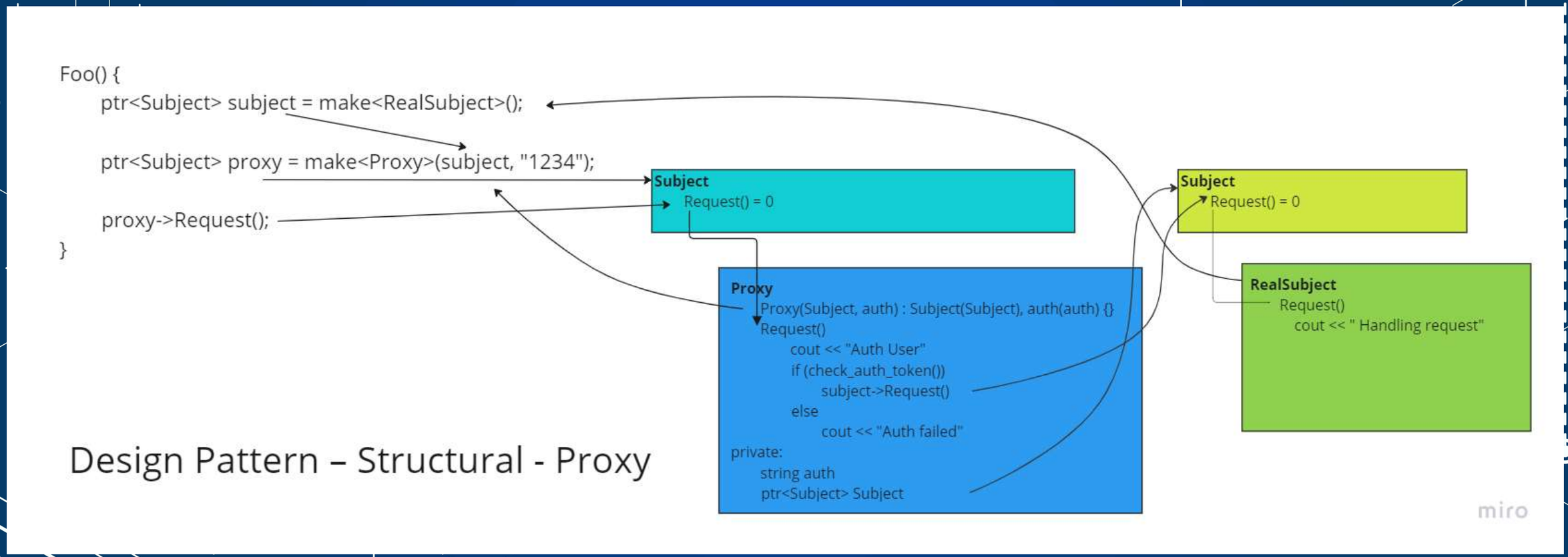
Decouples an abstraction from its implementation



godbolt.org/z/hbxP45qvf

Design Patterns-Structural-Proxy

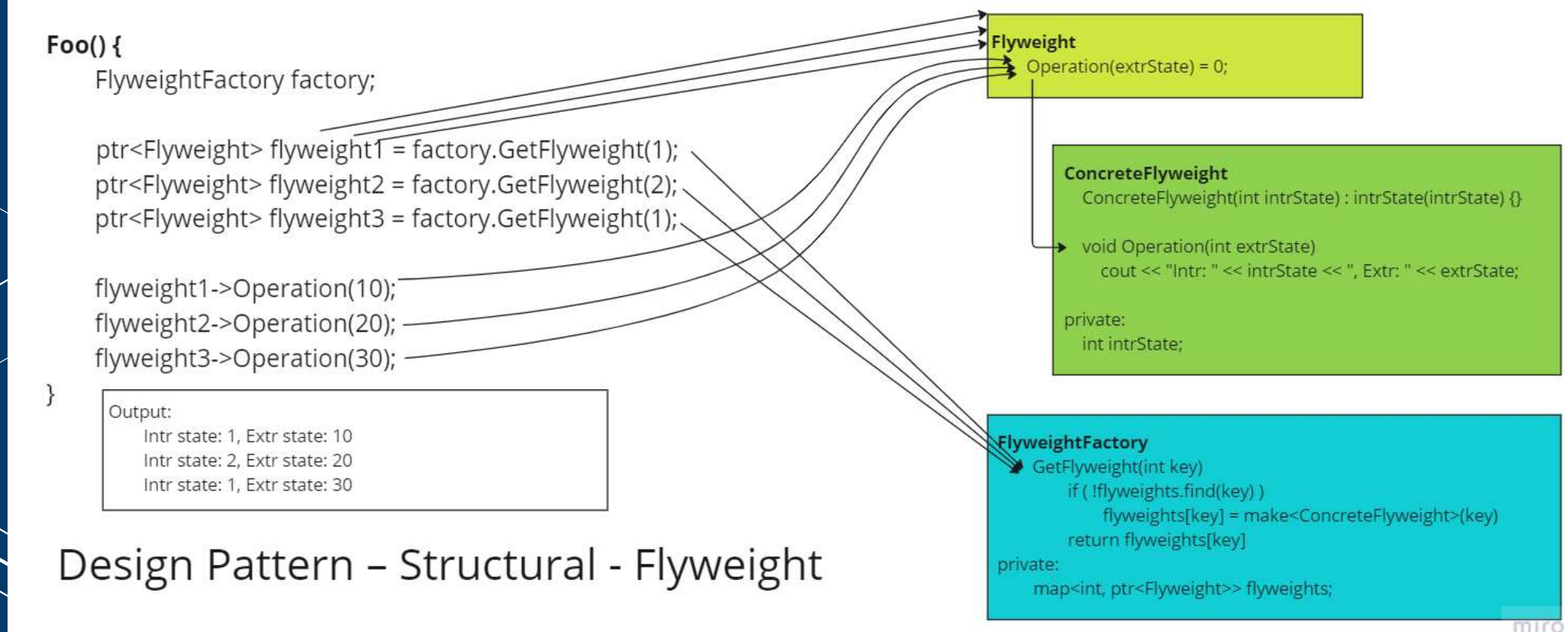
A placeholder for another object to control access, reduce complexity



godbolt.org/z/fG9hnTbPx

Design Patterns-Structural-Flyweight

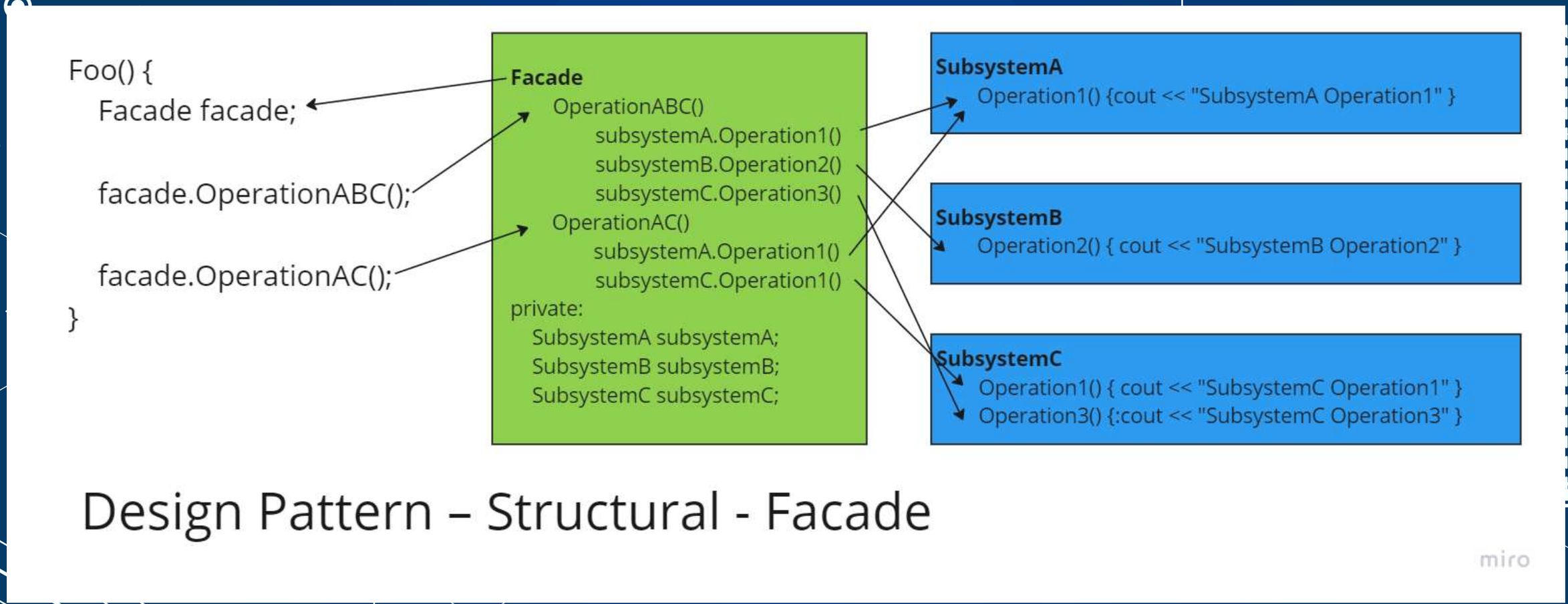
Reduces the cost of creating and manipulating of similar objects



godbolt.org/z/4cMndhn3r

Design Patterns-Structural-Facade

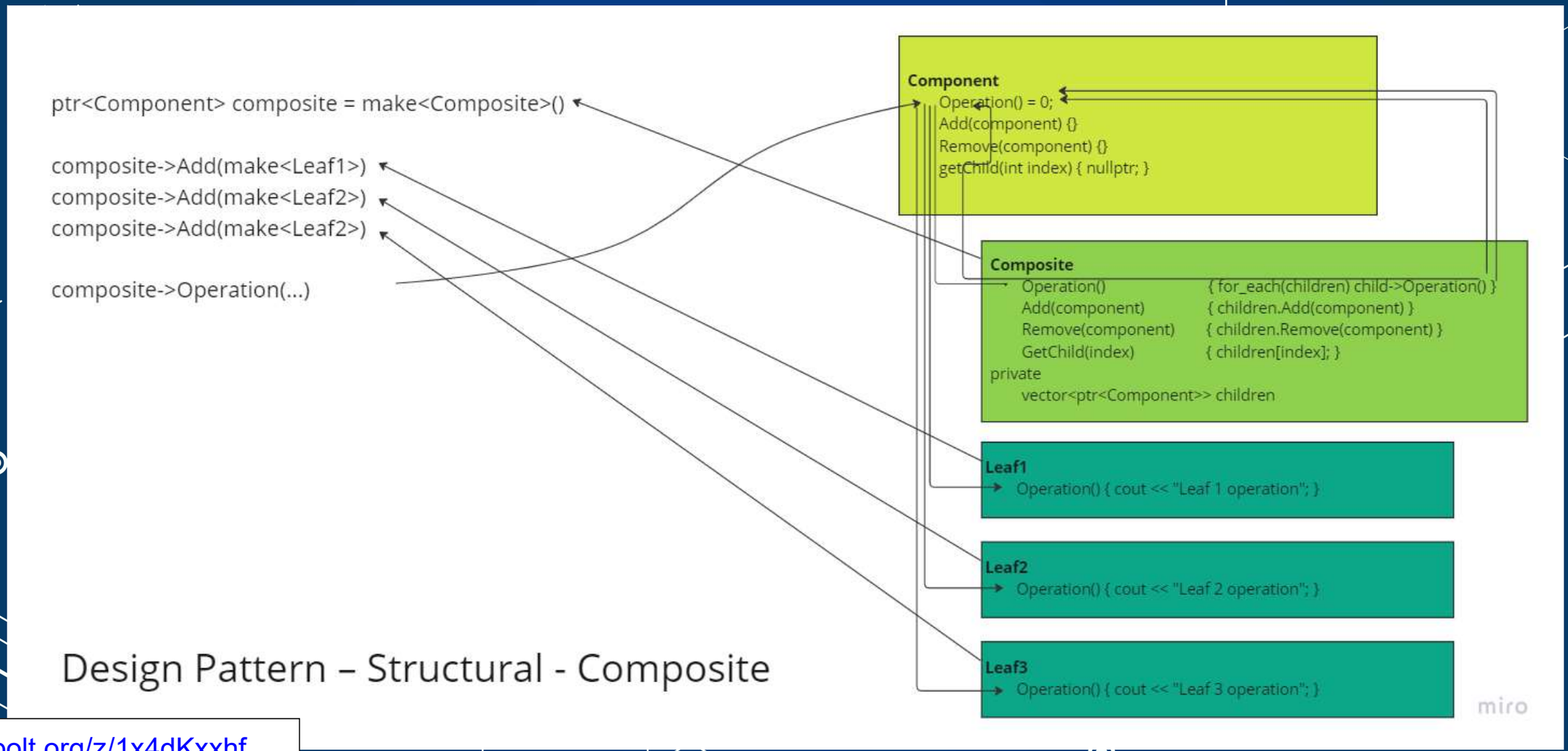
Provides a simplified interface to a larger body of code



godbolt.org/z/44M71anMK

Design Patterns-Structural-Composite

Dynamic collection of objects manipulated as one object



godbolt.org/z/1x4dKxxhf

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.

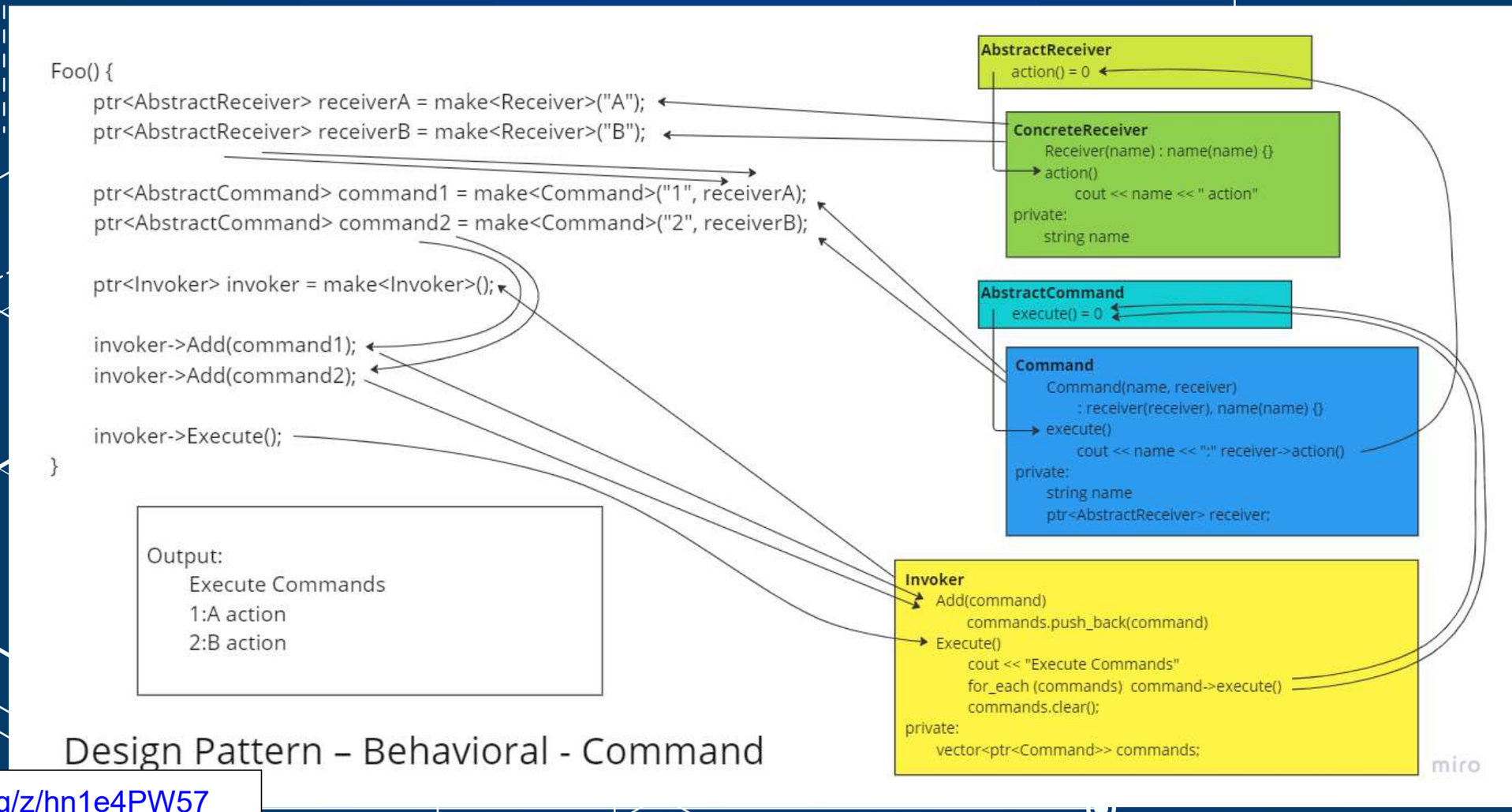
Design Patterns-Behavioral

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

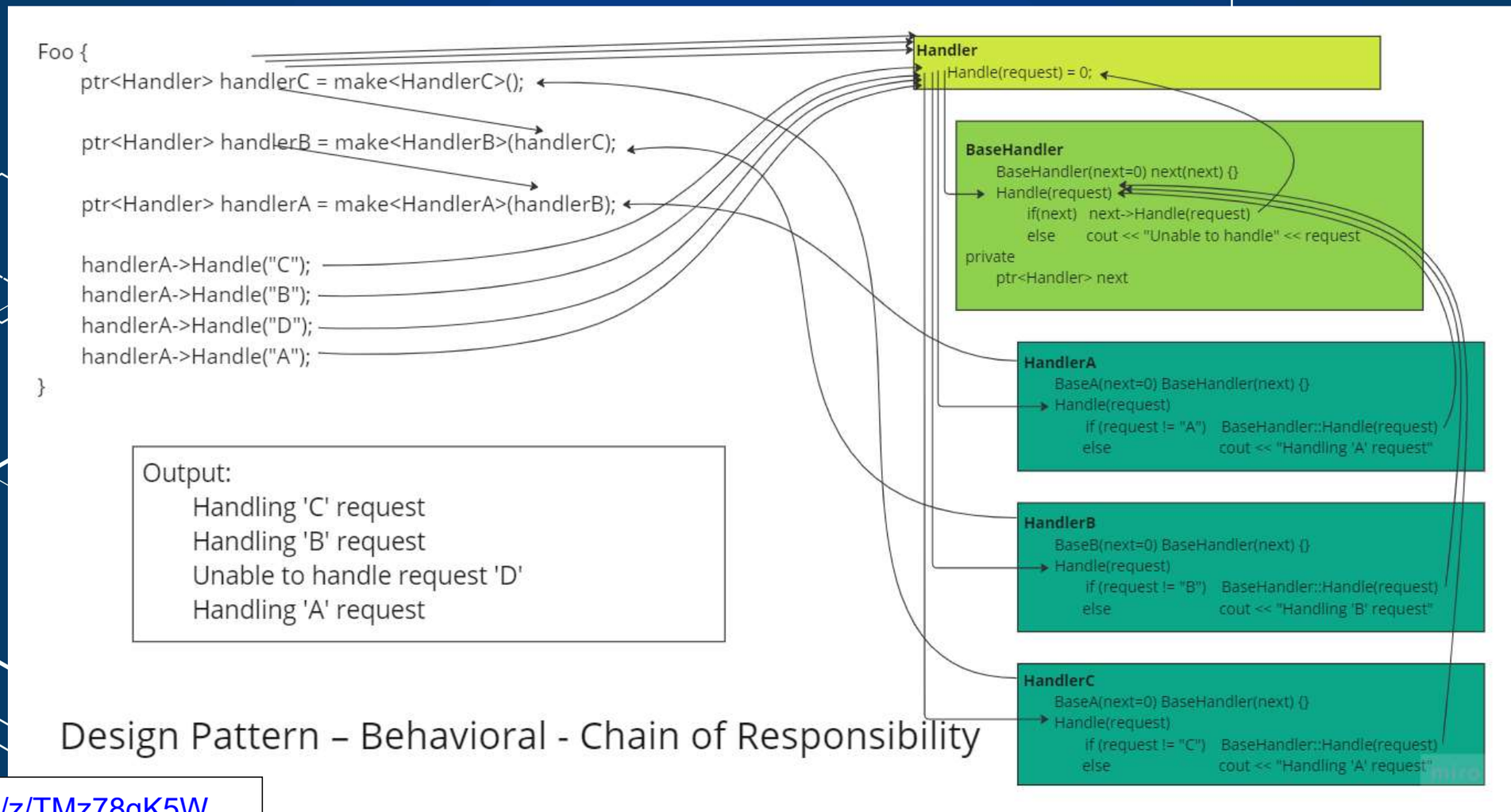
Batching of multiple actions and parameters until executed (Transaction Caching)



godbolt.org/z/hn1e4PW57

Design Patterns-Behavioral-Chain of Responsibility

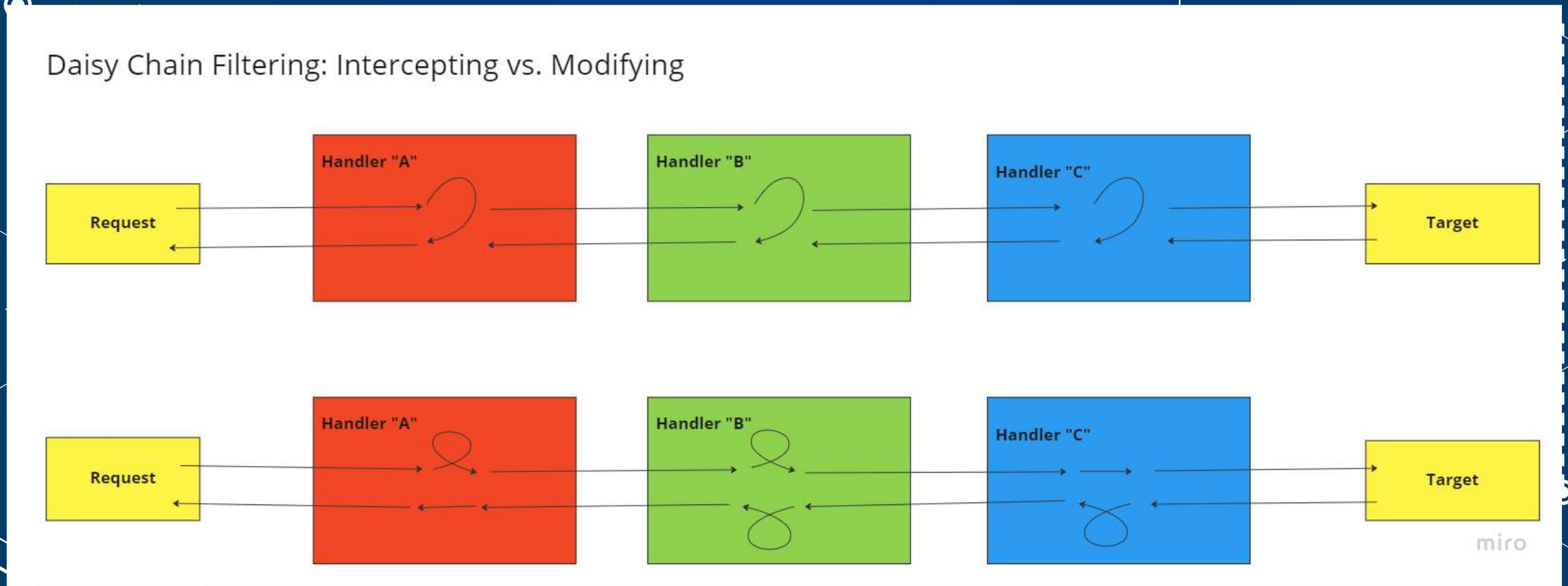
Delegates commands through cascade processing



godbolt.org/z/TMz78qK5W

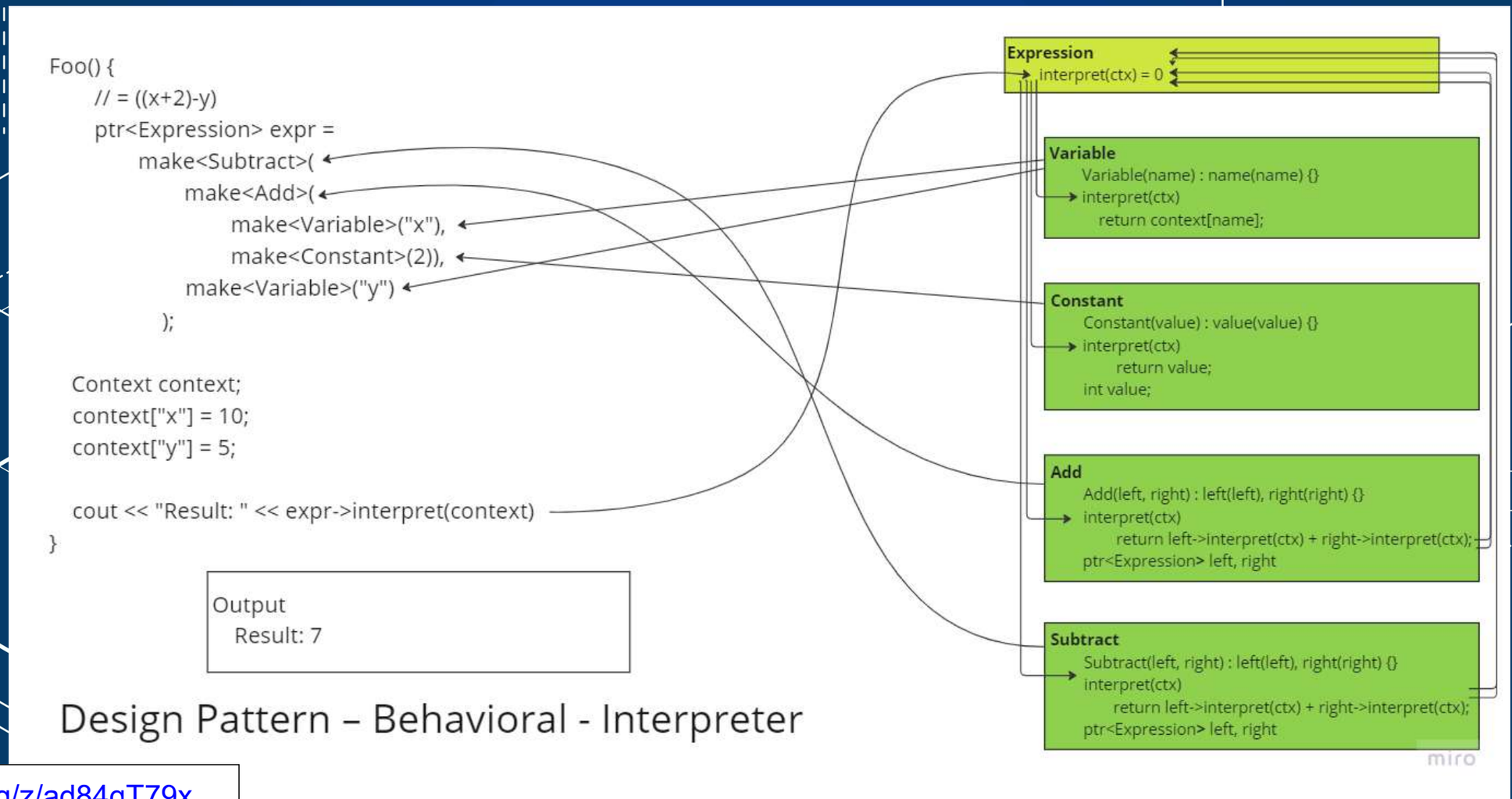
Design Patterns

Daisy Chain Filtering: Intercepting vs. Modifying



Design Patterns-Behavioral-Interpreter

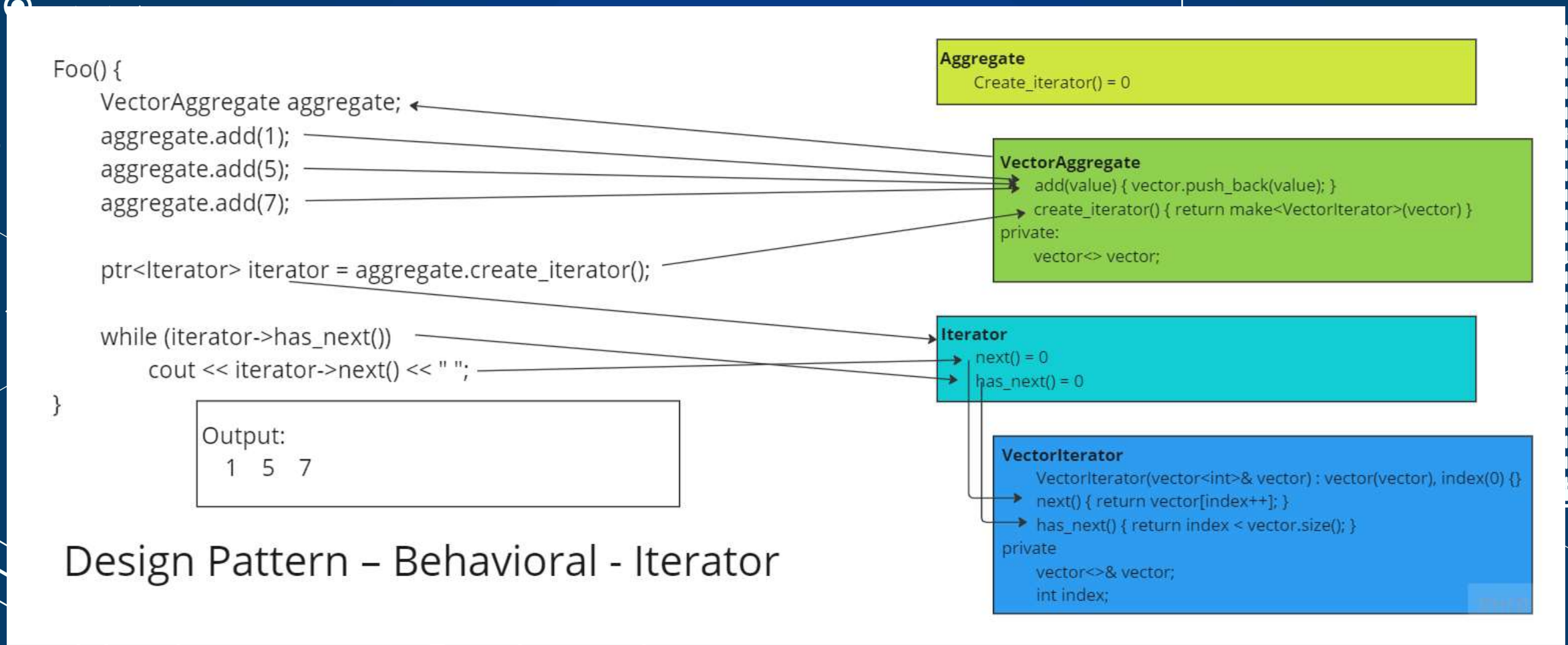
Implements a specialized language



godbolt.org/z/ad84qT79x

Design Patterns-Behavioral-Iterator

Access to the elements of an object hiding its implementation



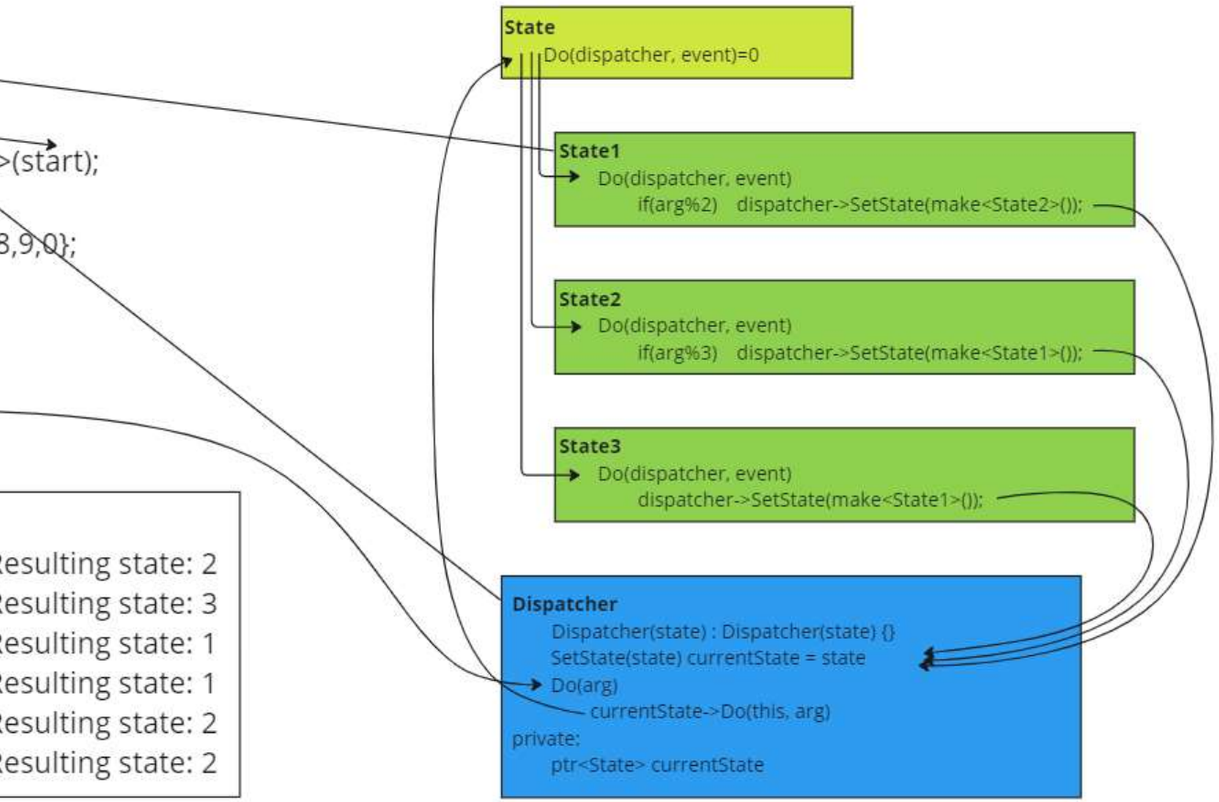
godbolt.org/z/e978j3v6G

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
```



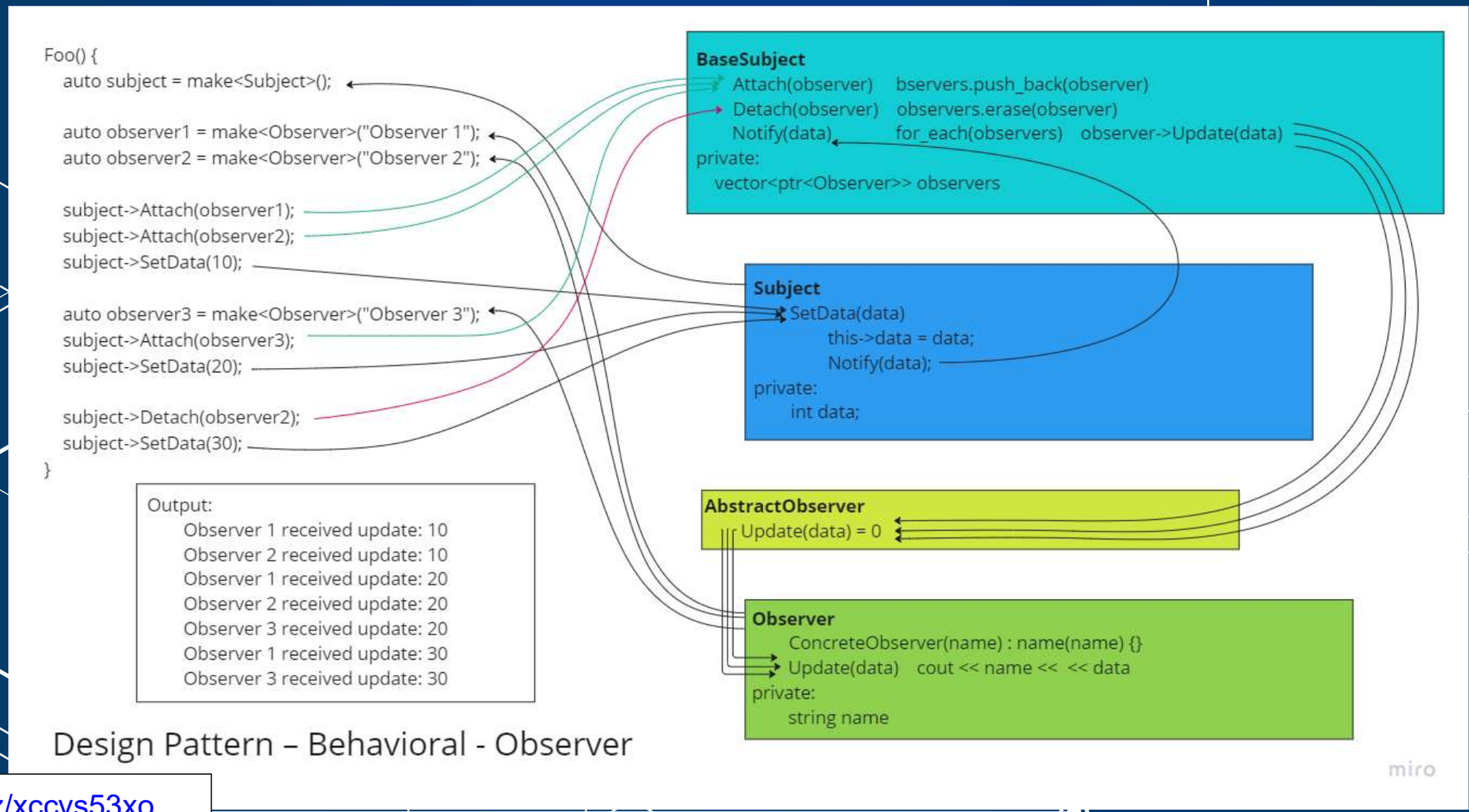
Design Patterns – Behavioral - State

miro

godbolt.org/z/97qdP9zf3

Design Patterns-Behavioral-Observer

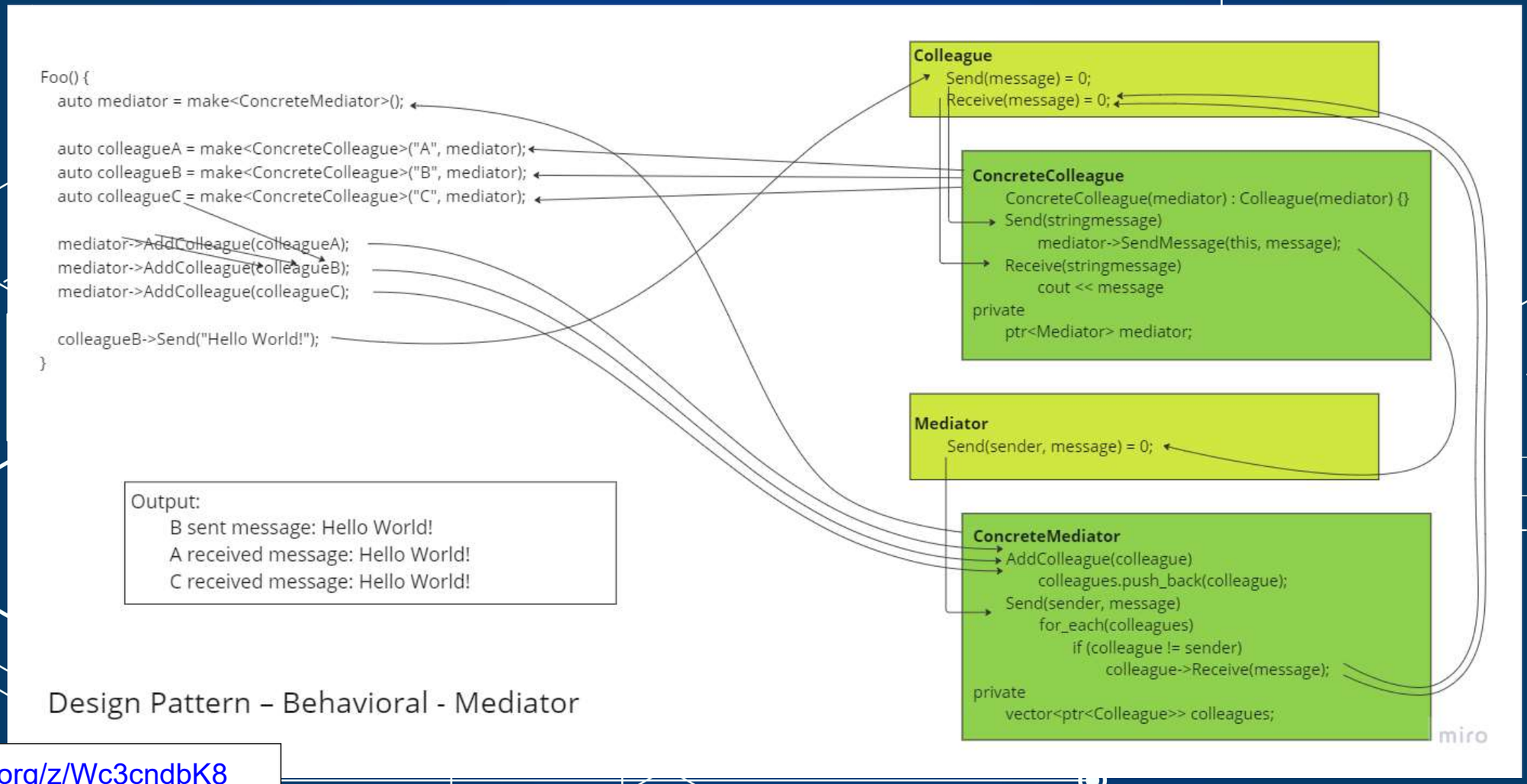
A publish/subscribe, all observer objects see event notifications



godbolt.org/z/xccvs53xo

Design Patterns-Behavioral-Mediator

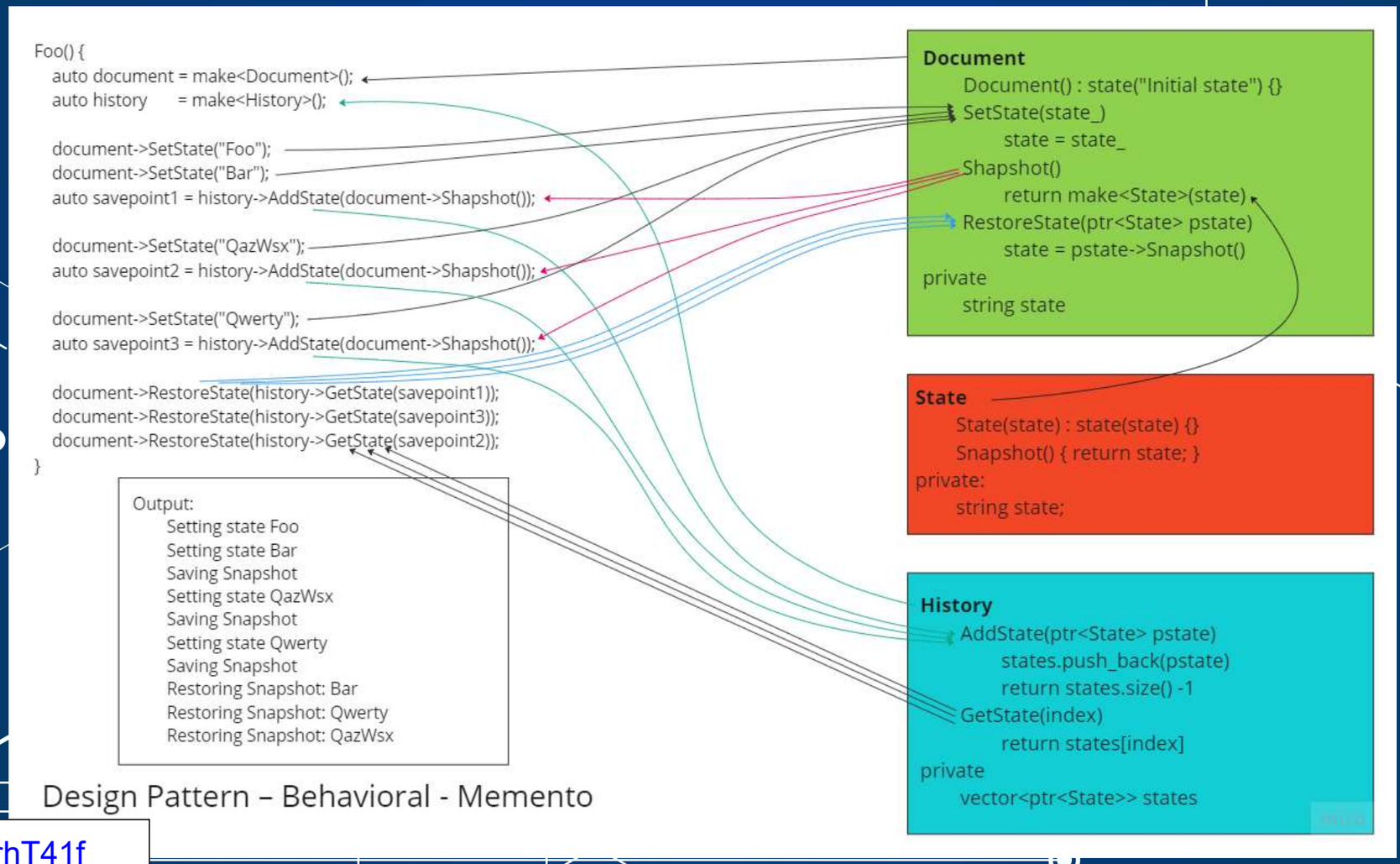
Like the Observer pattern but excludes the notifying the sender



godbolt.org/z/Wc3cndbK8

Design Patterns-Behavioral-Memento

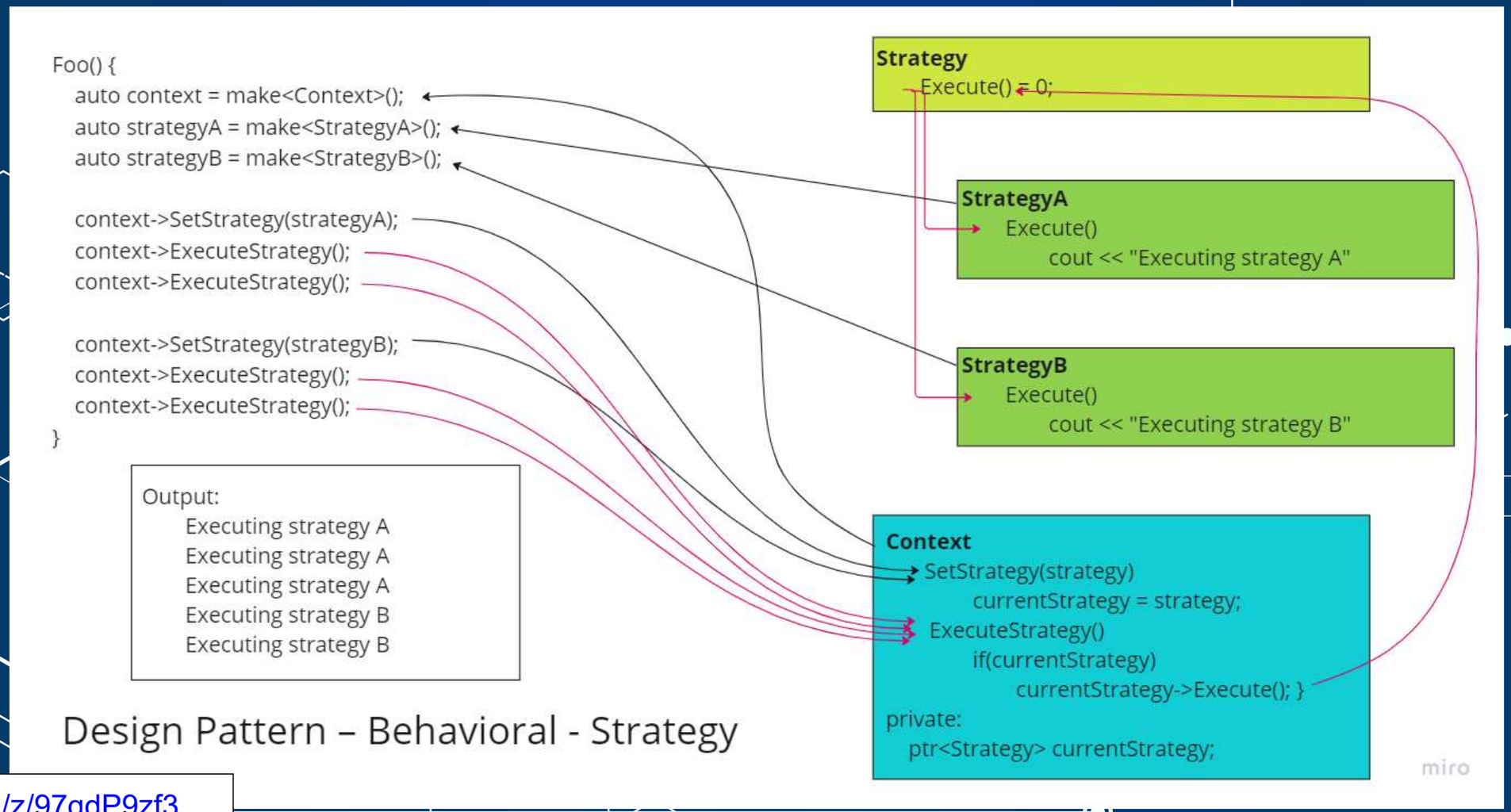
Provides the ability to restore an object state (undo)



godbolt.org/z/64nrhT41f

Design Patterns-Behavioral-Strategy

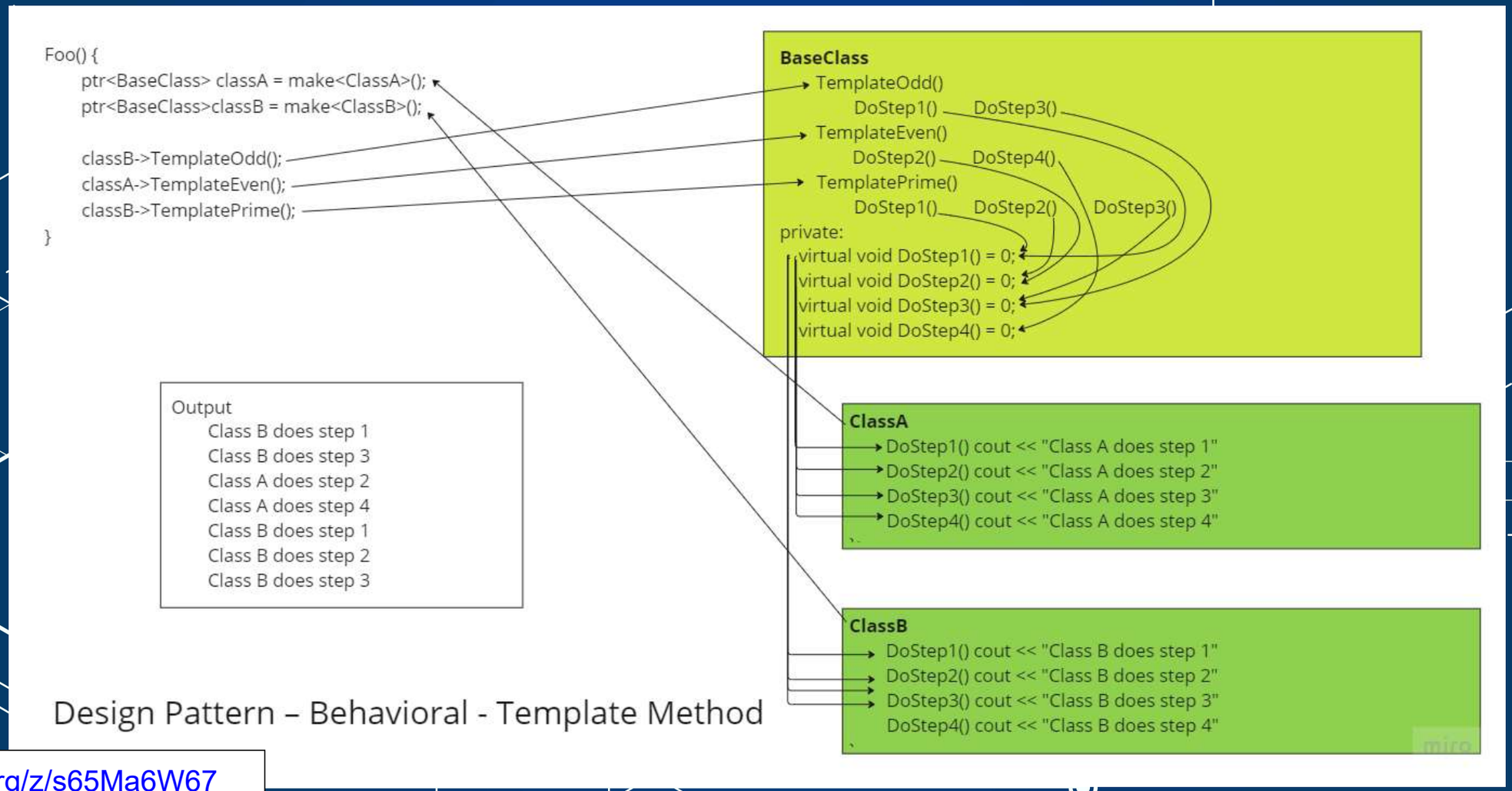
Allows switchable algorithms on-the-fly at runtime



godbolt.org/z/97qdP9zf3

Design Patterns-Behavioral-Template Method

Extends skeleton functionality by providing concrete behavior



godbolt.org/z/s65Ma6W67

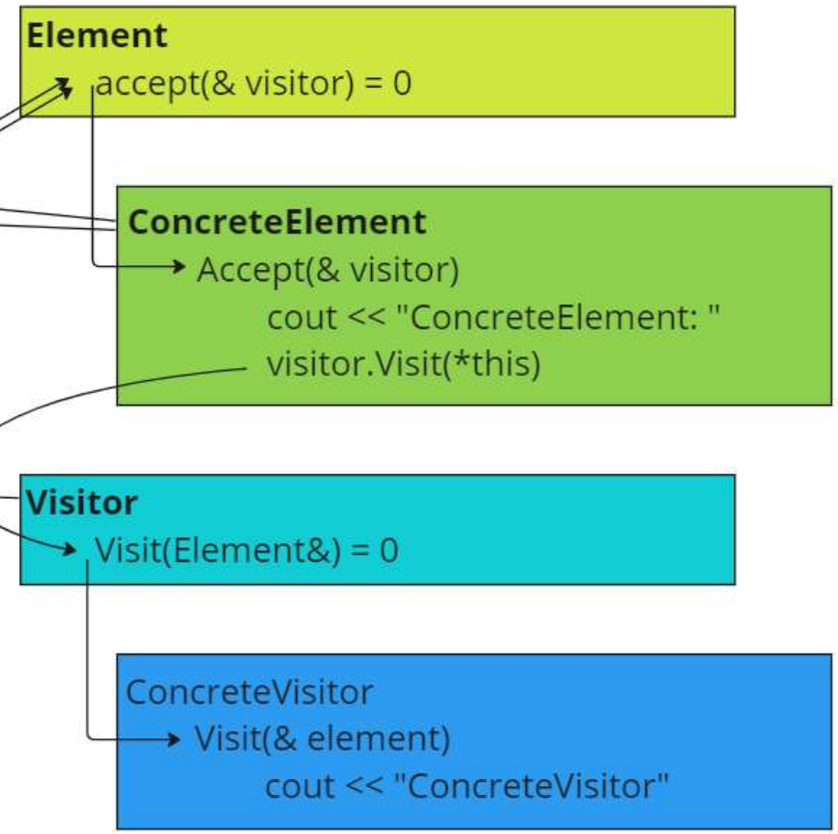
Design Patterns-Behavioral-Visitor

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



Design Pattern – Behavioral - Visitor

miro

godbolt.org/z/EK6vj7jnf

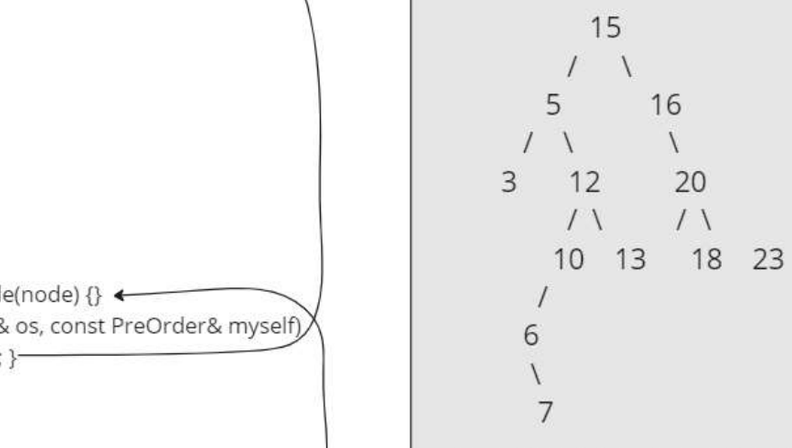
Design Patterns-Behavioral-Visitor

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:

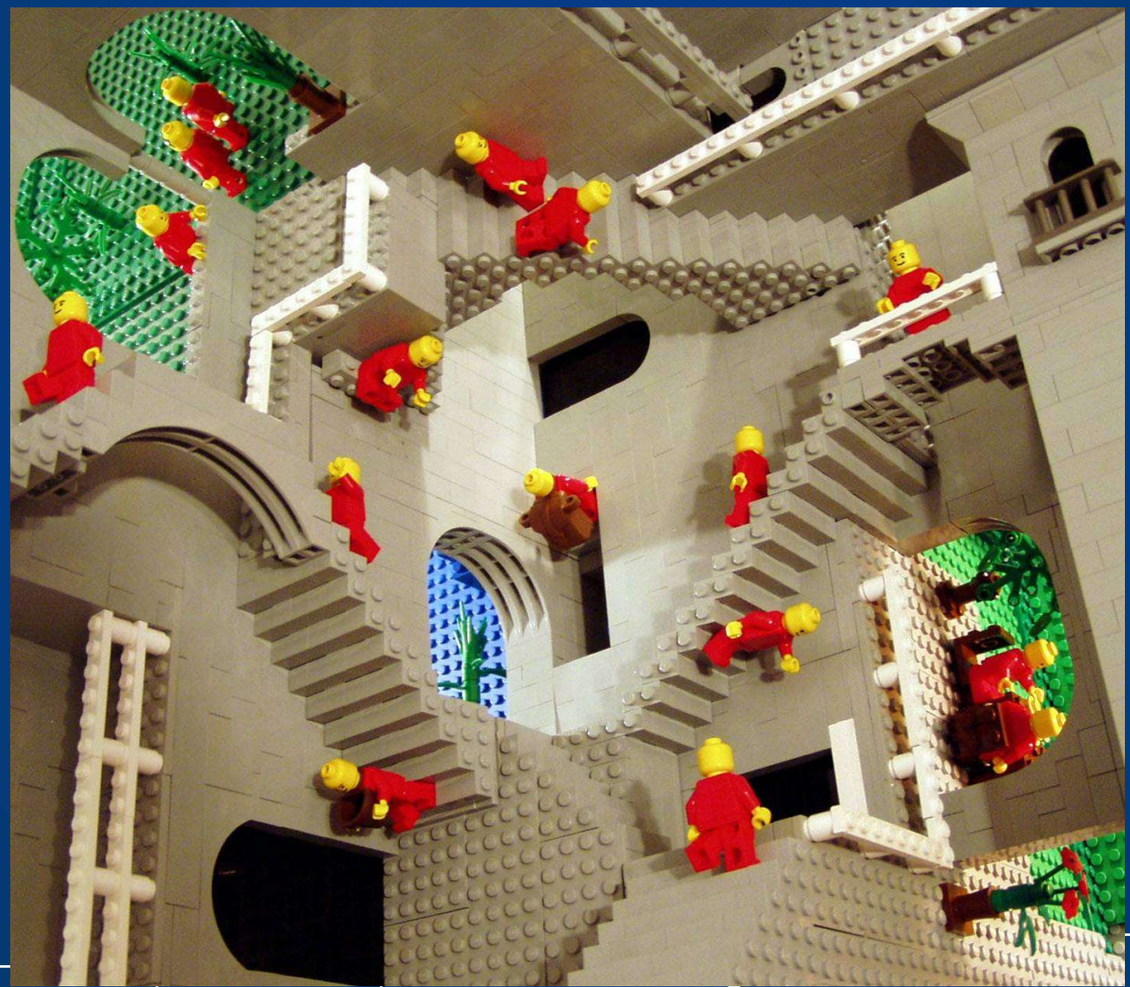
Given post-order(lrm) & in-order(lmr)
Rebuild tree
List nodes in pre-order(mlr)



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,

godbolt.org/z/Pv7jd5cG8

Design Patterns



Design Patterns