NODE Technical Book Club

C++ Software Design - Klaus Iglberger

G20: Favor Composition Over Inheritance

> Inheritance is the base class of evil.

- Inheritance is a powerful feature but it is hard to use it properly.
- It is overused and misused.
  - Design patterns mostly make use of composition.

G21: Use Command to Isolate What Things are Done

Intent of the **Command pattern**: Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

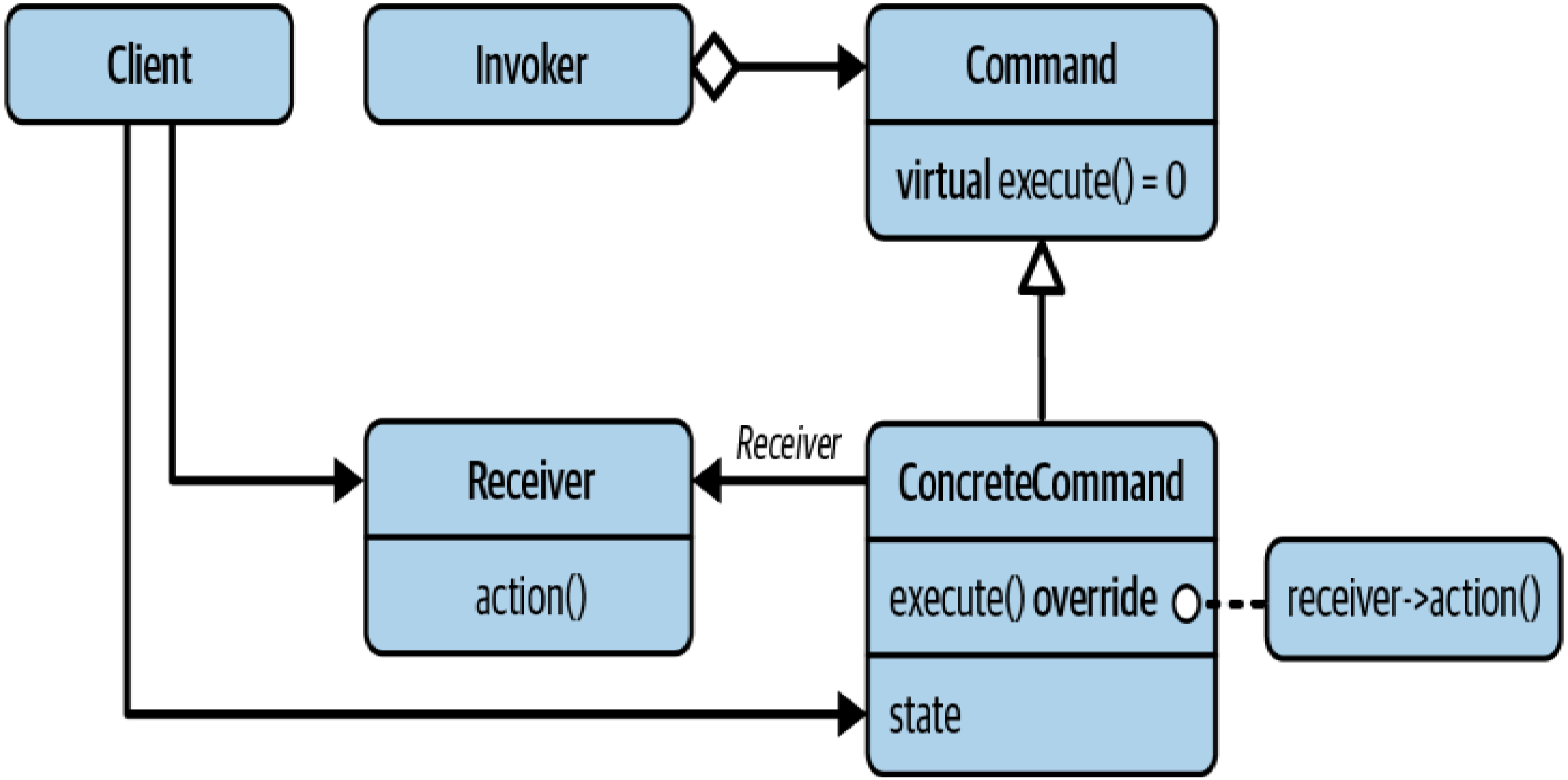


Figure 5-7. The UML representation of the Command design pattern

Example: Calculator

```
class CalculatorCommand
{
 public:
  virtual ~CalculatorCommand() = default;

  virtual int execute(int i) const = 0;
  virtual int undo(int i) const = 0;
};
```

```cpp
class Add : public CalculatorCommand
{
 public:
  explicit Add(int operand) : operand_(operand) {}

  int execute(int i) const override
    {
      return i + operand_;
    }
  int undo(int i) const override
    {
      return i - operand_;
    }

 private:
  int operand_{};
};
```

Add: Concrete command for addition.

```cpp
class Calculator
{
 public:
  void compute(std::unique_ptr<CalculatorCommand> command){
    current_ = command->execute( current_ );
    stack_.push( std::move(command) );
  }
  void undoLast(){
    if( stack_.empty() ) return;
    auto command = std::move(stack_.top());
    stack_.pop();
    current_ = command->undo(current_);
  }

  int result() const { return current_; }
  void clear() { current_ = 0; stack_ = {}; }

 private:
  int current_{};
  std::stack<std::unique_ptr<CalculatorCommand>> stack_;
};
```

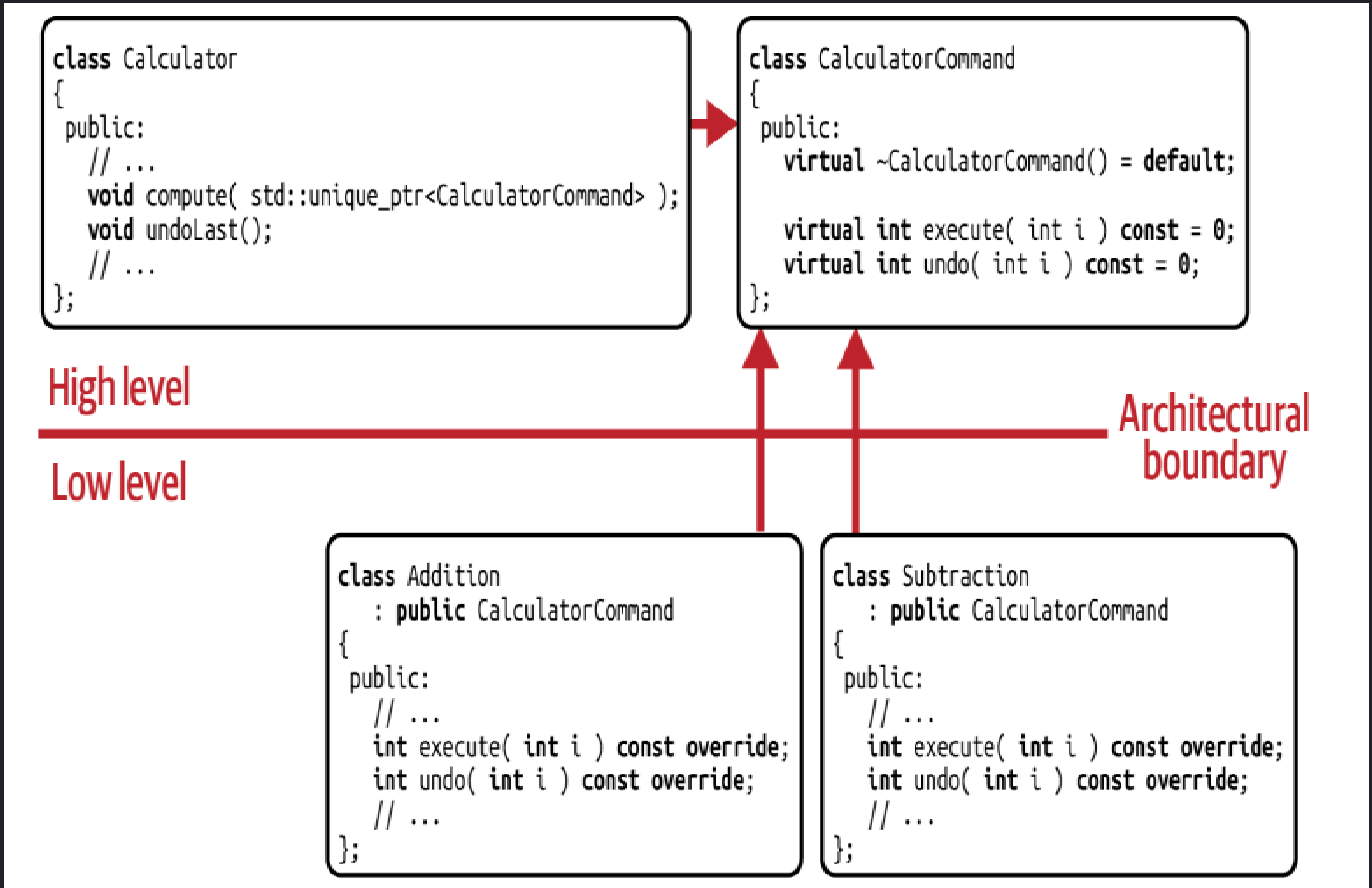


Figure 5-8. Dependency graph for the Command design pattern

std::for_each is an example of a command in the standard library.

```cpp
namespace std {

template<typename InputIt, typename UnaryFunction>
constexpr UnaryFunction
 for_each(InputIt first, InputIt last, UnaryFunction f);
} // namespace std
```

With the third argument, you can specify **what to do** with each element.

Command vs. Strategy

- Structurally they are same but they differ in intent.
- std::sort use strategy pattern because you can specify **how to do** the sorting.
- std::for_each use command pattern because you can specify **what to do** with each element.

G22: Prefer Value Semantics over Reference Semantics

- GOF style design patterns are firmly rooted in OOP so reference semantics are used.
- But virtual functions adds runtime overhead and use dynamic memory allocation.

Reference semantics examples:

- **std::span:** keeps a non-owning reference to a contiguous sequence of objects.
- **std::remove:**

```
std::vector<int> vec{1, -3, 27, 42, 4, -8, 22, 42, 37, 4, 18, 9};
auto const pos = std::max_element(begin(vec), end(vec));
vec.erase(std::remove(begin(vec), end(vec), *pos), end(vec));
// result: {1 -3 27 4 -8 22 42 37 18 9}
```

- References, especially pointers, makes our life much harder.
- It's harder to reason about the code so it's easier to introduce bugs.
- **Value semantics** is easier to reason about and less error-prone.

- STL containers like **std::vector** are value types.
- Their elements are copied when the container is copied instead of just copying the address.
- Copying might be expensive but **move semantics** can be used to avoid this.
- std::optional and std::variant are also good examples of value semantics.

std::function

- It represents an abstraction for a callable.
- It can hold a lambda, a function pointer, a function object.
- It can be used for implementing value semantics implementation of the design patterns.

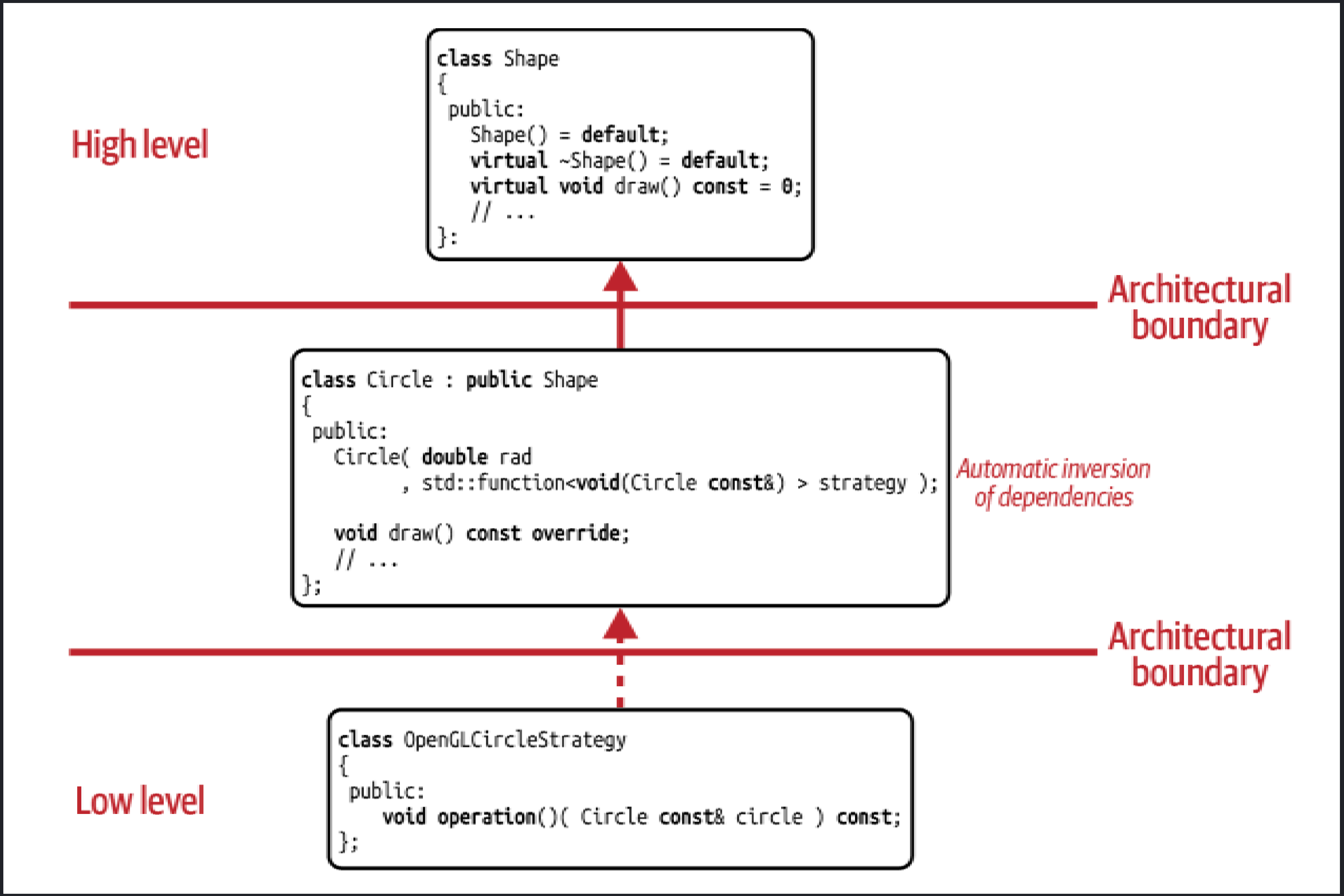
```cpp
void foo(int i)
{
   std::cout << "foo: " << i << '\n';
}

int main()
{
   // Create a default std::function instance. Calling it results
   // in a std::bad_function_call exception
 std::function<void(int)> f{};

 f = [](int i){ // Assigning a callable to 'f'
      std::cout << "lambda: " << i << '\n';
   };

 f(1); // Calling 'f' with the integer '1'

 auto g = f; // Copying 'f' into 'g'

 f = foo; // Assigning a different callable to 'f'

 f(2); // Calling 'f' with the integer '2'
 g(3); // Calling 'g' with the integer '3'

   return EXIT_SUCCESS;
}
```

Strategy Pattern with std::function

```
class Circle : public Shape
{
 public:
 using DrawStrategy = std::function<void(Circle const&, /*...*/)>;

 explicit Circle( double radius, DrawStrategy drawer )
      : radius_( radius )
 , drawer_( std::move(drawer) )
   {
      //...
   }

   void draw( /*some arguments*/ ) const override
   {
      drawer_( *this, /*some arguments*/ );
   }

 private:
   double radius_;
   DrawStrategy drawer_;
};
```

```cpp
class OpenGLCircleStrategy
{
 public:
   explicit OpenGLCircleStrategy( /* Drawing related arguments */ );

 void operator()( Circle const& circle, /*...*/ ) const;

 private:
   /* Drawing related data members, e.g. colors, textures, ... */
};
```

```cpp
std::vector<std::unique_ptr<Shape>> shapes{};

shapes.emplace_back(
      std::make_unique<Circle>(2.3, OpenGLCircleStrategy(/*...red...*/)));
shapes.emplace_back(
      std::make_unique<Square>(1.2, OpenGLSquareStrategy(/*...green...*/)));
shapes.emplace_back(
      std::make_unique<Circle>( 4.1, OpenGLCircleStrategy(/*...blue...*/) ) );

for( auto const& shape : shapes )
{
   shape->draw();
}
```


High level
class Shape
{
 public:
   Shape() = default;
   virtual ~Shape() = default;
   virtual void draw() const = 0;
   // ...
};
Architectural boundary
class Circle : public Shape
{
 public:
   Circle(double rad
         , std::function<void(Circle const&)> strategy);
   void draw() const override;
   // ...
};
Automatic inversion of dependencies
Architectural boundary
Low level
class OpenGLCircleStrategy
{
 public:
   void operation()(Circle const& circle) const;
};

std::function has performance overhead if you use the standard implementation.

Table 5-1. Performance results for different Strategy implementations

Strategy implementations	GCC 11.1	Clang 11.1
Object-oriented solution	1.5205 s	1.1480 s
`std::function`	2.1782 s	1.4884 s
Manual implementation of `std::function`	1.6354 s	1.4465 s
Classic Strategy	1.6372 s	1.4046 s

Final Comments?

See you in part 4!