C++ Design Patterns Documentation

Release 0.0.1

Hans-J. Schmid

Apr 14, 2019

Contents:

CHAPTER 1

Design Principles

1.1 SOLID

S = Single Responsibility Principle
O = Open-Closed Principle
L = Liskov Substitution Principle
I = Interface Segregation Principle
D = Dependency Inversion/Injection

1.1.1 Single Responsibility Principle

A class should only have a single responsibility.

Listing 1: SRP.cpp

```
Journal journal("My Journal");
journal.add("First Entry");
journal.add("Second Entry");
journal.add("Third Entry");

// Use a separate class/entity for saving.
// Saving journals is not a base responsibility of a journal.
PersistenceManager().save(journal, "journal.txt");
```

Full source code SRP.cpp

1.1.2 Open-Closed Principle

Entities should be open for extension but closed for modification.

Listing 2: OCP.cpp

```
Product apple{"Apple", Color::Green, Size::Small};
Product tree{"Tree", Color::Green, Size::Large};
Product house{"House", Color::Blue, Size::Large};

ProductList all{apple, tree, house};

BetterFilter bf;
ColorSpecification green(Color::Green);

auto green_things = bf.filter(all, green);
for (auto& product : green_things)
  std::cout << product.name << " is green" << std::endl;

SizeSpecification big(Size::Large);
// green_and_big is a product specification
AndSpecification<Product> green_and_big{big, green};

auto green_big_things = bf.filter(all, green_and_big);
for (auto& product : green_big_things)
  std::cout << product.name << " is green and big" << std::endl;
```

Full source code OCP.cpp

1.1.3 Liskov Substitution Principle

Objects should be replaceable with instances of their subtypes without altering program correctness.

Listing 3: LSP.cpp

```
Rectangle r{5, 5};
process(r);

// Square (subtype of Rectangle) violates the Liskov Substitution Principle
Square s{5};
process(s);
```

Full source code LSP.cpp

1.1.4 Interface Segregation Principle

Many client-specific interfaces better than one general-purpose interface.

Listing 4: ISP.cpp

```
Printer printer;
Scanner scanner;
Machine machine(printer, scanner);
std::vector<Document> documents{Document(std::string("Hello")),
                                Document(std::string("Hello"))};
machine.print(documents);
machine.scan(documents);
```

Full source code ISP.cpp

1.1.5 Dependency Inversion/Injection

Dependencies should be abstract rather than concrete.

Dependency Inversion Principle

1. **High-level modules should not depend on low-level modules. Both should depend on abstractions.** Example: reporting component should depend on a ConsoleLogger, but can depend on an ILogger.
2. **Abstractions should not depend upon details. Details should depend upon abstractions.** In other words, dependencies on interfaces and supertypes is better than dependencies on concrete types.

Inversion of Control (IoC) – the actual process of creating abstractions and getting them to replace dependencies.

Dependency Injection – use of software frameworks to ensure that a component's dependencies are satisfied.

Listing 5: BoostDI.cpp

```
// without DI
std::cout << "without DI\n";
auto e1 = std::make_shared<Engine>();
auto logger1 = std::make_shared<ConsoleLogger>();
auto c1 = std::make_shared<Car>(e1, logger1);
std::cout << *c1 << std::endl;

// with DI
std::cout << "with DI\n";
using namespace boost;
// whenever an ILogger is needed a ConsoleLogger instance will be created
auto injector = di::make_injector(di::bind<ILogger>().to<ConsoleLogger>());
// engine created with default constructor
auto c = injector.create<std::shared_ptr<Car>>();

std::cout << *c << std::endl;
```

Full source code BoostDI.cpp

CHAPTER 2

Creational Design Patterns

2.1 Builder

When **piecewise** object construction is complicated, provide an API for doing it succinctly.

Builder Example. Building a string:

- Building a string out of substrings
  - e.g. web server page rendering
- Concatenate a single std::string
  - my_text += boost::lexical_cast<string>(my_int);
- printf("%s = %n", "meaning of life", 42);
- Boost.Format
  - format("%1%,x=%2% : %3%-th try") % "toto" % 40.23 % 50
- Represent as OO structure and implement operator<< on an object graph

2.1.1 Life without Builder

Listing 1: LifeWithoutBuilder.cpp

```
// 1. Using out-of-the-box string concatenation
auto text = "hello";
string output;
output += "<p>";
output += text;
output += "</p>";
cout << "1. cout:" << endl;
cout << output << endl;
```

(continues on next page)

(continued from previous page)

```
  // 2. Using printf:
  cout << "\n2. Printf:" << endl;
  printf("<p>%s</p>\n", text);

  // 3. Using string streams
  string words[] = {"hello", "world"};
  ostringstream oss;
  oss << "<ul>\n";
  for (auto w : words) oss << "  <li>" << w << "</li>\n";
  oss << "</ul>\n";
  cout << "\n3. Output Stream:" << endl;
  printf("%s", oss.str().c_str());
```

Full source code LifeWithoutBuilder.cpp

2.1.2 Builder

A builder is a separate component with an API for building up a complex object. You can give the builder a public constructor or return it via a static function

Listing 2: Builder.cpp

```
  HtmlBuilder builder{"ul"};
  builder.add_child("li", "hello");
  builder.add_child("li", "world");
  cout << builder.str() << endl;
```

Full source code Builder.cpp

2.1.3 Fluent Builder

To make a fluent builder, return *this* or **this*.

Listing 3: FluentBuilder.cpp

```
  // add_child returns an HtmlBuilder.
  // Due to conversion operator HtmlElement() in HtmlBuilder this will be converted␣
↪to an
  // HtmlElement!
  HtmlElement htmlElement1 =
      HtmlElement::build("ul").add_child("li", "hello").add_child("li", "world");
  cout << htmlElement1.str() << endl;
```

Full source code FluentBuilder.cpp

2.1.4 Groovy Style Builder

Initializer lists let you implement Groovy-style builders with ease.

Listing 4: GroovyStyle.cpp

```
std::cout << P{IMG{"http://pokemon.com/pikachu.png"},
                P{IMG{"http://pokemon.com/pikachu.jpg"}}}
          << std::endl;
```

Full source code GroovyStyle.cpp

2.1.5 Builder Facets

Different facets of an object can be built with different builders working in tandem.

Listing 5: Facets.cpp

```
Person p = Person::create()
                .lives()
                .at("123 London Road")
                .with_postcode("SW1 1GB")
                .in("London")
                .works()
                .at("PragmaSoft")
                .as_a("Consultant")
                .earning(10e6);

cout << p << endl;
```

Full source code Facets.cpp

2.2 Factories

Definition Factory: A separate component responsible solely for the **wholesale** (**not piecewise**) creation of objects.

- Object creation logic becomes too convoluted
- Constructor is not descriptive
  - Name mandated by name of containing type
  - Cannot overload with same sets of arguments with different names
  - Can turn into 'optional parameter hell'
- Object creation (**non-piecewise** unlike Builder) can be outsourced to
  - A separate function (**Factory Method**)
  - That may exist in a separate class (**Factory**)
  - You can even have a hierarchy of classes with **Abstract Factory**
- A factory method is a function that creates objects
- A factory can take care of object creation
- A factory can reside inside the object or be external
- Hierarchies of factories can be used to create related objects

2.2.1 Point Example

Listing 6: PointExample.cpp

```
Point p{1, 2};
std::cout << p << std::endl;
```

Full source code PointExample.cpp

2.2.2 Factory Method

A **function** that helps create objects. Like a constructor but more descriptive.

Listing 7: FactoryMethod.cpp

```
auto p = Point::NewPolar(5, M_PI_4);
std::cout << p << std::endl;
```

Full source code FactoryMethod.cpp

2.2.3 Factory

A **class** that helps create objects.

Listing 8: Factory.cpp

```
auto cartesianPoint = PointFactory::NewCartesian(1.1, 2.2);
auto polarPoint = PointFactory::NewPolar(2.0, M_PI_4);

std::cout << cartesianPoint << std::endl;
std::cout << polarPoint << std::endl;
```

Full source code Factory.cpp

2.2.4 Inner Factory

An **inner class** that helps create objects.

Listing 9: InnerFactory.cpp

```
auto cartesianPoint = Point::Factory.NewCartesian(2, 3);
std::cout << cartesianPoint << std::endl;
```

Full source code InnerFactory.cpp

2.2.5 Abstract Factory

A factory construct used to construct object in hierarchies.

Listing 10: Builder.cpp AbstractFactory.cpp

```
// Would work but we want to have a more OO like interface.
auto d = make_drink("tea");

DrinkFactory df;
df.make_drink("coffee");
```

Full source code AbstractFactory.cpp

2.2.6 Functional Factory

Use of mappings from strings to factory functions.

Listing 11: FunctionalFactory.cpp

```
DrinkWithVolumeFactory dvf;
dvf.make_drink("tea");
```

Full source code FunctionalFactory.cpp

See also definition of class DrinkWithVolumeFactory

2.3 Prototype

Definition of Prototype: A partially or fully initialized object that you copy/clone and make use of.

- Complicated objects (e.g. cars) aren't designed from scratch
  - They reiterate existing designs
- An existing (partially constructed design) is a *Prototype*
- We make a copy/clone of the prototype and customize it
  - Requires 'deep copy' support
  - Painful without metadata!
- We make the cloning convenient (e.g. via a *Factory*)
- To implement a prototype, partially construct an object and store it somewhere
- Clone the prototype and then customize the instance
- Ensure deep copying! (Be careful of shallow copies esp. with respect to pointers!)

2.3.1 Prototype Factory

Use a factory to make prototypes more comfortable to use.

Listing 12: Prototype.cpp

```
auto john = EmployeeFactory::NewMainOfficeEmployee("John", 100);
auto jane = EmployeeFactory::NewAuxOfficeEmployee("Jane", 123);

cout << *john << endl << *jane << endl;
```

Full source code Prototype.cpp

2.3.2 Prototype with Boost Serialization

Use Boost Serialization for deep copying of protoype data.

Listing 13: Serialization.cpp

```
Employee john;
john.name = "John Doe";
john.address = new Address{"123 East Dr", "London", 123};

// Boost Serialization does a deep copy out-of-the box!
auto clone = [](Employee c) {
  // Write employee to an archive
  ostringstream oss;
  boost::archive::text_oarchive oa(oss);
  oa << c;

  string s = oss.str();

  // Read it back in. Deep copy automatically done!
  Employee result;
  istringstream iss(s);
  boost::archive::text_iarchive ia(iss);
  ia >> result;
  return result;
};

Employee jane = clone(john);
jane.name = "Jane";
jane.address->street = "123B West Dr";

cout << john << endl << jane << endl;
```

Full source code Serialization.cpp

2.4 Singleton

Definition of Singleton: A component which is instantiated only once.

"When discussing which patterns to drop, we found that we still love them all. (Not really—I'm in favor of dropping Singleton. Its use is almost always a design smell.)" - Erich Gamma

- For some components it only makes sense to have one in the system
  - Database repository
  - Object factory

- E.g. when the constructor call is expensive
  - We only do it once
  - We provide everyone with the same instance
- Want to prevent anyone creating additional copies
- Need to take care of lazy instantiation and multithreading
- A safe singleton is easy: just make a static variable and return a reference to it
- Singetons are difficult to test
- Consider defining singleton lifetime with a DI container

2.4.1 Singleton Database

Listing 14: SingletonDatabase.cpp

```
// Cannot do something like this because copy constructor does not exist!
// auto singletonDB = SingletonDatabase::get();

std::cout << SingletonDatabase::get().get_population("Tokyo") << "\n";

// We will always get the same instance of the singleton database!
assert(&SingletonDatabase::get() == &SingletonDatabase::get());
```

Full source code SingletonDatabase.cpp

CHAPTER 3

Structural Design Patterns

3.1 Adapter

Definition of Adapter: A construct which adapts an existing interface X to conform to a required interface Y.

- Electrical devices have different power (interface) requirements.
  - Voltage (5V, 220V)
  - Socket/plug type (Europe, UK, USA)
- We cannot modify our gadgets to support every possible interface.
- Thus, we use a device - an adapter - that gives us the interface we require.
- Implementing an Adapter is easy:
  - Determine the API you have and the API you need.
  - Create a component which aggregates (has a reference to, ...) the adaptee.
  - Intermediate representations can pile up. Use caching and other optimizations.

3.1.1 Stack

Listing 1: StructuralStack.cpp

```
// Stack from the STL is just an adapter of a vector.
// Stack has an underlying container (by default deque).
// Stack uses methods of underlying container
// e.g. push_back, pop_back, back, ...
// to implement stack specific methods like push, pop, top, ...
std::stack<int> stack;
stack.push(123);
int x = stack.top();
stack.pop();
```

Full source code StructuralStack.cpp

3.1.2 String

Listing 2: StructuralString.cpp

```
String string{"Hello    World"};

auto parts = string.to_lower().split();
for (const auto& part : parts) cout << "<" << part << ">" << endl;
```

Full source code StructuralString.cpp

3.2 Bridge

Definition of Bridge: A mechanism that decouples an interface (interface hierarchy) from an implementation (implementation hierarchy).

- Decouple abstraction from implementation.
- Both can exist as hierarchies.
- A stronger form of encapsulation.

3.2.1 PIMPL

PIMPL = **P**ointer to an **IMPL**ementation

See also Pimpl For Compile-Time Encapsulation

Listing 3: main.cpp

```
Person john(std::string("John"));
// greet method is implemented in class PersonImpl not in class Person
john.greet();
```

Full source code main.cpp

3.2.2 Shape Sizing

Listing 4: ShapeSizing.cpp

```
RasterRenderer rasterRenderer;
// Circle uses a raster renderer which is a reference to a Renderer.
Circle raster_circle{rasterRenderer, 10, 10, 5};
raster_circle.draw();
raster_circle.resize(2);
raster_circle.draw();
```

Full source code ShapeSizing.cpp

3.3 Composite

Definition of Composite: A mechanism for treating individual (scalar) and compositions of objects in a uniform manner.

- Objects use other objects' fields and members through inheritance and composition.
- Composition lets us make compound objects.
  - A mathematical expression composed of simpler expressions
  - A grouping of shapes that make up several shapes
- Composite design pattern is used to treat both single and composite objects **uniformly** (i.e. with identical APIs)
- Objects can use either objects either via inheritance or composition.
- Some composed and singular objects need similar/identical behaviors.
- Composite design pattern lets us treat both types of objects uniformly.
- C++ has no special support for the idea of 'enumeration' of objects.
- **Trick:** A single object can *masquerade* to become suitable for begin/end iteration.
  - See Neural Network code which unfortunately compiles only under Microsoft Windows.
  - Link to Neurons code.

```
Neuron *begin() { return this; }
Neuron *end() { return this + 1; }
```

3.3.1 Object Composition and Iteration

Listing 5: composite.cpp

```
AdditionExpression sum{
    make_shared<Literal>(2),
    make_shared<AdditionExpression>(make_shared<Literal>(3), make_shared<Literal>
↪(4))};

cout << "2+(3+4) = " << sum.eval() << endl;

vector<double> v;
sum.collect(v);
for (auto x : v) cout << x << " ";
cout << endl;

vector<double> values{1, 2, 3, 4};
double s = 0;
for (auto x : values) s += x;
cout << "average is " << (s / values.size()) << endl;

accumulator_set<double, stats<tag::mean>> acc;
for (auto x : values) acc(x);
cout << "average is " << mean(acc) << endl;
```

Full source code composite.cpp

3.3.2 Geometric Shapes

Listing 6: graphics.cpp

```
Group root("root");
Circle c1;
root.objects.push_back(&c1);

Group subgroup("sub");
Circle c2;
Rectangle r1;
subgroup.objects.push_back(&c2);
subgroup.objects.push_back(&r1);

root.objects.push_back(&subgroup);

root.draw();
```

Full source code graphics.cpp

3.4 Decorator

Definition of Decorator: Allows for adding behavior to individual objects without affecting the behavior of other objects of the same class.

- Want to augment existing functionality.
- Do not want to rewrite or alter existing code (Open-Closed Principle).
- Want to keep new functionality separate (Single Responsibility Principle)
- Need to be able to interact with existing structures
- Functional decorators let you wrap functions with before/after code (e.g. for logging).
- An aggregate decorator does not give you the underlying object's features, but can be composed at runtime.
- A decorator based on mixin inheritance is more flexible, exposes underlying object's features, but is only constructible at compile time because of implementation as C++ template functions.

3.4.1 Function Decorator

Listing 7: functionDecorator.cpp

```
// logger is a decorator
Logger logger{[](){cout << "Hello" << endl; }, "HelloFunc"};
logger();

// Template argument not deduced from lambda. Need a helper function.
// Logger2<T?> logger1{[](){cout << "Hello" << endl; }, "HelloFunc"};
make_logger2([]() { cout << "Hello" << endl; }, "HelloFunction");

// add_logger behaves like the original add function.
auto add_logger = make_logger3(add, "Add");
auto result = add_logger(2, 3);
```

Full source code functionDecorator.cpp

3.4.2 Wrapping Decorator

Listing 8: wrappingDecorator.cpp

```
Circle circle{5};
cout << circle.str() << endl;

ColoredShape red_circle{circle, "red"};
cout << red_circle.str() << endl;

TransparentShape red_half_visible_circle{red_circle, 128};
cout << red_half_visible_circle.str() << endl;

// Oops! Unfortunately this does not work!
// red_half_visible_circle.resize();
```

Full source code wrappingDecorator.cpp

3.4.3 Mixin Decorator

Listing 9: mixinDecorator.cpp

```
// Won't work without a default constructors. Here for Circle.
ColoredShape<Circle> green_circle{"green"};
green_circle.radius = 5;
cout << green_circle.str() << endl;

TransparentShape<ColoredShape<Square>> blue_invisible_square{0.5};
blue_invisible_square.color = "blue";
blue_invisible_square.side = 10;
cout << blue_invisible_square.str() << endl;
```

Full source code mixinDecorator.cpp

3.4.4 Improved Decorator

Listing 10: improvedDecorator.cpp

```
// Now we can provide transparency and radius in the constructor.
// Default constructors are not needed any more.
TransparentShape<Square> half_hidden_square{0.5, 15.f};
cout << half_hidden_square.str() << endl;
```

Full source code improvedDecorator.cpp

3.5 Façade

Definition of Façade: Provides a simple, easy to understand/use interface over a large and sophisticated body of code.

- Balancing complexity and presentation/usability.
- Typical home:
  - Many subsystems (electrical, sanitation).

  - Complex internal structure (e.g. floor layers).
  - End user not exposed to internals.
- Same with software!
  - Many systems working together provide flexibility, but...
  - API consumers want it to 'just work'.
- Make a library easier to understand, use and test.
- Reduce dependencies of user code on internal APIs that may change.
  - Allows more flexibility in developing/refactoring the library.
- Wrap a poorly designed collection of APIs with a single well-designed API.
- Build a Façade to provide a simplified API over a set of classes.
- May wish to (optionally) expose internals though the façade.
- May allow users to 'escalate' to use more complex APIs if they need to.

3.5.1 Bloom Terminal

Listing 11: Bloom.cpp

```
// Console is a façade for Windows, Viewports, Buffers,... used by the terminal app
auto window = Console::instance().multicolumn("Test", 2, 40, 40);

for (size_t i = 0; i < 40; i++) {
  window->buffers[1]->add_string("This is line " + boost::lexical_cast<string>(i));
}

window->Show();
```

Full source code Bloom.cpp

3.6 Flyweight

Definition of Flyweight: A space optimization technique that lets us use less memory by storing externally the data associated with similar objects.

- Avoiding redundancy when storing data, e.g. MMORPG:
  - Plenty of users with identical first/last names
  - No sense in storing same first/last name over and over again
  - Store a list of names and pointers to them
- Bold or italic text in the console:
  - Don't want each character to have an extra formatting character.
  - Operate on ranges (e.g. line, start/end).

3.6.1 First/Last Name

Listing 12: flyweight.cpp

```
User2 john_doe{"John", "Doe"};
User2 jane_doe{"Jane", "Doe"};

cout << "John " << john_doe << endl;
cout << "Jane " << jane_doe << endl;

// "Doe" is only saved once in the system.
assert(&jane_doe.last_name.get() == &john_doe.last_name.get());
```

Full source code flyweight.cpp

3.7 Null Object

Definition of Null Object: A no-op object that satisfies the dependency requirement of some other object.

- When component A uses component B, if typically assumes that B is actually present.
  - You inject B, not e.g. optional<B>.
  - You do not inject a pointer and then check for nullptr everywhere.
- There is no option of telling A not to use an instance of B.
  - Its use is hard-coded.
- Thus, we build a no-op, non-functioning inheritor of B and pass that into A.
- Structural or Behavioral design pattern?
- Inherit from the required object.
- Implement the functions with empty bodies.
  - Return default values.
  - If those values are used, you are in trouble.
- Supply an instance of the Null Object in lieu of an actual object.

3.7.1 Null Logger

Listing 13: nullobject.cpp

```
auto consoleLogger = make_shared<ConsoleLogger>();
BankAccount account1{consoleLogger, "First account", 100};

account1.deposit(2000);
account1.withdraw(2500);
account1.withdraw(1000);

auto nullLogger = std::make_shared<NullLogger>(10);
// Constructor of BankAccount needs a logger. But we don't want one.
// Solution: Provide a null logger which doesn't log anything.
BankAccount account2{nullLogger, "Second account", 100};
```

(continues on next page)

(continued from previous page)

```
account2.deposit(2000);
account2.withdraw(2500);
account2.withdraw(1000);
```

Full source code nullobject.cpp

3.8 Proxy

Definition of Proxy: A class that is functioning as an interface to a particular resource. That resource may be remote, expensive to construct, or may require logging or some other added functionality.

- You are calling foo.bar().
- This assumes that foo resides in the same process as bar.
- What if, later on, you want to put all Foo related operations into a separate process?
  - How can you avoid changing all your code?
- Proxy to the rescue!
  - Same interface, entirely different behavior.
- This is an example for a communication proxy.
  - There are many others: logging, virtual, guarding,...
- How is Proxy different from Decorator?
  - Proxy provides an identical interface; decorator provides an enhanced interface.
  - Decorator typically aggregates (or has reference to) what it is decorating; proxy doesn't have to.
  - Proxy might not even be working with a materialized object.
- A proxy has the same interface as the underlying object.
- To create a proxy, simply replicate the existing interface of an object.
- Add relevant functionality to the redefined member functions.
  - As well as constructor, destructor, etc.
- Different proxies (communication, logging, caching, etc.) have completely different behaviors.

3.8.1 Smart Pointer Proxy

Smart pointers from the standard library don't need an explicit delete. Smart pointers are proxies for underlying raw pointers.

Listing 14: smartPointerProxy.cpp

```
BankAccount* a = new CurrentAccount(1000);
a->deposit(1500);
delete a;

auto b = make_shared<CurrentAccount>(1000);
```

(continues on next page)

(continued from previous page)

```
BankAccount* actual = b.get();  // pointer's own operations on a.
b->deposit(150);                // underlying object's operations are on ->
                                // note this expression is identical to what's above
cout << *b << endl;
// no delete
```

Full source code smartPointerProxy.cpp

3.8.2 Virtual Proxy

Listing 15: virtualProxy.cpp

```
LazyBitmap img{"pokemon.png"};
// Image will only be loaded on first attempt to draw the image.
draw_image(img);
draw_image(img);
draw_image(img);
```

Full source code virtualProxy.cpp

3.8.3 Communication Proxy

Listing 16: communicationProxy.cpp

```
cout << "Local Pong:\n";
Pong localPong;
for (int i = 0; i < 5; ++i) {
  tryit(localPong);
}

// For a remote Pong you just have to implement the same interface Pingable.
cout << "\nRemote Pong:\n";
RemotePong remotePong;
for (int i = 0; i < 5; ++i) {
  tryit(remotePong);
}
```

Full source code communicationProxy.cpp

CHAPTER 4

Behavioral Design Patterns

4.1 Chain of Responsibility

Definition of Chain of Responsibility: A chain of components who all get a chance to process a command or query, optionally having a default processing implementation and an ability to terminate the processing chain.

- You click a graphical element on a form
  - Button handles it, might stop further processing
  - Underlying group box
  - Underlying window
- CCG computer game
  - Creature has attack and defense values.
  - Those can be boosted by other cards.
- Command Query Separation (CQS)
  - Command = asking for an action or change (e.g. please set your attack value to 2).
  - Query = asking for information (e.g. please give me your attack value).
  - CQS = having separate means fo sending commands and queries.
    - Antithetical to set fields directly.
- Chain of Responsibility can be implemented as a pointer chain or a centralized construct (event bus).
- Enlist objects in the chain, possibly controlling their order.
- Remove object from chain when no longer applicable (e.g. in its own destructor).

4.1.1 Pointer Chain

Listing 1: cor_pointer.cpp

```
Creature goblin{"Goblin", 1, 1};
CreatureModifier root{goblin};
DoubleAttackModifier r1{goblin};
DoubleAttackModifier r1_2{goblin};
IncreaseDefenseModifier r2{goblin};
// Uncomment the following lines to stop chain of responsibility instantly
// NoBonusModifier no{goblin};
// root.add(&no);

root.add(&r1);
root.add(&r1_2);
root.add(&r2);

// Start chain of responsibility.
root.handle();

cout << goblin << endl;
```

Full source code cor_pointer.cpp

4.1.2 Broker Chain

Listing 2: cor_broker.cpp

```
Game game;
Creature goblin{game, 2 /* attack */, 2 /* defense */, "Strong Goblin"};

cout << goblin << endl;

{
  // Goblin only temporarily changed.
  DoubleAttackModifier dam{game, goblin};
  cout << goblin << endl;
}

cout << goblin << endl;
```

Full source code cor_broker.cpp

4.2 Command

Definition of Command: An object which represents an instruction to perform a particular action. Contains all information necessary for the action to be taken.

- Ordinary C++ statements are perishable
  - Cannot undo a field assignment
  - Cannot directly serialize a sequence of actions
- Want an object that represents an operation
  - X should change its Y to Z

  - X should do W
- Usage: GUI commands, multi-level undo/redo, macro recording and more!
- Encapsulate all details of an operation in a separate object.
- Define instructions for applying the command (either in the command itself, or elsewhere).
- Optionally define instructions for undoing the command.
- Can create composite commands (a.k.a. macros).

4.2.1 Composite Command with Undo/Redo

Listing 3: command.cpp

```
BankAccount ba;
// Composite command.
CommandList commands{Command{ba, Command::deposit, 100},
                     Command{ba, Command::withdraw, 200}};
printBalance(ba);

commands.call();
printBalance(ba);

commands.undo();
printBalance(ba);
```

Full source code command.cpp

4.3 Interpreter

Definition of Interpretor: A component that processes structured text data. Does so by turning it into separate lexical tokens (lexing) and then interpreting sequences of said tokens (parsing).

- Textual input needs to be processed.
  - e.g. turned into OOP structures.
- Some examples:
  - Proramming language compilers, interpreters and IDEs.
  - HTML, XML and similar.
  - Numeric expressions (2+3/4).
  - Regular expressions.
- Turning strings into OOP based structures is a complicated process.
- Barring simple cases, an interpreter does two stages:
  - Lexing turns text in to a set of tokens, e.g.
    - 2*(3+4) -> Lit[2] Star LParen Lit[3] Plus Lit[4] Rparen
  - Parsing turns tokens into meaningful constructs
    - MultiplicationExpression{Integer{2}, AdditionExpression{Integer{3}, Integer{4}}}
- Parsed data can be traversed

4.3.1 Handwritten Interpreter

Listing 4: interpreter.cpp

```
// Hand-written interpreter (lexer and parser).
// For more complicated scenarios you could you e.g. Boost Sphinx.
string input{"(13-4)-(12+1)"};
auto tokens = lex(input);

cout << "Tokens of " << input << "\n";
for (auto& t : tokens) cout << t << "\n";
cout << endl;

auto parsed = parse(tokens);
cout << input << " = " << parsed->eval() << endl;
```

Full source code interpreter.cpp

4.4 Iterator

Definition of Iterator: An object that facilitates the traversal of a data structure.

- Iteration (traversal) is a core functionality of various data structures.
- An *iterator* is a class that facilitates the traversal
  - Keeps pointer to an element.
  - Knows how to move to a different element.
- Iterator types
  - Forward (e.g. on a list)
  - Bidirectional (e.g. on a doubly linked list)
  - Random access (e.g. on a vector)
- Iterator Requirements
  - Container Member Functions:
    - **begin()**
      - Points to the first element in the container. If empty is equal to **end()**.
    - **end()**
      - Points to the element immediately after the last element.
    - Facilitates use of standard algorithms.
    - Allow the use of range-based for loops:
      - *for (auto& x : my_container)*
    - Different names for different iterators.
  - Iterator Operators:
    - **operator !=**
      - Must return false if two iterators point to the same element.

        * **operator *** (dereferencing)
            - Must return a reference to (or a copy of) the data the iterator points to.
        * **operator ++**
            - Gets the iterator to point to the next element.
    - Additional operators as required (e.g. operator –, arithmetic, etc.)
- An iterator specifies how you can traverse an object.
- Typically needs to support comparison (**!=**), advancing (**++**) and dereferencing (*****).
    - May support other things, e.g. arithmetic, operator –, etc.
- Can have many different iterators (reverse, const, etc.)
    - Default on returned in begin()/end()
- Iterators can not be recursive.

4.4.1 STL Iterators

Listing 5: iteratorSTL.cpp

```
vector<string> names{"john", "jane", "jill", "jack"};

vector<string>::iterator it = names.begin();  // or begin(names)
cout << "first name is " << *it << "\n";

++it;  // advance the iterator
it->append(string(" goodall"));
cout << "second name is " << *it << "\n";

while (++it != names.end()) {
  cout << "another name: " << *it << "\n";
}

// traversing the entire vector backwards
// note global rbegin/rend, note ++ not --
// expand auto here
for (auto ri = rbegin(names); ri != rend(names); ++ri) {
  cout << *ri;
  if (ri + 1 != rend(names))  // iterator arithmetic
    cout << ", ";
}
cout << endl;

// constant iterators
vector<string>::const_reverse_iterator jack = crbegin(names);
// won't work
// *jack += "test";
```

Full source code iteratorSTL.cpp

4.4.2 Binary Tree Iterator

Listing 6: binaryTreeIterator.cpp

```
// in order traversal
BinaryTree<string> family{
  new Node<string>{"me",
      new Node<string>{"mother", new Node<string>{"mother's mother"},
        new Node<string>{"mother's father"}},
      new Node<string>{"father"}}};

// pre order traversal
for (auto it = family.begin(); it != family.end(); ++it) {
  cout << (*it).value << endl;
}
```

Full source code binaryTreeIterator.cpp

4.4.3 Boost Iterator Facade

Listing 7: facade.cpp

```
// Using 'Boost Iterator Facade' our Node looks like a traversable list.
Node alpha{"alpha"};
Node beta{"beta", &alpha};
Node gamma{"gamma", &beta};

for_each(ListIterator{&alpha}, ListIterator{},
         [](const Node& n) { cout << n.value << endl; });
```

Full source code facade.cpp

4.5 Mediator

Definition of Mediator: A component that facilitates communication between other components without them being aware of each other or having direct (referential) access to each other.

- Components may go in and out of a system at any time.
  - Chat room participants.
  - Players in an MMORPG.
- It makes no sense for them to have direct references to each other.
  - Those references may go dead.
- Solution: Have them all refer to the some central component that facilitates communication.
- Create the mediator and have each object in the system refer to it.
  - e.g. a reference field.
- Mediator engages in bidirectional communication with its connected components.
- Mediator has functions the components can call.
- Components have functions the mediator can call.

- Signals/slots (Boost.Signals2) and event processsing (RxCpp) libraries make communication easier to implement.

4.5.1 Chat Room

Listing 8: chat.cpp

```
ChatRoom room;

auto john = room.join(Person{"john"});
auto jane = room.join(Person{"jane"});

john->say("hi room");
jane->say("oh, hey john");

auto simon = room.join(Person{"simon"});
simon->say("hi everyone!");

jane->pm("simon", "glad you could join us, simon");
```

Full source code chat.cpp

4.5.2 Event Broker

Listing 9: mediator.cpp

```
// Using Boost Signal2.
Game game;
Player player{"Dmitri", game};
Coach coach{game};

player.score();
player.score();
player.score();
```

Full source code mediator.cpp

4.6 Memento

Definition of Memento: A token/handle representing the system state. Lets us roll back to the state when the token was generated. May or may not directly expose state information.

- An object or system goes through changes.
  - e.g. a bank account gets deposits and withdrawels.
- There are different ways of navigating those changes.
- One way is to record every change (Command design pattern) and teach a command to 'undo' itself.
- Another is simply to save snapshots of the system.
- Mementos are used to roll back changes arbitrarily.
- A memento is simply a token/handle class with (typically) no functions of its own.

- A memento is not required to expose directly the state(s) to which it reverts the system.
- Can be used to implement undo/redo.

4.6.1 Memento with Undo/Redo

Listing 10: memento.cpp

```
// Simple bank account with restore.
BankAccount ba1{100};
auto memento1 = ba1.deposit(50); // 150
auto memento2 = ba1.deposit(25); // 175
cout << ba1 << "\n";

// restore to memento1
ba1.restore(memento1);
cout << ba1 << "\n";

// restore to memento2
ba1.restore(memento2);
cout << ba1 << "\n--------------------\n";

// More elaborate bank account with undo.
BankAccount2 ba2{100};
ba2.deposit(50);  // 150
auto memento3 = ba2.deposit(25);  // 175
cout << ba2 << "\n";

ba2.undo();
cout << "Undo:\t " << ba2 << "\n";
ba2.undo();
cout << "Undo:\t " << ba2 << "\n";
ba2.redo();
cout << "Redo:\t " << ba2 << "\n";
// Restore still possible.
ba2.restore(memento3);
cout << "Restore: " << ba2 << "\n";
```

Full source code memento.cpp

4.7 Observer

Definition of Observer: An observer is an object that wishes to be informed about events happening in the system, typically by providing a callback function to call when events occur. The entity generating the events is sometimes called *observable*.

- We need to be informed when certain things happen
  - Object's property changes.
  - Object does something.
  - Some external event occurs.
- We want to listen to events and be notified when they occur.
- No built-in event functionality in C++.

- Function pointers, std::function, OOP constructs, special libraries
- Implementation of Observer is an intrusive approach: an observable must provide subscribe and unsubscribe functions and must have explicit notification code.
- Special care must be taken to prevent issues in multithreaded scenarios.
- Reentrancy is very difficult to deal with.
- Libraries such as Boost.Signals2 provide a usable implementation of Observer.

4.7.1 Observer with Boost Signal2

Listing 11: observer.cpp

```
Person p{123};
p.PropertyChanged.connect([](Person&, const string& property_name) {
  cout << property_name << " has been changed "
       << "\n";
});
p.SetAge(20);
```

Full source code observer.cpp

4.7.2 Thread Safety and Observer

Listing 12: observer2.cpp

```
Person p{14};
ConsoleListener cl;
p.subscribe(&cl);
p.subscribe(&cl);  // ignored
p.set_age(15);
p.set_age(16);
p.unsubscribe(&cl);
p.set_age(17);
```

Full source code observer2.cpp

4.8 State

Definition of State: A pattern in which the object's behavior is determined by its state. An object transitions from one state to another (something needs to *trigger* the transition). A formalized construct which manages states and transitions is called a *state machine*.

- Consider an ordinary telephone.
- What you do with it depends on the state of the phone/line.
  - If it's ringing or you want to make a call, you can pick it up.
  - Phone must be off the hook to take/make a call.
  - If you are calling someone, and it's busy, you put the handset down.
- Changes in state can be explicit or in response to events (e.g. Observer).

- Given sufficient complexity, it pays to formally define possible states and events/triggers.
- Can define:
  - State entry/exit behaviors.
  - Action when a particular event causes a transition.
  - Guard conditions enabling/disabling a transition.
  - Default action when no transitions are found for an event.

4.8.1 Handwritten State Machine

Listing 13: basic.cpp

```
map<State, vector<pair<Trigger, State>>> rules;

rules[State::OffHook] = {{Trigger::CallDialed, State::Connecting}};

rules[State::Connecting] = {{Trigger::HungUp, State::OffHook},
                            {Trigger::CallConnected, State::Connected}};

rules[State::Connected] = {{Trigger::LeftMessage, State::OffHook},
                           {Trigger::HungUp, State::OffHook},
                           {Trigger::PlacedOnHold, State::OnHold}};

rules[State::OnHold] = {{Trigger::TakenOffHold, State::Connected},
                        {Trigger::HungUp, State::OffHook}};

// Initializing state
State currentState{State::OffHook};

while (true) {
  cout << "The phone is currently " << currentState << endl;
select_trigger:
  cout << "Select a trigger (quit with 666):\n";

  int i = 0;
  for (auto item : rules[currentState]) {
    cout << i++ << ". " << item.first << "\n";
  }

  int input;
  cin >> input;
  if(input == 666)
    break;
  if (input < 0 || (input + 1) > rules[currentState].size()) {
    cout << "Incorrect option. Please try again."
         << "\n";
    goto select_trigger;
  }

  currentState = rules[currentState][input].second;
}

cout << "We are done using the phone\n";
```

Full source code basic.cpp

4.8.2 Boost State Machine - MSM

Listing 14: msm.cpp

```
msm::back::state_machine<PhoneStateMachine> phone;

auto info = [&]() {
  auto i = phone.current_state()[0];
  cout << "The phone is currently " << state_names[i] << "\n";
};

info();
phone.process_event(CallDialed{});
info();
phone.process_event(CallConnected{});
info();
phone.process_event(PlacedOnHold{});
info();
phone.process_event(PhoneThrownIntoWall{});
info();

// try process_event here :)
phone.process_event(CallDialed{});

cout << "We are done using the phone\n";
```

Full source code msm.cpp

4.9 Strategy (Policy)

Definition of Strategy: Enables the exact behavior of a system to be selected at either *run-time (dynamic)* or *compile-time (static)*. Also known as a *policy*.

- Many algorithms can be decomposed into higher-level and lower-level parts.
- Making tea can be decomposed into:
  - The process of making a hot beverage (boil water, pour into cup); and
  - Tea-specific things (get a teabag).
- The high-level algorithm can then be reused for making coffee or hot chocolate.
  - Supported by beverage-specific strategies.
- Define an algorithm at a high level.
- Define the interface you expect each strategy to follow.
- Provide for either dynamic or static(C++ templates) composition of strategy in the overall algorithm.

4.9.1 Static Strategy

Listing 15: strategy_static.cpp

```
// markdown
TextProcessor<MarkdownListStrategy> tpm;
tpm.append_list({"foo", "bar", "baz"});
cout << tpm.str() << endl;

// html
TextProcessor<HtmlListStrategy> tph;
tph.append_list({"foo", "bar", "baz"});
cout << tph.str() << endl;
```

Full source code strategy_static.cpp

4.9.2 Dynamic Strategy

Listing 16: strategy_dynamic.cpp

```
// markdown
TextProcessor tp;
tp.set_output_format(OutputFormat::Markdown);
tp.append_list({"foo", "bar", "baz"});
cout << tp.str() << endl;

// html
tp.clear();
tp.set_output_format(OutputFormat::Html);
tp.append_list({"foo", "bar", "baz"});
cout << tp.str() << endl;
```

Full source code strategy_dynamic.cpp

4.10 Template

Definition of Template: Allows us to define the 'skeleton' of the algorithm, with concrete implementations defined in subclasses.

- Algorithms can be decomposed into common parts and specifics.
- Strategy pattern does this through composition
  - High-level algorithm uses an interface.
  - Concrete implementations implement the interface.
- Template Method does the same thing through inheritance.
- Define an algorithm at a high level.
- Define constituent parts as pure virtual functions.
- Inherit the algorithm class, providing necessary function implementations.
- Similar to GP (Generative Programming).

4.10.1 Chess

Listing 17: template_method.cpp

```
// Game is template for Chess.
Chess chess;
chess.run();
```

Full source code template_method.cpp

4.11 Visitor

Definition of Visitor: A pattern where a component (visitor) is allowed to traverse the entire inheritance hierarchy. Implemented by propagating a single *visit()* function through the entire hierarchy.

- Dispatch: Which function to call?
  - Single dispatch: depends on name of request and type of receiver. This is standard C++ behavior.
  - Double dispatch: depends on name of request and type of two receivers - type of visitor and type of element being visited.
- Need to define a new operation on an entire class hierarchy
  - E.g. make a document model printable to HTML/Markdown.
- Do not want to keep modifying every class in the hierarchy.
- Create external component to handle the rendering.
  - But avoid type checks.
- Propagate a pure virtual *accept(Visitor&)* function through the entire hierarchy.
- Create visitor (interface) with *visit(Foo&), visit(Bar&)* for each element in the hierarchy.
- Each *accept()* simply calls *v.visit(*this)*.

4.11.1 Static Visitor

Listing 18: staticVisitor.cpp

```
Paragraph p{"Here are some colors: "};
ListItem red{"Red"};
ListItem green{"Green"};
ListItem blue{"Blue"};
List colors{red, green, blue};

vector<Element*>  document{&p, &colors};
ostringstream oss;
for_each(document.begin(), document.end(), [&](const Element* e) {
    // oss acts like a visitor
    e->printHTML(oss);
});
cout << oss.str();
```

Full source code staticVisitor.cpp

4.11.2 Double Dispatch

Listing 19: dynamicVisitor.cpp

```
Paragraph p{"Here are some colors: "};
ListItem red{"Red"};
ListItem green{"Green"};
ListItem blue{"Blue"};
List colors{red, green, blue};

vector<Element*> document{&p, &colors};
HTMLVisitor htmlVisitor;
ostringstream oss;
for_each(document.begin(), document.end(), [&](const Element* e) {
  // we call 'accept()' which in turn calls 'visit()'' with the proper class:
  // Paragraph, ListItem or List.
  e->accept(htmlVisitor);
});
cout << "HTML:\n" << htmlVisitor.str();

MarkdownVisitor markdownVisitor;
oss.clear();
for_each(document.begin(), document.end(), [&](const Element* e) {
    // we call 'accept()' which in turn calls 'visit()'' with the proper class:
    // Paragraph, ListItem or List.
    e->accept(markdownVisitor);
  });
cout << "\nMarkdown:\n" << markdownVisitor.str();
```

Full source code dynamicVisitor.cpp

4.11.3 Multiple Dispatch

Listing 20: multiDispatchVisitor.cpp

```
ArmedSpaceship spaceship;
Asteroid asteroid;
Planet planet;

collide(planet, spaceship);
collide(planet, asteroid);
collide(spaceship, asteroid);
// collide(planet, planet);
planet.collide(planet);
```

Full source code multiDispatchVisitor.cpp

CHAPTER 5

Indices and tables

- genindex
- modindex
- search