A Software Engineers guide towards Object Oriented Design

Kevin Wong

1 Introduction

2 Design Principle

Design principles are guidelines to assist in your object-oriented design.

2.1 Encapsulate what varies

Identify the aspects of your application that vary and separate them from what stays the same.

- Look for code that changes with every new requirements. We can see which behavior needs to be pulled out and separated from the stuff that does not change.
- Alter or extend the code that varies without affecting code that doesn't vary.
- Basis of almost every design pattern. All patterns tend to provide a way to let some part of your system vary independently on the other parts.
- Pay attention to how each pattern makes use of this principle.

2.2 Favor composition over inheritance

Identify the aspects of your application that vary and separate them from what stays the same. Also know as has-a is better than is-a.

- Inheritance is a powerfull technique that avoid code duplication. However it is a technique that can bet easily overused. It is also a technique that can lead to designs that are far too rigged and not extensible.
- Is-a is a relationship of inheritance e.g. dog is a animal.
- Has-a is a relationship of composition e.g. dog has a owner. Compisition can be a powerfull alternative to inheritance. How can we switch from Is-a to Has-a:

  - Instead of a CoffeeWithButter is-a Coffee.
  - what about a Coffee Has-a condiment?

- We can add any number of condiments easily at runtime.
- Implementing new condiments by adding a new class.
- No code duplication.
- Avoid class explosion.
- Instead of inheriting behavior, we can compose our objects with new behaviors.
- Composition often gives us more flexibility, even allows behavior changes at runtime.
- Composition is a common techniques used in design patterns.

2.3 Loose Coupling

Components should be independent, relying on knowledge of other component as little as possible.

- When two classes interact, changes in one class should not force changes on the other one.
- Loose coupling reduces the dependency between components. Keeping designs loosely coupled helps us to build object-oriented systems that can handle change well.
- Simple technique to achieve loose coupling: abstracting away a concrete type into an interface. This is a technique your are going to see reoccur over and over again in good design.
- A good example of loose couple is the Observer pattern.

2.4 Program to Interfaces, Not Implementations

Where possible, components should use abstract classes or interfaces instead of a specific implementation.

- "program to an Interface" really means "program to a super/abstract type".
- The point is to be able to exploit polymorphism by programming to a super type so that your actual runtime object isn't locked into you code.
- The "new" operator only work on concrete types. Doesn't that violates this principle? Here we can make use of creational design patterns.

- Encourages us to use interfaces/abstract classes when possible, rather than concrete classes.
- This principle frees classes from knowledge of concrete types.
- Improve extensibility and maintainability.

2.5 Single Responsiblity Principle

Classes should have only one reason to change.

- Look at the change in your class: Are parts of it changing while other parts are not?
- Change only matters if it really happens. So apply Single Responisiblity only when the need is real, or you are just creating complexity.

2.6 Open/Close Principle

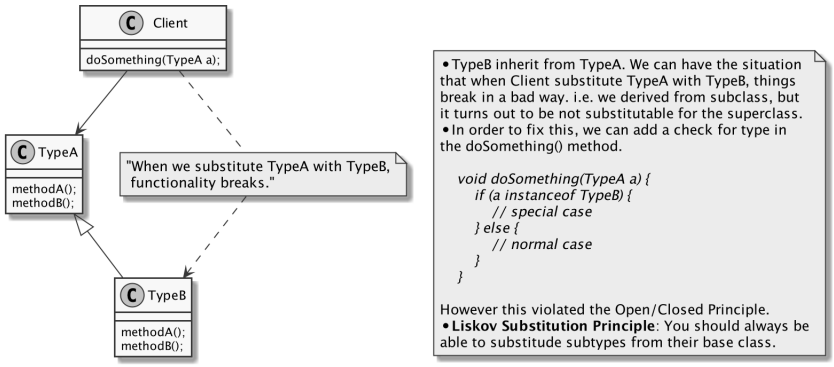
Object-Oriented designs should be open for extension, but closed for modification.

- An example of this is using composition to allow behavior extension of a class, while protecting the existing (proven) code of the class from modification.
- A design pattern that make use of this principle is the strategy pattern. In the duck example we are able to add new duck behaviors without changing the duck class.
- Improve maintainability and extensibility of a design.

2.7 Liskov's Substitution Principle

You should always be able to substitute subtypes for their base class.

- Using Inheritance the wrong way, can lead to undesired behavior when substituting the baseclass with the subsclass.

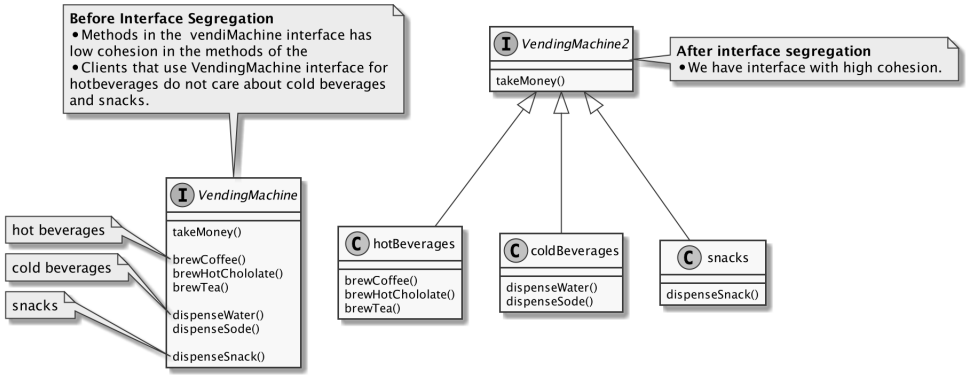


- Example is the Rectangle and Square class. When Square is derived from Rectangle, it violated the Liskov Substitution Principle, since changing the length of a Square also changes the width of the Square. This side-effect is not in the rectangle super class. i.e. a Square class can not substitude the Rectangle class.
- Don't assume that the IsA principle will always result in hierarchies that adhere to this principle. You will need to consider how substitutable a base class is for it's supertype as well.
- To adhere to this principle, we can use techniques like design by contract.

2.8 Interface Segregation Principle

Classes should not be forces to depend on methods that they don't use.

- Cohesion: How strong are the relationship between an interface's methods. i.e. We say a class is highly cohesive if their methods are related.
- Think able the client that uses the interface. Is the client going to use all the method in the interface? if yes then the interface has high cohesion.
- Highly cohesive interfaces leads to software that is easier to maintain and easier to extend.
- Segregate Interfaces needed to keep them focussed and cohesive. i.e. you free the clients from methods they are not intrested in.

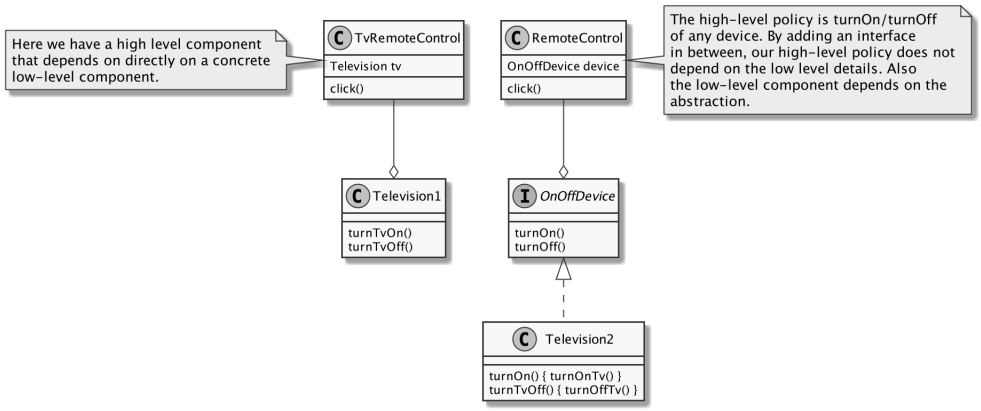


2.9 Dependency Inversion Principle

High-level modules should not depend on low-level modules

- Typical Object-Oriented Thinking: We take a problem, and we factor it into a high-level set of components (the policy-makers) that depend on low-level components, who are carrying out the real work. The problem with this approach is that is tightly couples our high-level components to our low-level ones.

- A symptom of tight coupling is that it's difficult to reuse the high-level components with a different implementation of the low-level component.
- Dependency Inversion Principle:
  1. High-level components should not depend on low-level components. Both should depend on abstractions.
  2. Abstractions should not depend on details. Details should depend on abstractions.
- What is high-level policy? It is the abstraction that underlies the application. i.e. In our example, it is a remote control that can control anything.



- This principle frees our high-level components from being dependent on the details of the low-level components.
- It helps design software that is reusable and resilient against changes.
- Abstraction allow details to remain isolated from each other.

3 Design Patterns

Design Patters are general solutions to common object-oriented problems. The goal of design patterns is to create object-oriented software that is more flexible, maintainable and reliable.

3.1 The Factory Method Pattern

- Problem: If we program to an Implementation we get locked into a specific type. Our code then needs modification if our set of concrete type get extended. When creating object using the "new" operator we are forcing ourselves to a concrete implementation. e.g. Duck duck = new MallardDuck();

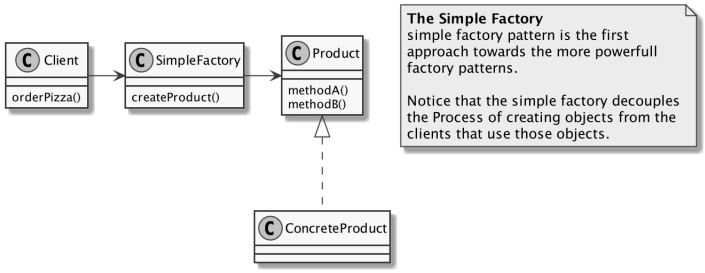
- Often we end up writing code with conditional logic to determine which concrete object to create:

```
Duck duck
if (picnic) {
    duck = new MallardDuck();
} else if (hunting) {
    duck = new DecoyDuck();
} else if (inBathTub) {
    duck = new MallardDuck();
}
```

- Here we make run-time decisions for which class to instantiate. When we see this code we know that when requirement changes, and we want to add new types, we need to open up this code and change it. This violates the Open/Close Principle.
- To solve this problem we look at what varies, the creation, and encapsulate it using a factory pattern.
- Three different factory patterns exist
  1. The Simple Factory
  2. The Factory Method
  3. The Abstract Factory

3.1.1 The Simple Factory Pattern

- With the Simple Factory pattern, we move the code that's responsible for making a product out of the main part of the code and into a separate factory object. The client uses the factory to choose which product to create.



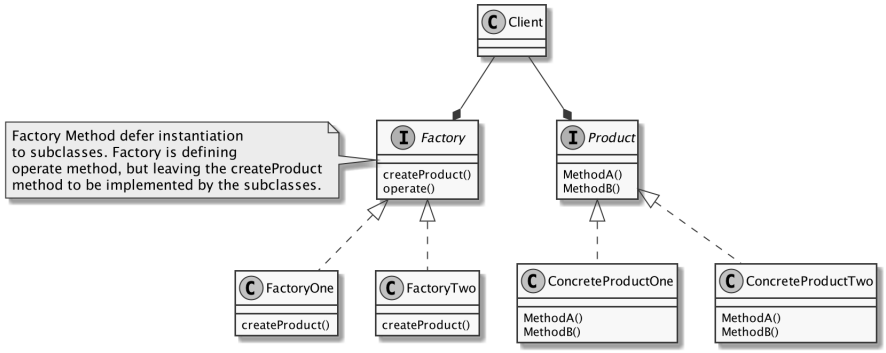

```
class Product {
public:
    virtual void doSomethingWithProduct() = 0;
};

class RegularProduct : public Product {
public:
    void doSomethingWithProduct() override {
        std::cout << "I am regular" << std::endl;
    };
};

class SpecialProduct : public Product {
public:
    void doSomethingWithProduct() override {
        std::cout << "I am special" << std::endl;
    };
};

class ProductFactory {
public:
    std::unique_ptr<Product> MakeProduct(std::string
            type) {
        if (type == "regular") {
            return std::make_unique<RegularProduct>();
        } else if (type == "special") {
            return std::make_unique<SpecialProduct>();
        }
    }
};
```

```
30
31  TEST(SimpleFactory, simple) {
32      auto factory = ProductFactory();
33      std::vector<std::unique_ptr<Product>> products;
34      products.emplace_back(factory.MakeProduct("regular"));
35      products.emplace_back(factory.MakeProduct("special"));
36
37      for (auto& p : products){
38          p->doSomethingWithProduct();
39      }
40  }
```

- The simple factory allows us to decouple the process of creating objects from the clients that uses those objects. The simple factory code is the only code that referece to the concrete products. All other code refereces to the product interface. We can add concrete products whenever we want, without impacting any code, except the factory.

3.1.2 The Factory Method Pattern

- In Factory Method we have a Factory class which operates on products we create but leaves the decision about which product to make to the Factory subclasses (instead of deciding itself, like Simple Factory).



- Factory Method embodies several design principles, include **encapsulate what varies**; i.e. factory encapsulate the different products that need to get made as well as the code to creates them; **Program to interace, not implementation**; i.e. we are programming to product interface in our client code, so we can change the product concrete types without having to change the client code. **Open/Close principle**; i.e. Our factory code is open to new types of products but the client is closed for modification. This means we can extend the system with fewer oppertunities for bugs to arise. **Depend on abstractions**. i.e. All code that refers to concrete products is taken completely out of the client, ecapsulated into a factory class which makes the client refers only to the pizza interface, not the concrete types of product.

3.2 The Abstract Factory

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

- We use the abstract factory pattern whenever we have a system that must be independent of how its products are created and represented, and the system is configured with one of multiple families of products.
- While Factory Method relies on inheritance, Abstract Factory relies on composition. Object creation is implemented in methods exposed in the factory interface.

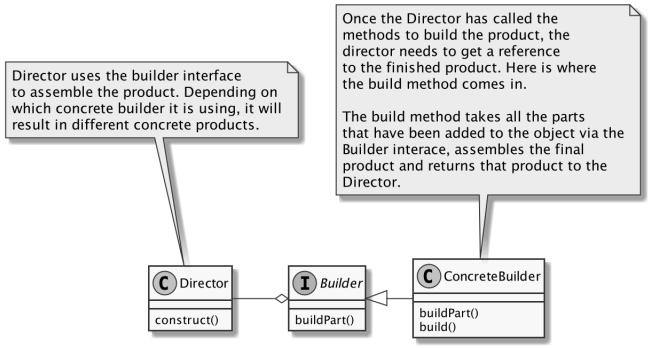
```
// animal.h
class animal {
public:
    virtual ~animal() = default;
    virtual std::string walk() = 0;
    virtual std::string talk() = 0;
};

struct animalItemFactory {
    virtual ~animalItemFactory() = default;
    virtual std::unique_ptr<animal> make() = 0;
};

// ape.h
class ape : public animal {
public:
    std::string walk() override;

    std::string talk() override;
};

struct apeFactory : animalItemFactory {
    std::unique_ptr<animal> make() override {
        return std::make_unique<ape>();
    }
};

// ape.cpp
std::string ape::talk() {
    std::string s = "woegawoega";
    std::cout << s << std::endl;
    return s;
}

std::string ape::walk() {
    std::string s = "with 2 legs";
    std::cout << s << std::endl;
    return s;
}

// dog.h
class dog : public animal {
public:
    std::string walk() override;

    std::string talk() override;
};

struct dogFactory : animalItemFactory {
    std::unique_ptr<animal> make() override {
        return std::make_unique<dog>();
    }
};

// dog.cpp
std::string dog::talk() {
    std::string s = "woof";
    std::cout << s << std::endl;
    return s;
}

std::string dog::walk() {
    std::string s = "with 4 legs";
    std::cout << s << std::endl;
    return s;
}

// animalFactory.h
class animalFactory {
public:
    static constexpr const auto kDog = "dog";
    static constexpr const auto kApe = "ape";

    static std::unique_ptr<animal> make(const std::string
            &name) {
        return factories_[name]->make();
    }

    static void registers(const std::string &name,
            std::unique_ptr<animalItemFactory> factory) {
        factories_[name] = std::move(factory);
    }

private:
    static std::map<std::string,
            std::unique_ptr<animalItemFactory>> factories_;
};

// animalFactory.cpp
static std::map<std::string,
        std::unique_ptr<animalItemFactory>> init() {
    std::map<std::string,
            std::unique_ptr<animalItemFactory>> mp;
    mp[animalFactory::kDog] =
            std::make_unique<dogFactory>();
    mp[animalFactory::kApe] =
            std::make_unique<apeFactory>();
    return mp;
}

std::map<std::string, std::unique_ptr<animalItemFactory>>
        animalFactory::factories_ = init();

// client
class zoo {
    zoo() : animal_(animalFactory::make("dog")) {
        animal_->talk();
        animal_->walk();
    }

    std::unique_ptr<animal> animal_;
};
```

3.3 The Builder Pattern

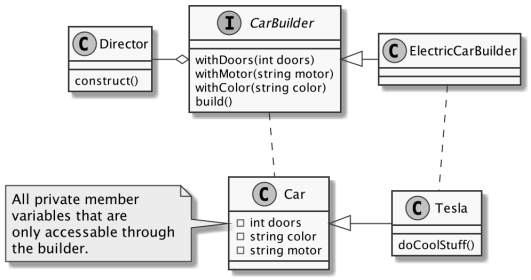
Separate the construction of a complex object from its representation so that the same construction process can create different representations. This pattern is concerend with encapsulating the complexities of how we build an individual object.

- Builder encapsulates the varying details of how a product is assembled, and moving the complexity to the concrete builders. The code is easy to

write, read and understand.



- The intent of the builder pattern is to separate the construction of a complex object from its representation, so that the same construction process can be used in different representations of a product.
- By creating an interface, we are building in flexibility to the builders we use. It keeps the director and client closed for modification.
- Builder's lets us vary a product's internal representation (structure of the product)
- Using builders gives us fine control over the construction process by splitting the process into steps and giving control of that process to the director.
- Example: A Car builder.



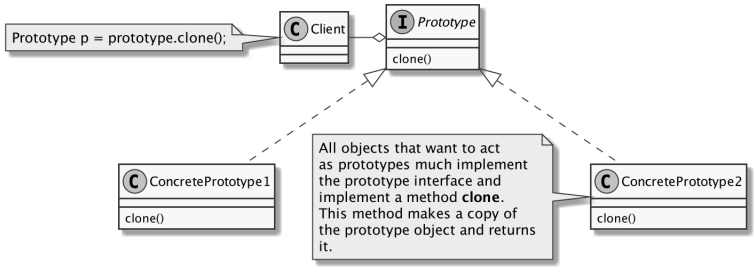

```
1   class CarBuilder; // forward declare
2   class ElectricCarBuilder; // forward declare
3
4   class Car {
5   protected:
6       int doors;
7       string color;
8       string motor;
9   };
10
11  class CarBuilder {
12      virtual CarBuilder& withDoors(int doors) = 0;
13      virtual CarBuilder& withMotor(string motor) = 0;
14      virtual CarBuilder& withColor(string color) = 0;
15  };
16
17  class ElectricCar : public Car {
18  public:
19      static ElectricCarBuilder createBuilder();
20      friend ElectricCarBuilder;
21
22      friend std::ostream& operator << (std::ostream &os,
            const ElectricCar &obj) {
23          return os << "doors: " << obj.doors
24                    << " color: " << obj.color
25                    << " motor: " << obj.motor;
26      }
27  };
28
29  class ElectricCarBuilder : CarBuilder {
30  private:
31      typedef ElectricCarBuilder Self;
32      ElectricCar c;
33  public:
```

```
34      ElectricCar& electric_car;
35
36      ElectricCarBuilder(ElectricCar& e_car) :
                electric_car(e_car){}
37      ElectricCarBuilder() : electric_car(c) { }
38
39      // setters
40      Self& withDoors(int doors) override {
41          std::cout << __func__ << std::endl;
42          electric_car.doors = doors;
43          return *this;
44      }
45      Self& withMotor(string motor) override {
46          std::cout << __func__ << std::endl;
47          electric_car.motor = motor;
48          return *this;
49      }
50      Self& withColor(string color) override {
51          std::cout << __func__ << std::endl;
52          electric_car.color = color;
53          return *this;
54      }
55      ElectricCar build(){
56          return c;
57      }
58  };
59
60  ElectricCarBuilder ElectricCar::createBuilder() {
61      return ElectricCarBuilder();
62  };
63
64  TEST(builder, simple_facet_builder) {
65      auto builder = ElectricCar::createBuilder();
66      auto e_car = builder
67              .withDoors(4)
68              .withMotor("Electric")
69              .withColor("Red")
70              .build();
71  }
```

3.4 The Prototype Pattern

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

- We encapsulate the creation of a new object inside an object we call the prototype object. We create new objects of the same kind by cloning (copying) that prototype object.
- The main benifit is that we separate the client from the details of object creation. We we need flixibility to vary the types of objects that we create, we encapsulate what varies by moving the object creating code out of the client and into another part of the system. With the prototype pattern we move the creating code into a method of the object itself.
- Why not factory/new?
    1. When we copy an existing object, we get the complex setup for free.
    2. We don't need to know the concrete class of the object that we are creating.
    3. The client is independent of how an object is created.
- We are programming to the interface, not the implementation. So the client doesn't need to know the concrete class of the prototype it's using to create new objects, or details of how new objects of that type are created.



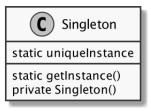
- Prototype is particularly helpful when we're creating objects that are complex or expensive to create new. We have an existing object we can leverage, so by copying that existing object, we may be albe to create new objects more efficiently.
- Example: Contact Prototype.

```
struct Address {
    string street;
    string city;
    int suite;
};

struct Contact {
    string name;
    Address* work_address;

    Contact(const string& name, Address* const
            work_address)
            : name(name),
              work_address(new Address(*work_address)) {}

    ~Contact() {
        delete work_address;
    }

    Contact(const Contact& other)
            : name(other.name),
              work_address(new
                      Address(*other.work_address)) {}
};

struct EmployeeFactory {
public:
    // these are the address prototypes
    static Contact main, aux;

    // we clone the prototype by passing the prototype
            into the copy constructor.
    static unique_ptr<Contact>
            NewMainOfficeEmployee(string name, int suite) {
        return NewEmployee(name, suite, main);
    }
    static unique_ptr<Contact>
            NewAuxOfficeEmployee(string name, int suite) {
        return NewEmployee(name, suite, aux);
    }
private:
    static unique_ptr<Contact> NewEmployee (string name,
            int suite, Contact& proto) {
        auto result = make_unique<Contact>(proto);
        result->name = name;
        result->work_address->suite = suite;
        return result;
    }
};

// these are the contact prototypes
Contact EmployeeFactory::main{"", new Address{"123
        EsatDr", "London", 0}};
Contact EmployeeFactory::aux{"", new Address{"123B
        EsatDr", "London", 0}};

TEST(prototype, factory_prototype) {
    auto john =
            EmployeeFactory::NewMainOfficeEmployee("John",
            100);
    auto jane =
            EmployeeFactory::NewAuxOfficeEmployee("Jane",
            123);

    EXPECT_EQ(jane->work_address->suite, 123);
    EXPECT_EQ(john->work_address->suite, 100);
}
```

3.5 The Singleton Pattern

Ensure a class only has one instance, and provide a global point of access to it.

- Problem:
  1. We need to ensure only one instnace of a class exist.
  2. Instance must be easily accessible to clients.
- Encapsulates the code that is managing a resource.



- The singleton's sole responsibility is to manage a resource.

```
class SingletonDatabase {
private:
    SingletonDatabase(){
        // 1. private constructor.
        std::cout << "initializing database" << std::endl;

        std::ifstream ifs("capitals.txt");

        std::string s, s2;
        while (std::getline(ifs, s)) {
            getline(ifs, s2);
            int pop = boost::lexical_cast<int>(s2);
            capitals[s] = pop;
        }

        instance_count++;
```

```
    }
    std::map<std::string, int> capitals;

    static SingletonDatabase* instance;

public:
    static int instance_count;
    // 2. delete the copy constructor
    SingletonDatabase(SingletonDatabase const&) = delete;
    // 3. delete the copy assignment
    void operator=(SingletonDatabase const&) = delete;

    static SingletonDatabase& get() {
        // Safe Singleton: this static invocation will
                only happen once!
        // So thread safe!
        static SingletonDatabase db;
        return db;
    }

    int get_population(const std::string& name) {
        return capitals[name];
    }
};

int SingletonDatabase::instance_count = 0;

TEST(singleton_pattern, naive_approach) {
    auto& db1 = SingletonDatabase::get();
    auto& db2 = SingletonDatabase::get();
    EXPECT_EQ(1, db1.instance_count);
    EXPECT_EQ(1, db2.instance_count);
}
```

3.6 The Chain of Responsibility Pattern

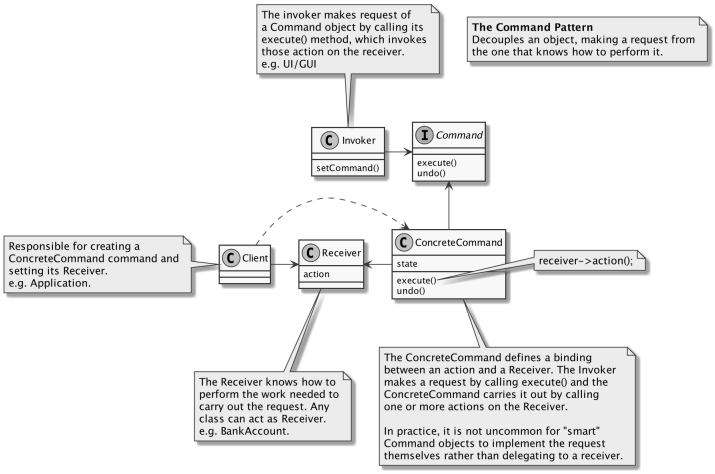
...

- ...

3.7 The Command Pattern

This pattern encapsulates a request as an object, thereby letting you parameterize other objects with different request, queue or log requests, and support undoable operations.

- This pattern encapsulates a request by binding together a set of actions on a specific reveicer. It packages the action and the reveicer up into an object that exposes just one method, execute. When called execute() causes the action to be invoked on the receiver.



- this pattern:
  1. decouples what is done, from who does it
  2. decouples what is done, from when its done

```
class BankAccount {
public:
    BankAccount()
            : balance_(0),
              overdraft_limit_(-500) {
    }

    void Deposit(int amount) {
        balance_ += amount;
    }

    void Withdraw(int amount) {
        if ((balance_ - amount) >= overdraft_limit_) {
            balance_ -= amount;
        }
    }

    int balance_;
    int overdraft_limit_;
};

class ICommand {
public:
    virtual ~ICommand() = default;
    virtual void Execute() const = 0;
    virtual void Undo() const = 0;
};

class Command : ICommand {
public:
    BankAccount &account_;
    int amount_;
    enum Action {
        deposit, withdraw
    } action_;

    Command(BankAccount &account, const Action action,
            int amount)
            : account_(account),
              amount_(amount),
              action_(action) {
    }

    void Execute() const override {
        switch (action_) {
            case deposit:
                account_.Deposit(amount_);
                break;
            case withdraw:
                account_.Withdraw(amount_);
                break;
            default:
                break;
        }
    }

    void Undo() const override {
        switch (action_) {
            case withdraw:
                account_.Deposit(amount_);
                break;
            case deposit:
                account_.Withdraw(amount_);
                break;
            default:
                break;
        }
    }
};

TEST(command, bankAccount) {
    auto ba = BankAccount();
    EXPECT_EQ(ba.balance_, 0);
    EXPECT_EQ(ba.overdraft_limit_, -500);

    // we have encapsulated everything
    // that needs to be done on the back account.
    std::vector<Command> commands; // list of commands
    commands.emplace_back(Command(ba, Command::deposit,
            100));
    commands.emplace_back(Command(ba, Command::withdraw,
            200));

    for (auto &c : commands) {
        c.Execute();
    }
    EXPECT_EQ(ba.balance_, -100);
}
```

3.8 The Interpreter Pattern

...

- ...

3.9 The Iterator Pattern

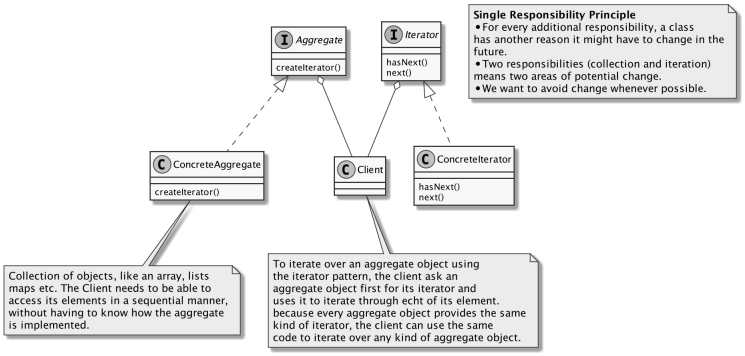
This pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

- Problem: There are many ways to store objects and data structure e.g. using collection type: array, vector, list, linkedlist, set etc.

    1. When we have a function that needs to go through the items, the function needs to change if we choose a different collection type. e.g. changing from vector to list.
    2. What if we need to write code that operates one several collection types? We need to branch out for every collection type.

- The Iterator pattern makes use of the Single Responsibility Principle, and separates the Aggregation and the Iteration responsibilities of a collection

into two different classes.



- In C++, Container member functions Requirement:
  1. begin() points to first element in the container, if empty, is equal to end();
  2. end() points to the element immediately after the last element.
- In C++, Iterators operators Requirements:
  1. operator != must return false if two iterators point to the same element.
  2. operator * (dereferencing) must return a reference to (or a copy of) the data the iterator points to.
  3. operators ++ gets the iterator to point to the next element.
- Example: Binary Tree Iterator:

```
template<typename T>
struct BinaryTree;

template<typename T>
struct Node {
    T value = T();
    Node<T> *left = nullptr;
    Node<T> *right = nullptr;
    Node<T> *parent = nullptr;
    BinaryTree<T> *tree = nullptr;

    explicit Node(const T &value)
        : value(value) {
    }

    ~Node() {
        if (left) {
            delete left;
        }
        if (right) {
            delete right;
        }
    }

    Node(const T &value, Node<T> *const left, Node<T>
            *const right)
        : value(value), left(left), right(right) {
        this->left->tree = this->right->tree = tree;
        this->left->parent = this->right->parent = this;
    }

    void set_tree(BinaryTree<T> *t) {
        tree = t;
        if (left) {
            left->set_tree(t);
        }
        if (right) {
            right->set_tree(t);
        }
    }
};

template<typename T>
struct BinaryTreeIterator {
    Node<T> *current;

    explicit BinaryTreeIterator(Node<T> *const current)
            : current(current) {
    }

    bool operator!=(const BinaryTreeIterator<T> &other) {
        return current != other.current;
    }

    Node<T> &operator*() {
        return *current;
    }

    BinaryTreeIterator<T> &operator++() {
        if (current->right) {
            current = current->right;
            while (current->left) {
                current = current->left;
            }
        } else {
            Node<T> *p = current->parent;
```

```
            while (p && current == p->right) {
                current = p;
                p = p->parent;
            }
            current = p;
        }
        return *this;
    }
};

template<typename T>
struct BinaryTree {
    Node<T> *root = nullptr;

    typedef BinaryTreeIterator<T> iterator;

    explicit BinaryTree(Node<T> *const root)
            : root(root) {
        root->set_tree(this);
    }

    ~BinaryTree() {
        if (root) {
            delete root;
        }
    }

    iterator end() {
        return iterator{nullptr};
    }

    iterator begin() {
        Node<T> *n = root;
        if (n) {
            while (n->left) {
                n = n->left;
            }
        }
        return iterator{n};
    }
};

TEST(iterator, tree) {
    BinaryTree<std::string> family{
        new Node<std::string>{"me",
            new Node<std::string>{"mother",
                new Node<std::string>{"grandma"},
                new Node<std::string>{"grandpa"}},
            new Node<std::string>{"father"}
        }
    };

    for (auto it = family.begin(); it != family.end();
            ++it) {
        std::cout << (*it).value << std::endl;
    }
}
```

3.10 The Mediator Pattern

...

- ...

3.11 The Memento Pattern

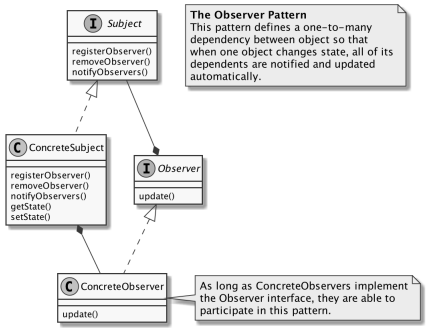
...

- ...

3.12 The Observer Pattern

This pattern defines a one-to-many dependency between object so that when one object changes state, all of its dependents are notified and updated automatically.

- This pattern exemplifies the loosely coupling design principle. Any changes we make to the subject or the observer never affect the other.



- Subjects and Observers are loosely coupled. They interact, but have little knowledge of each other.
    1. Subject knows that the Observer implements a specific interface.
    2. Subject doesn't need to know the concrete class of the Observer.
    3. Observer can be added, removed or replaced at any time. Subject doesn't care; It keeps doing its job.
    4. Any changes made to the Subject or the Observer never affect the other.
- As long as ConcreteObservers implement the Observer interface, they are able to participate in this pattern.
- Example:

```
class Observer;

class Subject {
public:
    virtual void register(std::shared_ptr<Observer> o) =0;
    virtual void remove(std::shared_ptr<Observer> o) = 0;
    virtual void notify() = 0;
};

class SimpleSubject : public Subject {
public:
    void register(std::shared_ptr<Observer> o) override {
        // add observer to list
        observers_.push_back(o);
    };

    void remove(std::shared_ptr<Observer> o) override {
        // remove observer from list
        observers_.erase(
               std::remove(std::begin(observers_),
                      std::end(observers_), o),
                      std::end(observers_));
    };

    void notify() override {
        std::for_each(std::begin(observers_),
               std::end(observers_),
               [this](auto observer){
            observer->update(value_);
        });
    };

    void setValue(int value){
        value_ = value;
        notify();
    }

private:
    int value_ = 0;
    std::vector<std::shared_ptr<Observer>> observers_;
};

class Observer {
public:
    virtual void update(int value) = 0;
};

class SimpleObserver : public Observer {
public:
    void update(int value) {
        value_ = value;
        std::cout << " updated to " << value_ << std::endl;
    };

    int getValue(){
        return value_;
    }

private:
    int value_ = 0;
};

TEST(observer, observer_example) {
    SimpleSubject simpleSubject;
    auto simpleObserver =
           std::make_shared<SimpleObserver>();

    simpleSubject.register(simpleObserver);
    simpleSubject.setValue(123);
    EXPECT_EQ(123, simpleObserver->getValue());

    simpleSubject.removeObserver(simpleObserver);
    simpleSubject.setValue(234);
    EXPECT_EQ(123, simpleObserver->getValue());
}
```

3.13 The State Pattern

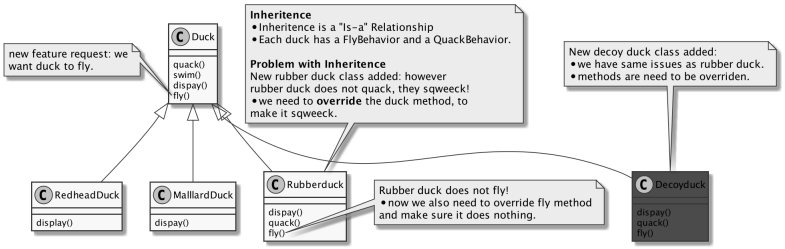
...

- ...

3.14 The Strategy Pattern

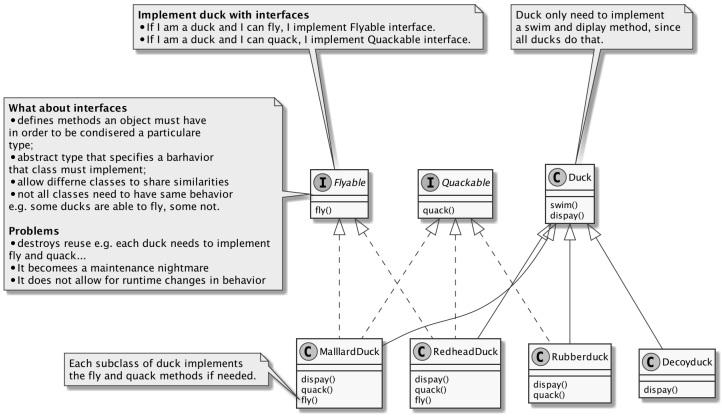
The strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangable. This lets the algorithms vary independent of clients that use them.

- When you overuse inheritance, you can end up with designs that are inflexible and difficult to change. e.g. The Duck simulator



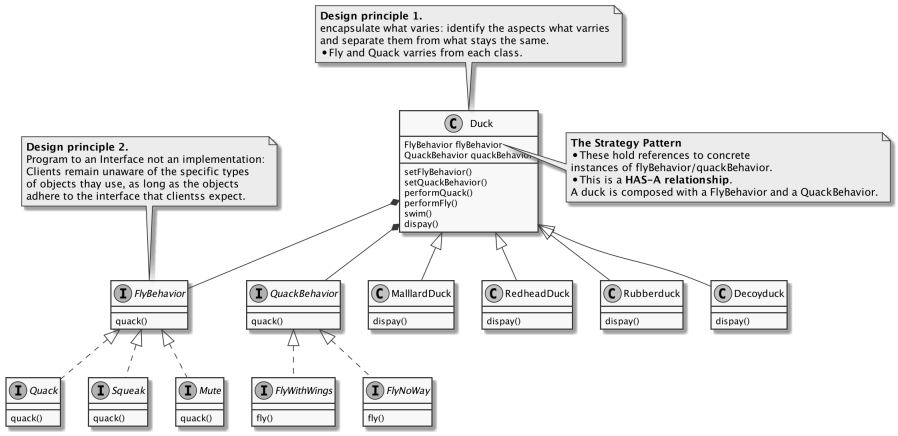
- Inheritence doesn't work well here: Behavior changes across subclasses, and its not appropriate for all subclasses to have all behaviors. i.e. new rubber duck class added, however rubber duck does not quack, they sqweeck. We need to override the duck method, to make it sqweeck.

- When using interfaces instead of inheritence. Interfaces defines methods an object must have in order to be condisered a particular type.

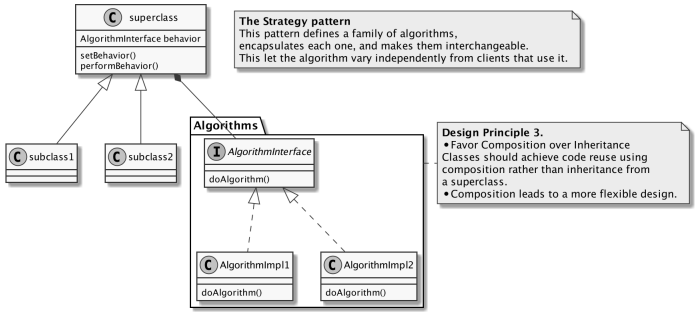


- Interfaces doesn't work well here: Interfaces supply no implementation and destroys code reuse. This makes a maintenance nightmare. i.e. every concrete subclass needs to implement its own flying and quacking behavior.

- Strategy make uses of design principles **encapsulate what varies** and **program to interface not, implementation**. i.e. we pull out varying like the fly() and quack() methods, separate them in a "behavior" classes of which duck has reference to. The behavior classes conform to an interface, so we can interchange the behaviors without changing the duck.



- Using strategy pattern we have a Has-A relationship between objects and its behaviors. by moving the behaviors out of the main inheritance hierarchy, we get the benifit of being able to choose which algorithm each object gets.



- C++ code:

```
class FlyBehavior {
public:
    virtual ~FlyBehavior() = default;
    virtual void fly() = 0;
};

class FlyWithWings : public FlyBehavior {
public:
    void fly() override {
        std::cout << "Flap flap!" << std::endl;
    }
};

class FlyNoWay : public FlyBehavior {
public:
    void fly() override {
        std::cout << "I cant fly!" << std::endl;
    }
};

class QuackBehavior {
public:
    virtual ~QuackBehavior() = default;
    virtual void quack() = 0;
};

class Quack : public QuackBehavior {
public:
    void quack() override {
        std::cout << "Quuuuaaaack!" << std::endl;
    }
};

class Mute : public QuackBehavior {
public:
    void quack() override {
        std::cout << "...........!" << std::endl;
    }
};

class Duck {
public:
    Duck(std::unique_ptr<FlyBehavior> fly_behavior,
         std::unique_ptr<QuackBehavior> quack_behavior)
        : fly_behavior_(move(fly_behavior)),
          quack_behavior_(move(quack_behavior)){}

    virtual ~Duck() = default;
    void doFly() {
        fly_behavior_->fly();
    };
    void doQuack() {
        quack_behavior_->quack();
    };
    virtual void Display() = 0;

```

```
protected:
    std::unique_ptr<FlyBehavior> fly_behavior_;
    std::unique_ptr<QuackBehavior> quack_behavior_;
};

class MallardDuck : public Duck {
public:
    MallardDuck()
            : Duck(std::make_unique<FlyWithWings>(),
                   std::make_unique<Quack>()){}
    MallardDuck(std::unique_ptr<FlyBehavior>
            fly_behavior, std::unique_ptr<QuackBehavior>
            quack_behavior)
            : Duck(move(fly_behavior),
                   move(quack_behavior)){}
    void Display() override {
        std::cout << "Display a Mallard Duck" << std::endl;
    };
};

class DecoyDuck : public Duck {
public:
    DecoyDuck()
            : Duck(std::make_unique<FlyNoWay>(),
                   std::make_unique<Mute>()){}
    void Display() override {
        std::cout << "Display a Decoy Duck" << std::endl;
    };
};

TEST(strategy, duck) {
    MallardDuck mallerd;
    DecoyDuck decoy;

    std::vector<Duck*> ducks;
    ducks.push_back(&mallerd);
    ducks.push_back(&decoy);

    for(auto& duck: ducks){
        duck->doFly();
        duck->doQuack();
        duck->Display();
    }
}
```

3.15 The Template Method Pattern

...

- ...

3.16 The Visitor Pattern

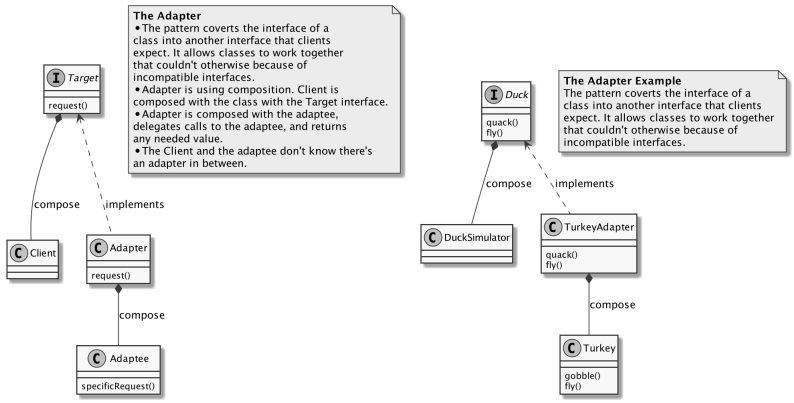
...

- ...

3.17 The Adapter Pattern

The pattern coverts the interface of a class into another interface that clients expect. It allows classes to work together that couldn't otherwise because of incompatible interfaces.

- Client (Duck Simulator) is composed with the class with the Target interface.
- Adapter (TurkeyAdapter) is composed with the adaptee (Turkey) and delegates calls to the adaptee, and returns any needed value.
- The Client (Duck Simulator) and the adaptee (Turkey) don't know there's an adapter (TurkeyAdapter) in between.



- C++ code:

```
1   class Duck {
2   public:
3       virtual void fly() const = 0;
4       virtual void quack() const = 0;
5   };
6
7   class MallardDuck : public Duck{
8   public:
9       MallardDuck(){}
10
11      void fly() const override {
12         std::cout << "fly" << std::endl;
13      };
14      void quack() const override {
15          std::cout << "quack" << std::endl;
16      };
17  };
18
19  // this is the client
20  class DuckSimulator {
21  public:
22      void TestDuck(const Duck& duck) {
23          duck.fly();
24          duck.quack();
25      }
26  };
27
28  TEST(adapter, duck) {
29      DuckSimulator ds;
30      MallardDuck mallardDuck;
31      ds.TestDuck(mallardDuck);
32  }
33
34  class Turkey {
35  public:
36       virtual void gobble() = 0;
37       virtual void fly() = 0;
38  };
39
40  class WildTurkey : public Turkey {
41      void gobble() {
42          std::cout << "gobble" << std::endl;
43      }
44      void fly() {
45          std::cout << "fly" << std::endl;
46      }
47  };
48
49  // we can't use turkeys in the duck simulator because it
        expects a duck interface.
50  // we need to create an adapter.
51  class TurkeyAdapter : public Duck {
52  public:
53      TurkeyAdapter(std::shared_ptr<Turkey> turkey) :
54          turkey_(turkey){};
55      void fly() const override{
56          turkey_->gobble();
57      };
58      void quack() const override {
59          turkey_->fly();
60      };
61  private:
62      std::shared_ptr<Turkey> turkey_;
63  };
64
65  TEST(adapter, turkey_duck_adapter) {
66      DuckSimulator ds;
67      MallardDuck mallardDuck;
68      ds.TestDuck(mallardDuck);
69
70      auto wildTurkey = std::make_shared<WildTurkey>();
71      TurkeyAdapter turkeyAdapter(wildTurkey);
72      ds.TestDuck(turkeyAdapter);
73  }
```

3.18 The Bridge Pattern

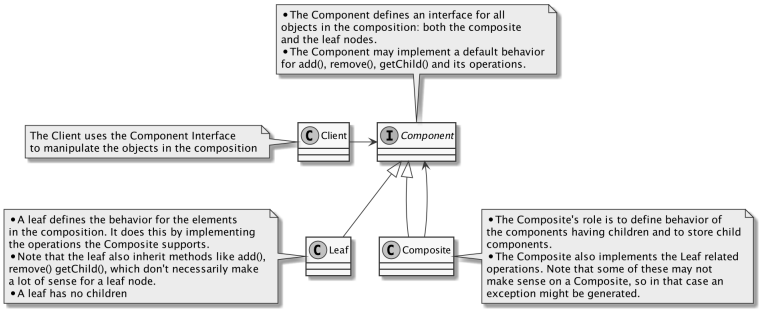
....

- ....

3.19 The Composite Pattern

The Coposite Pattern allows you to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

- Using a composite structure, we can apply tha same operations over both composites and individual objects. In other words, in most cases we can ignore the differences between compositions of objects and individual objects.



- Image a graphical user interface; there you'll oftern find a top level component like a Frame of Panel, containing other components, like menus, text, panes, scrollbars and buttons. So you GUI consist of sevaral parts, but when you display it, you generally think of it as a whole. You tell the top level component to display, and count on that component to display all its parts. We call the compnents that contain other components, "compopsite objects", and components that don't contain other components, "leaf objects".

```
1  struct GraphicsObject {
2      virtual void draw() = 0;
3  };
4
5  struct MyCircle : GraphicsObject {
6      void draw() override {
7          std::cout<< "Cirlce" << std::endl;
8      }
9  };
10
11 struct Group : GraphicsObject {
12     std::string name_;
13     std::vector<GraphicsObject*> objects_;
14
15     Group(const std::string& name) : name_{name} {
16     }
17
18     void draw() override {
19         std::cout << "Group " << name_ << " contains: " <<
                   std::endl;
20         for (auto&& o : objects_) {
21             o->draw();
22         }
23     }
24 };
```

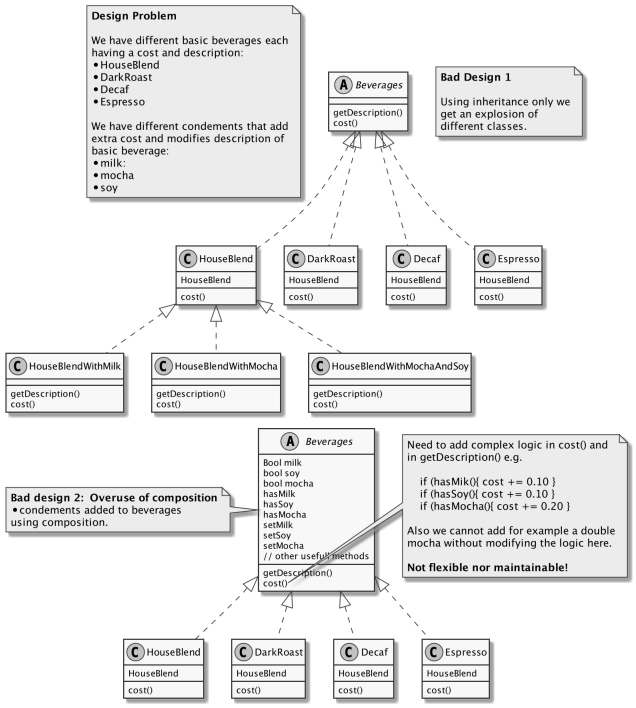
```
25
26 TEST(composite, geometric_shapes) {
27     MyCircle c1, c2, c3, c4;
28
29     Group root("root");
30     root.objects_.push_back(&c1);
31
32     Group subgroup("sub");
33     subgroup.objects_.push_back(&c2);
34     root.objects_.push_back(&subgroup);
35
36     Group subsubgroup("subsub");
37     subsubgroup.objects_.push_back(&c3);
38     subsubgroup.objects_.push_back(&c4);
39     subgroup.objects_.push_back(&subsubgroup);
40
41     root.draw();
42     // this draws all the graphical objects in the
             "tree", from the root down.
43     // we have a uniform interface that treat
44     // both singular objects as groups in a uniform
             manner.
45 }
```

- The clients don't have to worry about whether they're dealing with a composite object or a leaf object. so they dont have to write if statements everywhere to make sure they're calling the right methods on the right objects. Often, they can make one method call and execute an operation over an entire structure.

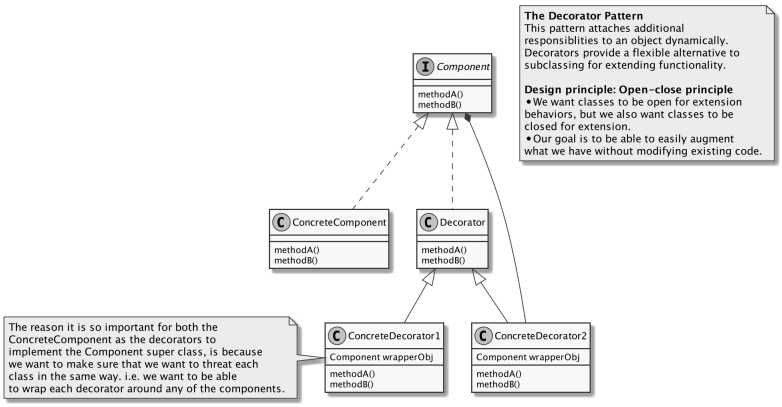
3.20 The Decorator Pattern

This pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

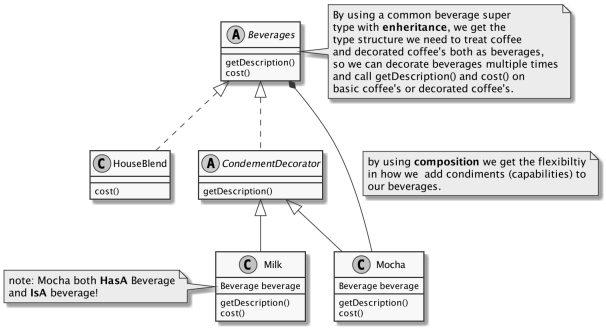
- Here we compare two "beverages and condiments" designs using inheritance and composition.



- Using exesive use of inheritance or composition can lead to unflexible and unmaintainable designs.
- Making use of the decorator pattern we can add additional responsibilities to an object dynamically. Making full use of the open-close principle.



- The Decorator pattern applied to the beverage design:



- C++ code:

```
class Beverage {
protected:
    std::string description_ = "Unknown Beverage";
public:
    virtual std::string getDescription() {
        return description_;
    }
    virtual double cost() = 0;
};

class DarkRoast : public Beverage {
public:
    DarkRoast(){
        description_ = "Dark Roast Coffee";
    }
    double cost() {
        return 0.99;
    }
};

class CondimentDecorator : public Beverage {
    virtual std::string getDescription() = 0;
    virtual double cost() = 0;
};

class Whip : public CondimentDecorator {
private:
    std::shared_ptr<Beverage> beverage_;
public:
    Whip (std::shared_ptr<Beverage> beverage):
            beverage_(beverage) {
    }
    std::string getDescription() {
        return beverage_->getDescription() + ", Whip";
    };
    double cost() {
        return beverage_->cost() + 0.10;
    };
};

TEST(decorator, StarBuzzCoffee) {
    std::shared_ptr<Beverage> beverage =
            std::make_shared<DarkRoast>();
    std::cout << beverage->getDescription() << std::endl;
    std::cout << beverage->cost() << std::endl;

    beverage = std::make_shared<Whip>(beverage);
    std::cout << beverage->getDescription() << std::endl;
    std::cout << beverage->cost() << std::endl;
}
```

- ....

3.21 The Facade Pattern

....

- ....

3.22 The Flyweight Pattern

....

- ....

3.23 The Null Object Pattern

....

- ....

3.24 The Proxy Pattern

The Proxy Pattern provides a surrogate or placeholder for another object to control access to it.

- There are a few ways proxies controll access:
    1. A remote proxy controls access to a remote object;
    2. A virtual proxy controls access to a resource that is exprensive to create;
    3. A protection proxy controls access to a resource bases on access rights;
-