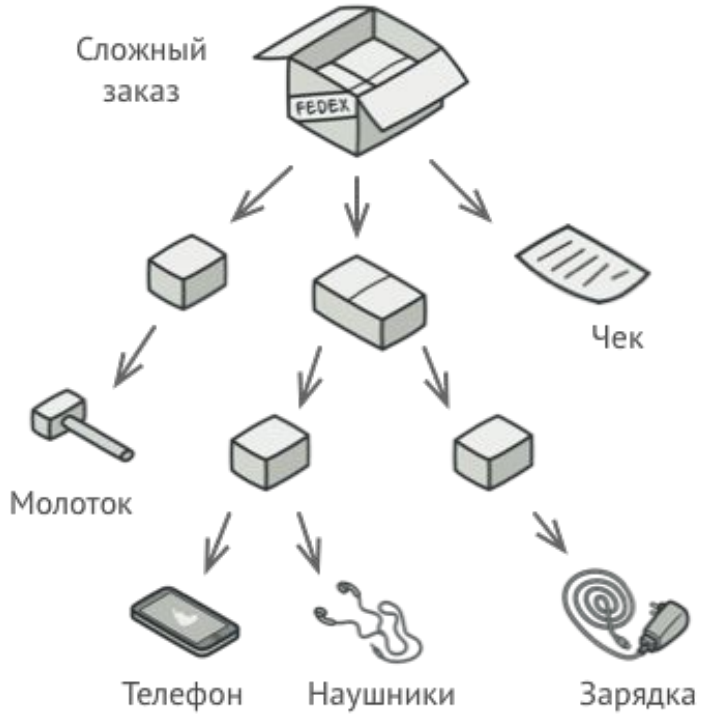
Компоновщик. Декоратор. Заместитель. Поведенческие паттерны.

Компоновщик

Суть паттерна. *Компоновщик* — это структурный паттерн проектирования, который позволяет сгруппировать множество объектов в древовидную структуру, а затем работать с ней так, как будто это единичный объект.

Проблема. Паттерн Компоновщик имеет смысл только тогда, когда основная модель вашей программы может быть структурирована в виде дерева.

Например, есть два объекта: Продукт и Коробка. Коробка может содержать несколько Продуктов и других Коробок поменьше. Те, в свою очередь, тоже содержат либо Продукты, либо Коробки и так далее. Теперь предположим, ваши Продукты и Коробки могут быть частью заказов. Каждый заказ может содержать как простые Продукты без упаковки, так и составные Коробки. Ваша задача состоит в том, чтобы узнать цену всего заказа.



Если решать задачу в лоб, то вам потребуется открыть все коробки заказа, перебрать все продукты и посчитать их суммарную стоимость. Но это слишком хлопотно, так как типы коробок и их содержимое могут быть вам неизвестны. Кроме того, наперёд неизвестно и количество уровней вложенности коробок, поэтому перебрать коробки простым циклом не выйдет.

Решение. Компоновщик предлагает рассматривать Продукт и Коробку через единый интерфейс с общим методом получения стоимости.

Продукт просто вернёт свою цену. Коробка спросит цену каждого предмета внутри себя и вернёт сумму результатов. Если одним из внутренних предметов окажется коробка поменьше, она тоже будет перебирать своё содержимое, и так далее, пока не будут посчитаны все составные части. Для вас, клиента, главное, что теперь не нужно ничего знать о структуре заказов. Вы вызываете метод получения цены, он возвращает цифру, а вы не тонете в горах картона и скотча.

Для военной стратегической игры ”Пунические войны ”, описывающей военное противостояние между Римом и Карфагеном, каждая боевая единица (всадник, лучник) имеет свою собственную разрушающую силу. Эти единицы могут объединяться в группы для образования более сложных военных подразделений, например, римские легионы, которые, в свою очередь, объединяясь, образуют целую армию. *Как рассчитать боевую мощь таких иерархических соединений?*

Паттерн Компоновщик предлагает следующее решение. Он вводит абстрактный базовый класс Component с поведением, общим для всех примитивных и составных объектов. Для случая стратегической игры - это метод getStrength() для подсчета разрушающей силы. Подклассы Primitive and Composite являются производными от класса Component. Составной объект Composite хранит компоненты-потомки абстрактного типа Component, каждый из которых может быть также Composite.

```
#include <vector>
#include <assert.h>

// Component
class Unit{
  public:
    virtual int getStrength() = 0;
    virtual void addUnit(Unit* p) {
      assert( false);
    }
    virtual ~Unit() {}
};

// Primitives
class Archer: public Unit{
  public:
    virtual int getStrength() {
      return 1;
    }
};

class Infantryman: public Unit{
  public:
    virtual int getStrength() {
      return 2;
    }
};
```

```
// Composite
class CompositeUnit: public Unit
{
  public:
    int getStrength() {
      int total = 0;
      for(int i=0; i<c.size(); ++i)
        total += c[i]->getStrength();
      return total;
    }
    void addUnit(Unit* p) {
        c.push_back( p);
    }
    ~CompositeUnit() {
      for(int i=0; i<c.size(); ++i)
        delete c[i];
    }
  private:
   std::vector<Unit*> c;
};
```

```
// Вспомогательная функция для создания легиона
CompositeUnit* createLegion(){
  // Римский легион содержит:
  CompositeUnit* legion = new CompositeUnit;
  // 3000 тяжелых пехотинцев
  for (int i=0; i<3000; ++i)
    legion->addUnit(new Infantryman);
  // 1200 легких пехотинцев
  for (int i=0; i<1200; ++i)
    legion->addUnit(new Archer);
   return legion;
}

int main(){
  // Римская армия состоит из 4-х легионов
  CompositeUnit* army = new CompositeUnit;
  for (int i=0; i<4; ++i)
    army->addUnit( createLegion());
  cout << "Roman army damaging strength is "
       << army->getStrength() << endl;
  delete army;
  return 0;
}
```

Применимость

- **Когда вам нужно представить древовидную структуру объектов.**

Паттерн Компоновщик предлагает хранить в составных объектах ссылки на другие простые или составные объекты. Те, в свою очередь, тоже могут хранить свои вложенные объекты и так далее. В итоге вы можете строить сложную древовидную структуру данных, используя всего две основные разновидности объектов.

- **Когда клиенты должны единообразно трактовать простые и составные объекты.**

Благодаря тому, что простые и составные объекты реализуют общий интерфейс, клиенту безразлично, с каким именно объектом ему предстоит работать.

Преимущества и недостатки

«+»:

- Упрощает архитектуру клиента при работе со сложным деревом компонентов.
- Облегчает добавление новых видов компонентов.

«-»:

- Создаёт слишком общий дизайн классов.

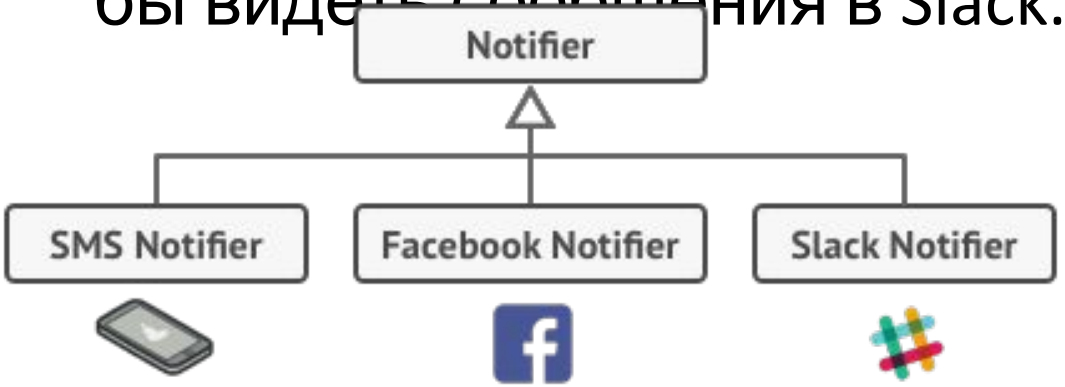
Декоратор

Суть паттерна. *Декоратор* — это структурный паттерн проектирования, который позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные «обёртки».

Проблема. Вы работаете над библиотекой оповещений, которую можно подключать к разнообразным программам, чтобы получать уведомления о важных событиях.

Основой библиотеки является класс Notifier с методом send, который принимает на вход строку-сообщение и высылает её всем администраторам по электронной почте. Сторонняя программа должна создать и настроить этот объект, указав кому отправлять оповещения, а затем использовать его каждый раз, когда что-то случается.

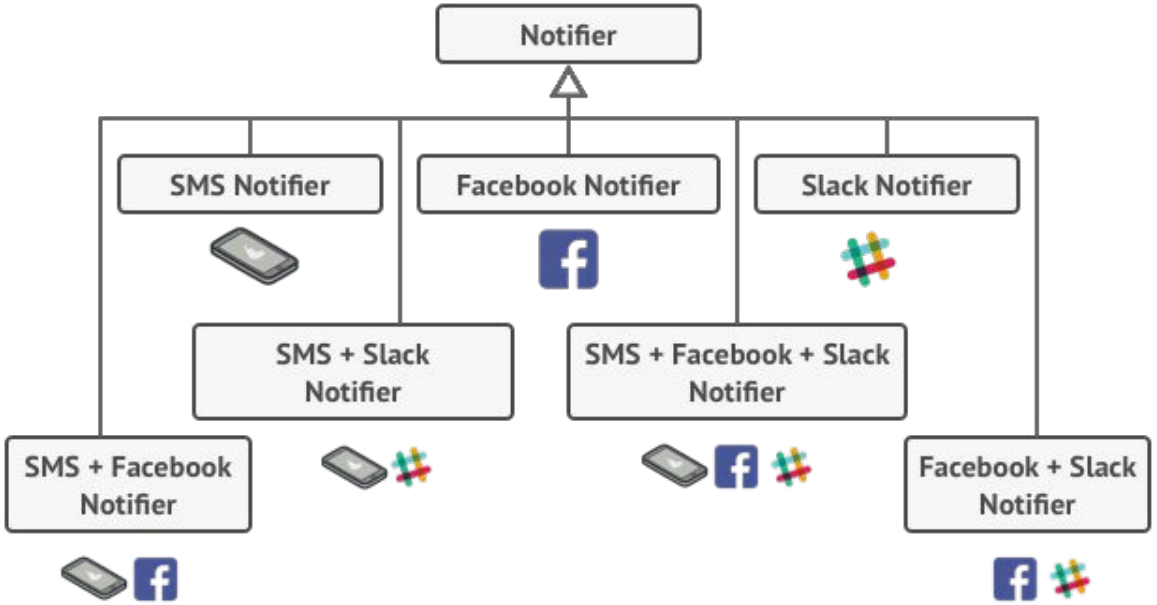
В какой-то момент стало понятно, что одних email-оповещений пользователям мало. Некоторые из них хотели бы получать извещения о критических проблемах через SMS. Другие хотели бы получать их в виде сообщений Facebook. Корпоративные пользователи хотели бы видеть сообщения в Slack.



Сначала вы добавили каждый из этих типов оповещений в программу, унаследовав их от базового класса Notifier. Теперь пользователь выбирал один из типов оповещений, который и использовался в дальнейшем.

Но затем кто-то резонно спросил, почему нельзя выбрать несколько типов оповещений сразу? Ведь если вдруг в вашем доме начался пожар, вы бы хотели получить оповещения по всем каналам, не так ли?

Вы попытались реализовать все возможные комбинации подклассов оповещений. Но после того как вы добавили первый десяток классов, стало ясно, что такой подход невероятно раздувает код программы.



Итак, нужен какой-то другой способ комбинирования поведения объектов, который не приводит к взрыву количества подклассов.

Решение. Наследование — это первое, что приходит в голову многим программистам, когда нужно расширить какое-то существующее поведение. Но механизм наследования имеет несколько досадных проблем.

- Он **статичен**. Вы не можете изменить поведение существующего объекта. Для этого вам надо создать новый объект, выбрав другой подкласс.
- Он **не разрешает наследовать поведение нескольких классов одновременно**. Из-за этого вам приходится создавать множество подклассов-комбинаций для получения совмещённого поведения.

Одним из способов обойти эти проблемы является механизм *композиции*. Это когда один объект *содержит* ссылку на другой и делегирует ему работу, вместо того чтобы самому *наследовать* его поведение. Как раз на этом принципе построен паттерн Декоратор.

Декоратор имеет альтернативное название — *обёртка*. Оно более точно описывает суть паттерна: вы помещаете целевой объект в другой объект-обёртку, который запускает базовое поведение объекта, а затем добавляет к результату что-то своё.

Оба объекта имеют общий интерфейс, поэтому для пользователя нет никакой разницы, с каким объектом работать — чистым или обёрнутым. Вы можете использовать несколько разных обёрток одновременно — результат будет иметь объединённое поведение всех обёрток сразу.

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

class IComponent {
public:
    virtual void operation() = 0;
    virtual ~IComponent(){}
};

class Component : public IComponent {
public:
    virtual void operation() {
     std::cout<<"World!"<<std::endl;
    }
};
```

```
class DecoratorOne : public IComponent {
    std::shared_ptr<IComponent> m_component;
public:
    DecoratorOne(IComponent* component): m_component(component) {}

    virtual void operation() {
      std::cout << ", ";
      m_component->operation();
    }
};

class DecoratorTwo : public IComponent {
    std::shared_ptr<IComponent> m_component;
public:
    DecoratorTwo(IComponent* component): m_component(component) {}

    virtual void operation() {
      std::cout << "Hello";
      m_component->operation();
    }
};
```

```
int main() {
    DecoratorTwo obj(new DecoratorOne(new
    Component()));
    obj.operation(); // prints "Hello, World!\n"

    return 0;
}
```

Применимость

- **Когда вам нужно добавлять обязанности объектам на лету, незаметно для кода, который их использует.**

Объекты помещают в обёртки, имеющие дополнительные поведения. Обёртки и сами объекты имеют одинаковый интерфейс, поэтому клиентам без разницы, с чем работать — с обычным объектом данных или с обёрнутым.

- **Когда нельзя расширить обязанности объекта с помощью наследования.**

Во многих языках программирования есть ключевое слово final, которое может заблокировать наследование класса. Расширить такие классы можно только с помощью Декоратора.

Преимущества и недостатки

«+»:

- Большая гибкость, чем у наследования.
- Позволяет добавлять обязанности на лету.
- Можно добавлять несколько новых обязанностей сразу.
- Позволяет иметь несколько мелких объектов вместо одного объекта на все случаи жизни.

«-»:

- Трудно конфигурировать многократно обёрнутые объекты.
- Обилие крошечных классов.

Заместитель

Суть паттерна. *Заместитель* — это структурный паттерн проектирования, который позволяет подставлять вместо реальных объектов специальные объекты-заменители. Эти объекты перехватывают вызовы к оригинальному объекту, позволяя сделать что-то *до* или *после* передачи вызова оригиналу.

Проблема. Для чего вообще контролировать доступ к объектам? Рассмотрим такой пример: у вас есть внешний ресурсоёмкий объект, который нужен не все время, а изредка.

Мы могли бы создавать этот объект не в самом начале программы, а только тогда, когда он кому-то реально понадобится. Каждый клиент объекта получил бы некий код отложенной инициализации. Но, вероятно, это привело бы к множественному дублированию кода.

В идеале, этот код хотелось бы поместить прямо в служебный класс, но это не всегда возможно. Например, код класса может находиться в закрытой сторонней библиотеке.

Решение. Паттерн Заместитель предлагает создать новый класс-дублёр, имеющий тот же интерфейс, что и оригинальный служебный объект. При получении запроса от клиента объект-заместитель сам бы создавал экземпляр служебного объекта и переадресовывал бы ему всю реальную работу.

Но в чём же здесь польза? Вы могли бы поместить в класс заместителя какую-то промежуточную логику, которая выполнялась бы до (или после) вызовов этих же методов в настоящем объекте. А благодаря одинаковому интерфейсу, объект-заместитель можно передать в любой код, ожидающий сервисный объект.

```
/**"Subject" */
class IMath {
public:
   virtual double add(double, double) = 0;
   virtual double sub(double, double) = 0;
   virtual double mul(double, double) = 0;
   virtual double div(double, double) = 0;
};

/* "Real Subject"*/
class Math : public IMath
{
public:
   virtual double add(double x, double y)  {return x + y;}
   virtual double sub(double x, double y)  {return x - y;}
   virtual double mul(double x, double y)  {return x * y;}
   virtual double div(double x, double y) {return x / y;}
};
```

```
/*"Proxy Object“*/
class MathProxy : public IMath
{
public:
   MathProxy()
   {
      math = new Math();
   }
   virtual ~MathProxy()
   {
      delete math;
   }
   virtual double add(double x, double y) {return math->add(x, y);}
   virtual double sub(double x, double y)  {return math->sub(x, y);}
   virtual double mul(double x, double y) {return math->mul(x, y);}
   virtual double div(double x, double y) {return math->div(x, y);}

private:
   IMath *math;
};
```

```
#include <iostream>

using std::cout;
using std::endl;

int main()
{
   // Create math proxy
   IMath *proxy = new MathProxy();

   // Do the math
   cout << "4 + 2 = " << proxy->add(4, 2) << endl;
   cout << "4 - 2 = " << proxy->sub(4, 2) << endl;
   cout << "4 * 2 = " << proxy->mul(4, 2) << endl;
   cout << "4 / 2 = " << proxy->div(4, 2) << endl;

   delete proxy;
   return 0;
}
```

Применимость

- **Ленивая инициализация (виртуальный прокси). Когда у вас есть тяжёлый объект, грузящий данные из файловой системы или базы данных.**

Вместо того, чтобы грузить данные сразу после старта программы, можно сэкономить ресурсы и создать объект тогда, когда он действительно понадобится.

- **Защита доступа (защищающий прокси). Когда в программе есть разные типы пользователей, и вам хочется защищать объект от неавторизованного доступа. Например, если ваши объекты — это важная часть операционной системы, а пользователи — сторонние программы (хорошие или вредоносные).**

Прокси может проверять доступ при каждом вызове и передавать выполнение служебному объекту, если доступ разрешён.

- **Локальный запуск сервиса (удалённый прокси). Когда настоящий сервисный объект находится на удалённом сервере.**

В этом случае заместитель транслирует запросы клиента в вызовы по сети в протоколе, понятном удалённому сервису.

- **Логирование запросов (логирующий прокси). Когда требуется хранить историю обращений к сервисному объекту.**

Заместитель может сохранять историю обращения клиента к сервисному объекту.

- **Кеширование объектов («умная» ссылка). Когда нужно кешировать результаты запросов клиентов и управлять их жизненным циклом.**

Заместитель может подсчитывать количество ссылок на сервисный объект, которые были отданы клиенту и остаются активными. Когда все ссылки освобождаются, можно будет освободить и сам сервисный объект (например, закрыть подключение к базе данных).

Кроме того, Заместитель может отслеживать, не менял ли клиент сервисный объект. Это позволит использовать объекты повторно и здóрово экономить ресурсы, особенно если речь идёт о больших прожорливых сервисах.

Преимущества и недостатки

«+»:

- Позволяет контролировать сервисный объект незаметно для клиента.
- Может работать, даже если сервисный объект ещё не создан.
- Может контролировать жизненный цикл служебного объекта.

«-»:

- Усложняет код программы из-за введения дополнительных классов.
- Увеличивает время отклика от сервиса.

Поведенческие паттерны

Паттерны поведения рассматривают вопросы о связях между объектами и распределением обязанностей между ними. Для этого могут использоваться механизмы, основанные как на наследовании, так и на композиции.

- Цепочка Обязанностей (Chain Of Responsibilities)
- Команда (Command)
- Итератор (Iterator)
- Посредник (Mediator)
- Хранитель (Memento)
- Объект Null (Null Object)
- Наблюдатель (Observer)
- Спецификация (Specification)
- Состояние (State)
- Стратегия (Strategy)
- Шаблонный Метод (Template Method)
- Посетитель (Visitor)

Итератор

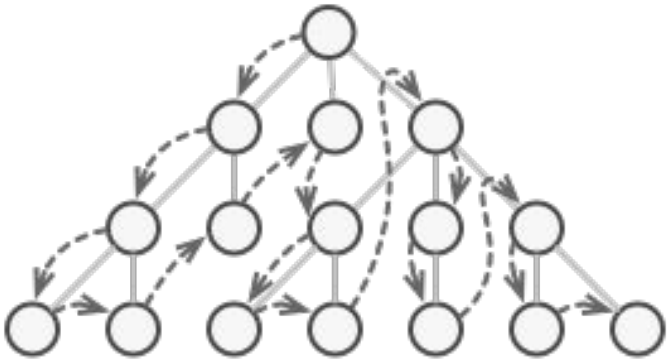
Суть паттерна. *Итератор* — это поведенческий паттерн проектирования, который даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.

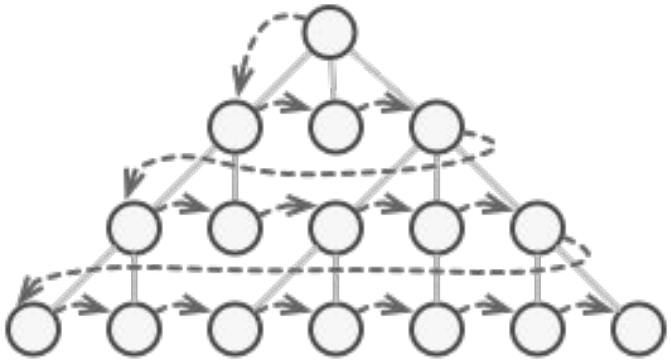
Проблема. Коллекции — самая распространённая структура данных, которую вы можете встретить в программировании. Это набор объектов, собранный в одну кучу по каким-то критериям.

Большинство коллекций выглядят как обычный список элементов. Но есть и экзотические коллекции, построенные на основе деревьев, графов и других сложных структур данных.

Но как бы ни была структурирована коллекция, пользователь должен иметь возможность последовательно обходить её элементы, чтобы проделывать с ними какие-то действия.

Но каким способом следует перемещаться по сложной структуре данных? Например, сегодня может быть достаточным обход дерева в глубину, но завтра потребуется возможность перемещаться по дереву в ширину. А на следующей неделе и того хуже — понадобится обход коллекции в случайном порядке.





Решение. Идея паттерна Итератор состоит в том, чтобы вынести поведение обхода коллекции из самой коллекции в отдельный класс.

Объект-итератор будет отслеживать состояние обхода, текущую позицию в коллекции и сколько элементов ещё осталось обойти. Одну и ту же коллекцию смогут одновременно обходить различные итераторы, а сама коллекция не будет даже знать об этом.

К тому же, если вам понадобится добавить новый способ обхода, вы сможете создать отдельный класс итератора, не изменяя

```
class Stack{
    int items[10];
    int sp;
  public:
    friend class StackIter;
    Stack()
    {
        sp =  - 1;
    }
    void push(int in)
    {
        items[++sp] = in;
    }
    int pop()
    {
        return items[sp--];
    }
    bool isEmpty()
    {
        return (sp ==  - 1);
    }
};
```

```
class StackIter
{
    const Stack &stk;
    int index;
  public:
    StackIter(const Stack &s): stk(s)
    {
        index = 0;
    }
    void operator++()
    {
        index++;
    }
    bool operator()()
    {
        return index != stk.sp + 1;
    }
    int operator *()
    {
        return stk.items[index];
    }
};
```

```
bool operator == (const Stack &l, const Stack &r)
{
  StackIter itl(l), itr(r);
  for (; itl(); ++itl, ++itr)
    if (*itl !=  *itr)
      break;
  return !itl() && !itr();
}

int main()
{
  Stack s1;
  int i;
  for (i = 1; i < 5; i++)
    s1.push(i);
  Stack s2(s1), s3(s1), s4(s1), s5(s1);
  s3.pop();
  s5.pop();
  s4.push(2);
  s5.push(9);
  cout << "1 == 2 is " << (s1 == s2) << endl;
  cout << "1 == 3 is " << (s1 == s3) << endl;
  cout << "1 == 4 is " << (s1 == s4) << endl;
  cout << "1 == 5 is " << (s1 == s5) << endl;
}
```

Применимость

- **Когда у вас есть сложная структура данных, и вы хотите скрыть от клиента детали её реализации (из-за сложности или вопросов безопасности).**

Итератор предоставляет клиенту всего несколько простых методов перебора элементов коллекции. Это не только упрощает доступ к коллекции, но и защищает её данные от неосторожных или злоумышленных действий.

- **Когда вам нужно иметь несколько вариантов обхода одной и той же структуры данных.**

Нетривиальные алгоритмы обхода структуры данных могут иметь довольно объёмный код. Этот код будет захламлять всё вокруг — будь то сам класс коллекции или часть бизнес-логики программы. Применив итератор, вы можете выделить код обхода структуры данных в собственный класс, упростив поддержку остального кода.

- **Когда вам хочется иметь единый интерфейс обхода различных структур данных.**

Итератор позволяет вынести реализации различных вариантов обхода в подклассы. Это позволит легко взаимозаменять объекты итераторов, в зависимости от того, с какой структурой данных приходится работать.

Преимущества и недостатки

«+»:

- Упрощает классы хранения данных.
- Позволяет реализовать различные способы обхода структуры данных.
- Позволяет одновременно перемещаться по структуре данных в разные стороны.

«-»:

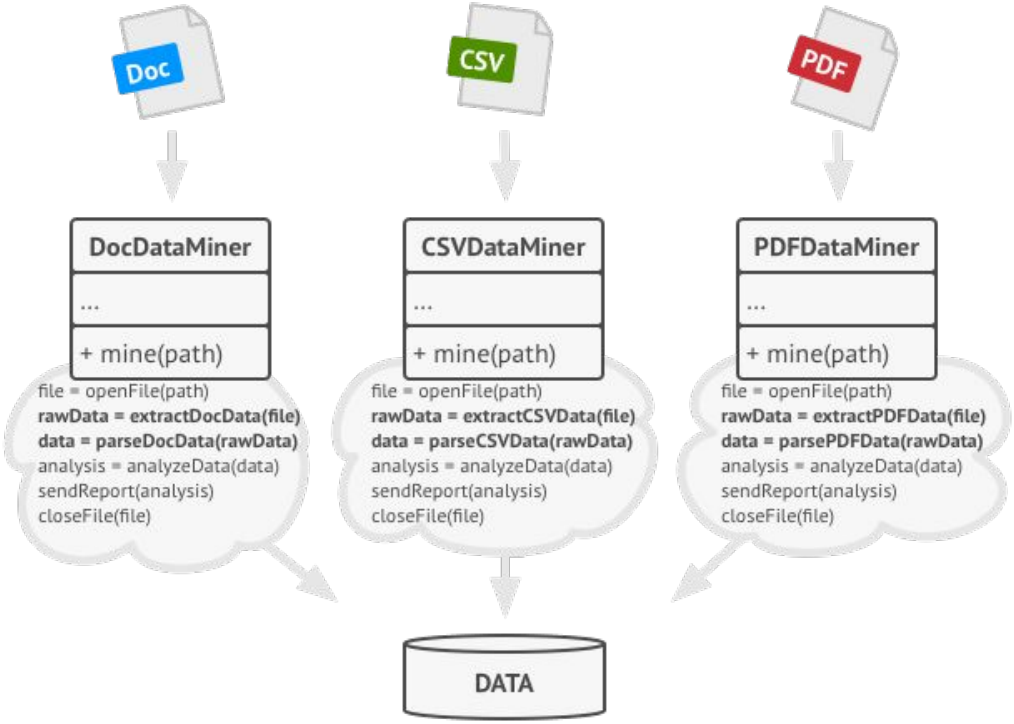
- Не оправдан, если можно обойтись простым циклом.

Шаблонный метод

Суть паттерна. *Шаблонный метод* — это поведенческий паттерн проектирования, который определяет скелет алгоритма, перекладывая ответственность за некоторые его шаги на подклассы. Паттерн позволяет подклассам переопределять шаги алгоритма, не меняя его общей структуры.

Проблема. Вы пишете программу для дата-майнинга в офисных документах. Пользователи будут загружать в неё документы в разных форматах (PDF, DOC, CSV), а программа должна извлекать из них полезную информацию.

В первой версии вы ограничились только обработкой DOC-файлов. В следующей версии добавили поддержку CSV. А через месяц прикрутили работу с PDF-документами.



В какой-то момент вы заметили, что код всех трёх классов обработки документов хоть и отличается в части работы с файлами, но содержат довольно много общего в части самого извлечения данных. Было бы здорово избавится от повторной реализации алгоритма извлечения данных в каждом из классов.

К тому же остальной код, работающий с объектами этих классов, наполнен условиями, проверяющими тип обработчика перед началом работы. Весь этот код можно упростить, если слить все три класса воедино либо свести их к общему интерфейсу.

Решение. Паттерн Шаблонный метод предлагает разбить алгоритм на последовательность шагов, описать эти шаги в отдельных методах и вызывать их в одном *шаблонном* методе друг за другом.

Это позволит подклассам переопределять некоторые шаги алгоритма, оставляя без изменений его структуру и остальные шаги, которые для этого подкласса не так важны.

В нашем примере с дата-майнингом мы можем создать общий базовый класс для всех трёх алгоритмов. Этот класс будет состоять из шаблонного метода, который последовательно вызывает шаги разбора документов.

1. Стандартизуйте основу алгоритма в шаблонном методе базового класса.
2. Для шагов, требующих особенной реализации, определите "замещающие" методы.
3. Производные классы реализуют "замещающие" методы.

```
class Base{
   void a()
   {  cout << "a  "; }
   void c()
   {  cout << "c  ";}
   void e()
   {  cout << "e  ";}
   // 2. Для шагов, требующих особенной реализации, определите
   //    "замещающие" методы.
   virtual void ph1() = 0;
   virtual void ph2() = 0;
  public:
   // 1. Стандартизуйте основу алгоритма в шаблонном методе
   //    базового класса
   void execute()
   {
      a();
      ph1();
      c();
      ph2();
      e();
   }
};
```

```
class One: public Base{
  // 3. Производные классы реализуют "замещающие" методы.
    /*virtual*/void ph1()
   {
      cout << "b  ";
   }
    /*virtual*/void ph2()
   {
      cout << "d  ";
   }
};

class Two: public Base{
    /*virtual*/void ph1()
   {
      cout << "2  ";
   }
    /*virtual*/void ph2()
   {
      cout << "4  ";
   }
};
```

```
int main()
{
  Base *array[] =
  {
     &One(), &Two()
  };
  for (int i = 0; i < 2; i++)
  {
    array[i]->execute();
    cout << '\n';
  }
}
```

Вывод программы: a b c d e a 2 c 4 e

Применимость

- **Когда подклассы должны расширять базовый алгоритм, не меняя его структуры.**

Шаблонный метод позволяет подклассам расширять определённые шаги алгоритма через наследование, не меняя при этом структуру алгоритмов, объявленную в базовом классе.

- **Когда у вас есть несколько классов, делающих одно и то же с незначительными отличиями. Если вы редактируете один класс, то приходится вносить такие же правки и в остальные классы.**

Паттерн шаблонный метод предлагает создать для похожих классов общий суперкласс и оформить в нём главный алгоритм в виде шагов. Отличающиеся шаги можно переопределить в подклассах.

Это позволит убрать дублирование кода в нескольких классах с похожим поведением, но отличающихся в деталях.

Преимущества и недостатки

«+»:

- Облегчает повторное использование кода.

«-»:

- Вы жёстко ограничены скелетом существующего алгоритма.
- Вы можете нарушить *принцип подстановки Барбары Лисков*, изменяя базовое поведение одного из шагов алгоритма через подкласс.
- С ростом количества шагов шаблонный метод становится слишком сложно поддерживать.