Современные средства разработки ПО

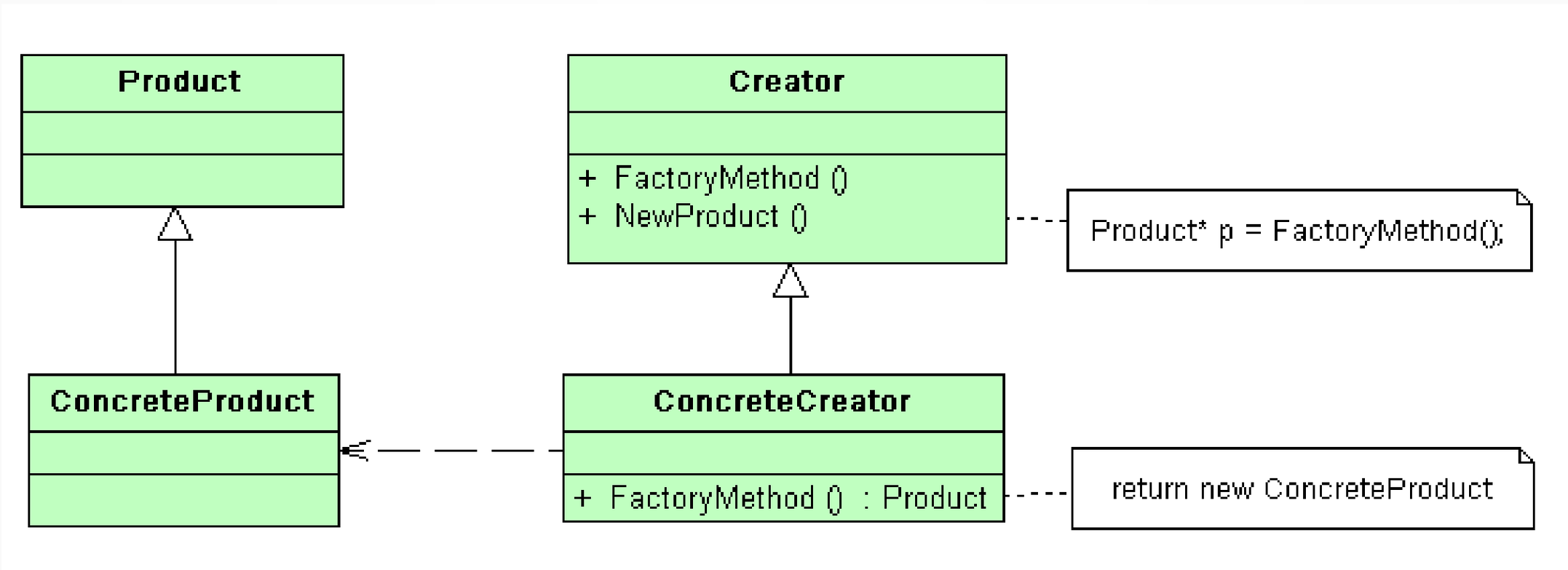
ШП (продолжение).

МГТУ им. Н.Э. Баумана, ИУ-6, Фетисов Михаил Вячеславович
fetisov.michael@bmstu.ru

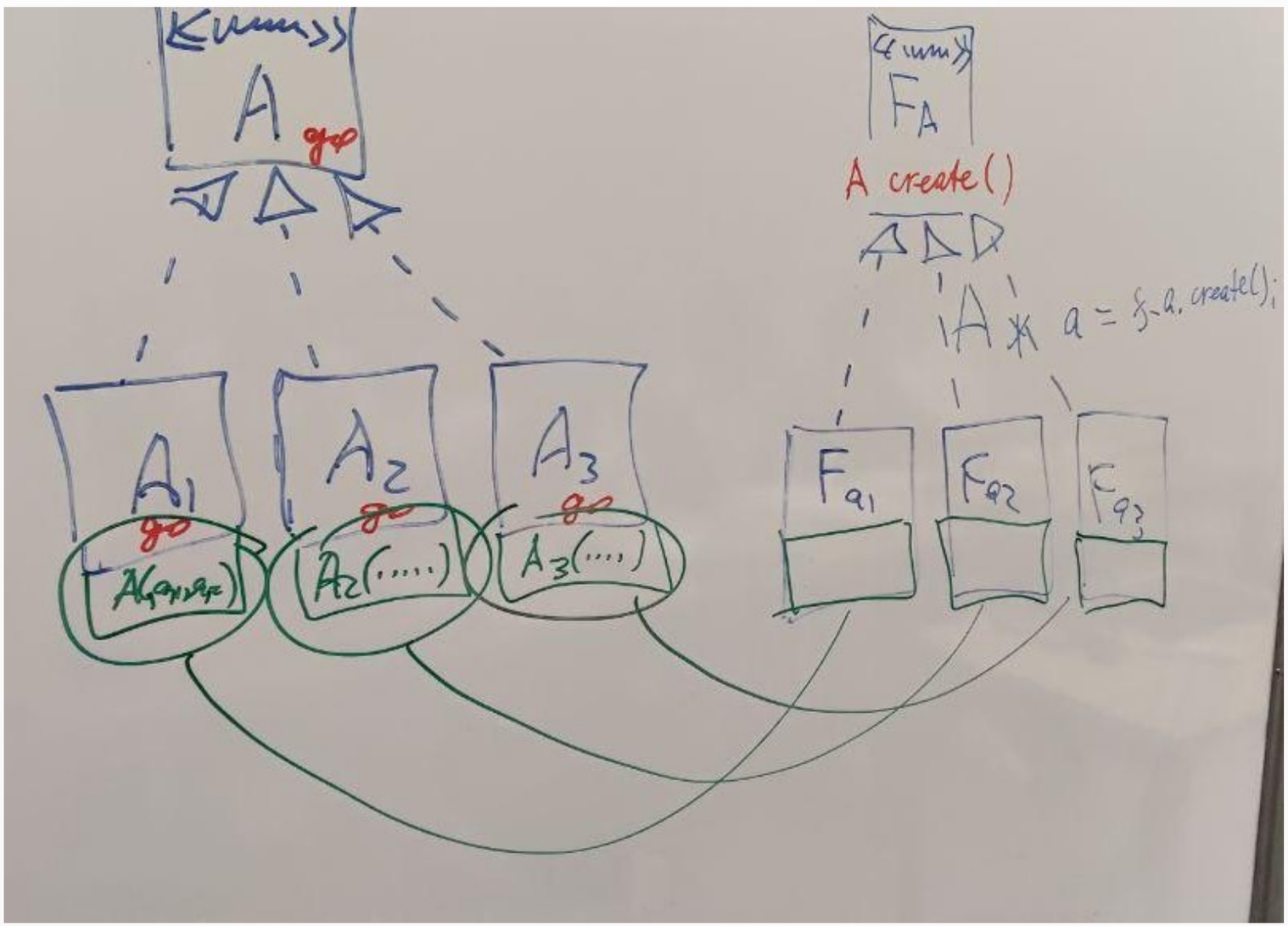
ШП Фабричный метод
Factory method

- Задача:
  - Нужно создавать реализации интерфейса внутри реализации методов других классов.
  - При этом конструкторы реализаций интерфейса могут иметь сложные параметры инициализации.
- **Factory method** – Определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс создавать.

ШП Фабричный метод
Factory method



ШП Фабричный метод
Factory method



```cpp
#include <iostream>
#include <string>
using namespace std;

struct Product{
  virtual string getName() = 0;
  virtual ~Product(){}
};

struct ConcreteProductA: Product{
  string getName(){return "ConcreteProductA";}
};

struct ConcreteProductB: Product{
  string getName(){return "ConcreteProductB";}
};

struct Creator{
  virtual Product* factoryMethod() = 0;
};

struct ConcreteCreatorA: Creator{
  Product* factoryMethod(){return new ConcreteProductA();}
};

struct ConcreteCreatorB: Creator{
  Product* factoryMethod(){return new ConcreteProductB();}
};

int main()
{
  ConcreteCreatorA CreatorA;
  ConcreteCreatorB CreatorB;
  // Массив создателей
  Creator*creators[] = {&CreatorA, &CreatorB};
  //Перебирайте создателей и создавайте продукты
  for(auto&& creator: creators){
    Product* product=creator->factoryMethod();
    cout << product->getName() << endl;
    delete product;
  }
  return 0;
}
```

ШП Фабричный метод
Factory method

- Достоинства:
  - позволяет сделать код создания объектов более универсальным, не привязываясь к конкретным классам (ConcreteProduct), а оперируя лишь общим интерфейсом (Product);
  - позволяет установить связь между параллельными иерархиями классов.
- Недостатки:
  - необходимость создавать наследника Creator для каждого нового типа продукта (ConcreteProduct).

ШП «Прототип» (Prototype)

- Задача:
  - Алгоритму требуется ещё одна реализация интерфейса
- **Prototype –** Определяет интерфейс создания объекта через клонирование другого объекта вместо создания через конструктор.

```
class Meal {
public:
    virtual ~Meal();
    virtual void eat() = 0;
    virtual Meal *clone() const = 0;
    //...
};
class Spaghetti : public Meal {
public:
    Spaghetti( const Spaghetti &);
    void eat();
    Spaghetti *clone() const { return new Spaghetti( *this ); }
    //...
};
```

ШП «Прототип»
Паттерн используется чтобы:

- избежать дополнительных усилий по созданию объекта стандартным путём (имеется в виду использование конструктора, так как в этом случае также будут вызваны конструкторы всей иерархии предков объекта), когда это непозволительно дорого для приложения.
- избежать наследования создателя объекта (object creator) в клиентском приложении, как это делает паттерн abstract factory.

ШП «Отложенная инициализация» (Lazy initialization)

- Задача:
  - Наш алгоритм должен работать с очень большой структурой данных.
  - Чаще всего, работа ведётся только с частью этой структуры, но заранее мы не знаем с какой именно.
- **Lazy initialization** – Объект, инициализируемый во время первого обращения к нему.

```cpp
#include <iostream>
#include <map>
#include <string>

class Fruit {
 public:
  static Fruit* GetFruit(const std::string& type);
  static void PrintCurrentTypes();

 private:
  // Note: constructor private forcing one to use static |GetFruit|.
  Fruit(const std::string& type) : type_(type) {}

  static std::map<std::string, Fruit*> types;

  std::string type_;
};

// static
std::map<std::string, Fruit*> Fruit::types;

// Lazy Factory method, gets the |Fruit| instance associat
// |type|.  Creates new ones as needed.
Fruit* Fruit::GetFruit(const std::string& type) {
  auto [it, inserted] = types.emplace(type, nullptr);
  if (inserted) {
    it->second = new Fruit(type);
  }
  return it->second;
}

// For example purposes to see pattern in action.
void Fruit::PrintCurrentTypes() {
  std::cout << "Number of instances made = " << types.size() << std::endl;
  for (const auto& [type, fruit] : types) {
    std::cout << type << std::endl;
  }
  std::cout << std::endl;
}

int main() {
  Fruit::GetFruit("Banana");
  Fruit::PrintCurrentTypes();

  Fruit::GetFruit("Apple");
  Fruit::PrintCurrentTypes();

  // Returns pre-existing instance from first time |Fruit| with "Banana" was
  // created.
  Fruit::GetFruit("Banana");
  Fruit::PrintCurrentTypes();
}

// OUTPUT:
//
// Number of instances made = 1
// Banana
//
// Number of instances made = 2
// Apple
// Banana
//
```

ШП «Отложенная инициализация» (Lazy initialization)

- Достоинства:
  - Инициализация выполняется только в тех случаях, когда она действительно необходима;
  - Ускоряется начальная инициализация.
- Недостатки:
  - Невозможно явным образом задать порядок инициализации объектов;
  - Возникает задержка при первом обращении к объекту, что может оказаться критичным при параллельном выполнении другой ресурсоёмкой операции. Вследствие этого требуется тщательно просчитывать целесообразность использования «ленивой» инициализации в многопоточных программных системах, особенно — ОС.

ШП Singleton (одиночка)

- Задача:
  - нужно гарантировать, что код работает с общим (глобальным) объектом
- Порождающий шаблон проектирования, гарантирующий, что в однопроцессном приложении будет единственный экземпляр некоторого класса, и предоставляющий глобальную точку доступа к этому экземпляру

ШП Singleton (одиночка) Применение

- ШП полезен, если:
  - должен быть ровно один экземпляр некоторого класса, легко доступный всем клиентам;
  - единственный экземпляр должен расширяться путём порождения подклассов, и клиентам нужно иметь возможность работать с расширенным экземпляром без модификации своего кода.

ШП Singleton (одиночка) Особенности

- Плюс:
  - контролируемый доступ к единственному экземпляру.
- Минусы:
  - глобальные объекты могут быть вредны для объектного программирования, в некоторых случаях приводят к созданию немасштабируемого проекта;
  - усложняет написание модульных тестов и следование TDD.

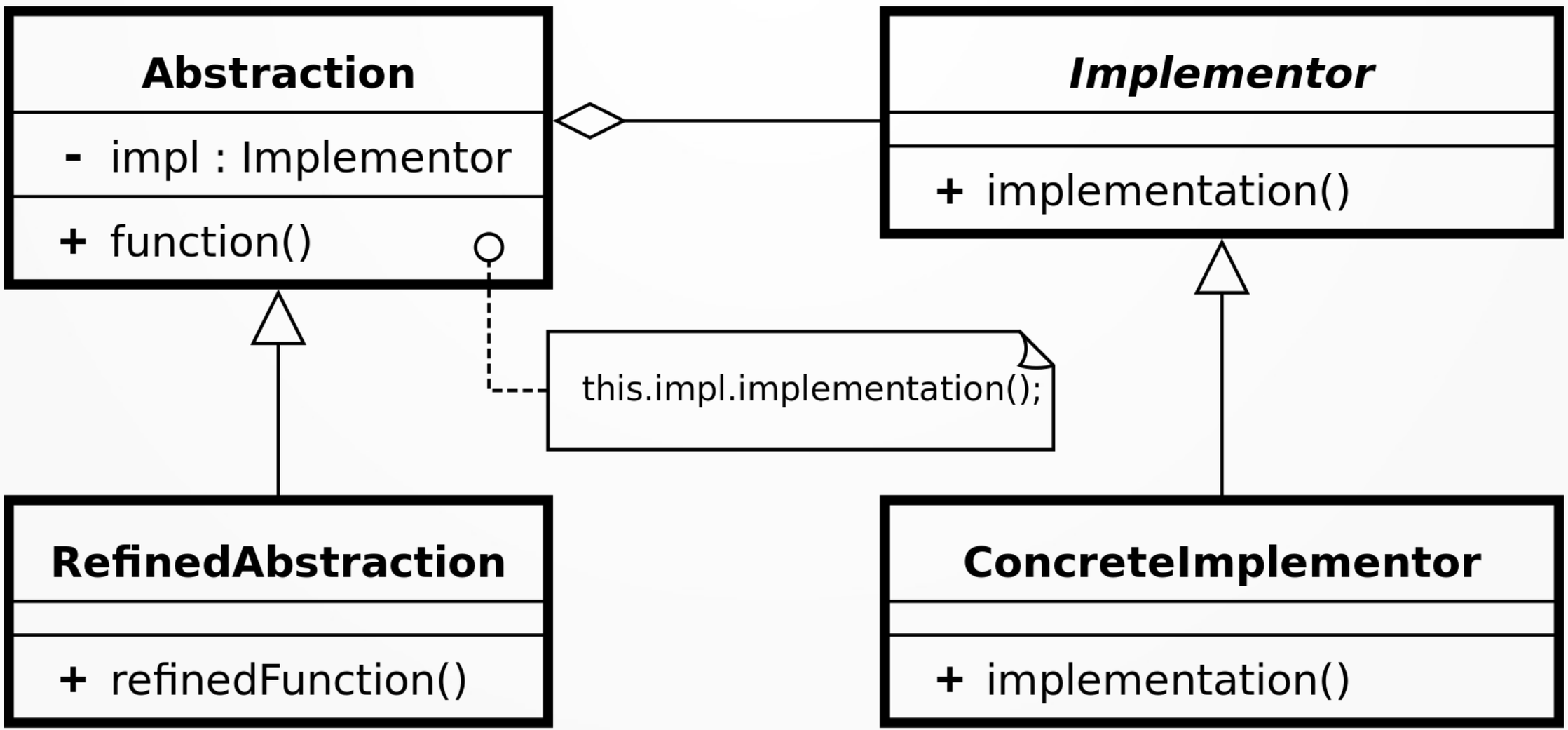
ШП Singleton (одиночка)
Пример на C++

```
class OnlyOne
{
public:
        static OnlyOne& Instance()
        {
                static OnlyOne theSingleInstance;
                return theSingleInstance;
        }
private:
        OnlyOne(){}
        OnlyOne(const OnlyOne& root) = delete;
        OnlyOne& operator=(const OnlyOne&) = delete;
};
```

ШП Bridge (мост)

- Задача:
  - например, нужно реализовать журналирование в различные места: файл, консоль, удалённый компьютер;
  - для каждого варианта логгера нужны варианты реализации: однопотоковая и многопотоковая
- Выделяется дополнительная иерархия, в которой реализовано представление изначальной иерархии
- Позволяет разделять абстракцию и реализацию так, чтобы они могли изменяться независимо

ШП Bridge (мост)
Диаграмма классов



ШП Bridge (мост)
Особенности

- Плюс:
  - упрощает модифицирование кода реализации.
- Минусы:
  - усложняет реализацию;
  - ухудшает производительность.
- ШП Bridge и Adapter имеют схожую структуру, однако, цели их использования различны. Если паттерн Adapter применяют для адаптации уже существующих классов в систему, то паттерн Bridge используется на стадии ее проектирования.

```cpp
#include <iostream>

using namespace std;

class Drawer {
public:
    virtual void drawCircle(int x, int y, int radius) = 0;
};

class SmallCircleDrawer : public Drawer {
public:
    const double radiusMultiplier = 0.25;

    void drawCircle(int x, int y, int radius) override
    {
        cout << "Small circle center " << x << ", " << y << " radius = " <<
                radius*radiusMultiplier << endl;
    }
};

class LargeCircleDrawer : public Drawer {
public:
    const double radiusMultiplier = 10;

    void drawCircle(int x, int y, int radius) override
    {
        cout << "Large circle center " << x << ", " << y << " radius = " <<
                radius*radiusMultiplier << endl;
    }
};

class Shape {
protected:
    Drawer* drawer;

public:
    Shape(Drawer* drw) {
        drawer = drw;
    }
    Shape() {}

    virtual void draw() = 0;
    virtual void enlargeRadius(int multiplier) = 0;
};
```

```cpp
class Circle : public Shape {
    int x, y, radius;
public:
    Circle(int _x, int _y, int _radius, Drawer* drw)
    {
        drawer = drw;
        setX(_x);
        setY(_y);
        setRadius(_radius);
    }
    void draw() override {
        drawer->drawCircle(x, y, radius);
    }
    void enlargeRadius(int multiplier) override {
        radius *= multiplier;
    }
    void setX(int _x) {
        x = _x;
    }
    void setY(int _y) {
        y = _y;
    }
    void setRadius(int _radius) {
        radius = _radius;
    }
};

int main(int argc, char *argv[])
{
    Shape* shapes[2] = { new Circle(5,10,10, new LargeCircleDrawer()),
                         new Circle(20,30,100, new SmallCircleDrawer())};
    for (int i =0 ; i < 2; i++)
    {
        shapes[i]->draw();
    }
    return 0;
}
// Output
Large circle center = 5,10 radius = 100
Small circle center = 20,30 radius = 25.0
```

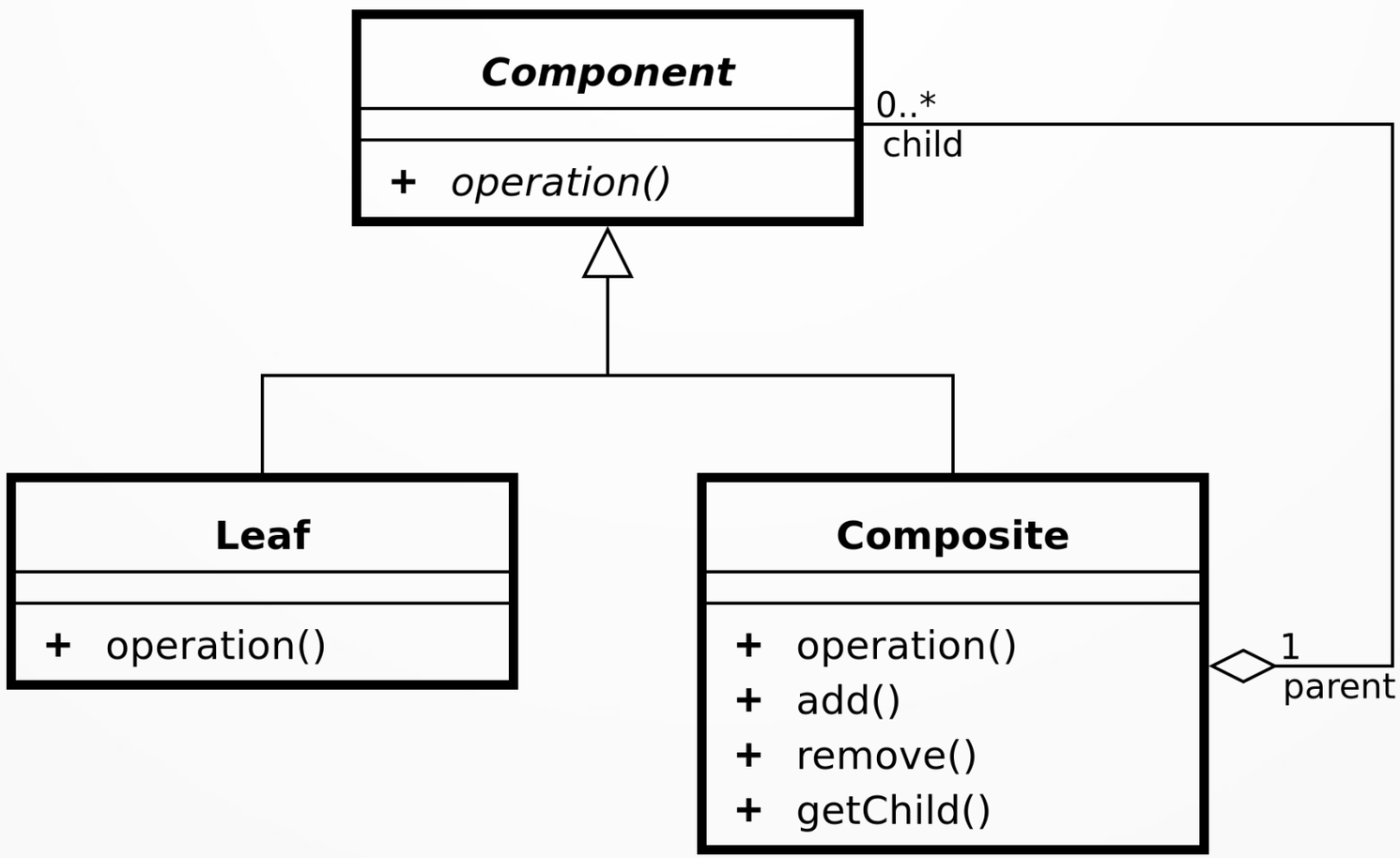
ШП Composite (компоновщик)

- Пример задачи:
  - необходимо реализовать текстовый редактор, в котором элементы образуют иерархию единообразных объектов
- Объединяет объекты в древовидную структуру для представления иерархии от частного к целому
- Компоновщик позволяет клиентам обращаться к отдельным объектам и к группам объектов одинаково.

ШП Composite (компоновщик)
Примеры использования

- Многие редакторы (MS Word, LibreOffice Writer и т. д.)
- HTML, DOM

ШП Composite (компоновщик)
Диаграмма классов



```cpp
#include <iostream>
#include <vector>
#include <memory>

#include <cassert>

using namespace std;

// Component
class Unit
{
public:
    virtual ~Unit()                         = default;
    virtual int getStrength() const         = 0;
    virtual void addUnit(shared_ptr<Unit> p)  { assert( false); }
};

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

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

class Horseman: public Unit
{
public:
    virtual int getStrength() const { return 3; }
};


// Composite
class CompositeUnit: public Unit
{
public:
    int getStrength() const
    {
        int total = 0;
        for(auto c : _composite)
            total += c->getStrength();
        return total;
    }
    void addUnit(shared_ptr<Unit> p) { _composite.push_back(p); }

private:
   vector<shared_ptr<Unit>> _composite;
};
```

```
// Вспомогательная функция для создания легиона
shared_ptr<Unit> createLegion()
{
  // Римский легион содержит:
  shared_ptr<Unit> legion = make_shared<CompositeUnit>();
  // 3000 тяжелых пехотинцев
  for (int i=0; i < 3000; ++i)
    legion->addUnit(make_shared<Infantryman>());
  // 1200 легких пехотинцев
  for (int i=0; i < 1200; ++i)
    legion->addUnit(make_shared<Archer>());
  // 300 всадников
  for (int i=0; i < 300; ++i)
    legion->addUnit(make_shared<Horseman>());

  return legion;
}

int main()
{
  // Римская армия состоит из 4-х легионов
  shared_ptr<Unit> army = make_shared<CompositeUnit>();

  for (int i=0; i < 4; ++i)
    army->addUnit(createLegion());

  cout << "Roman army damaging strength is "
       << army->getStrength()
       << endl;

  return 0;
}
```

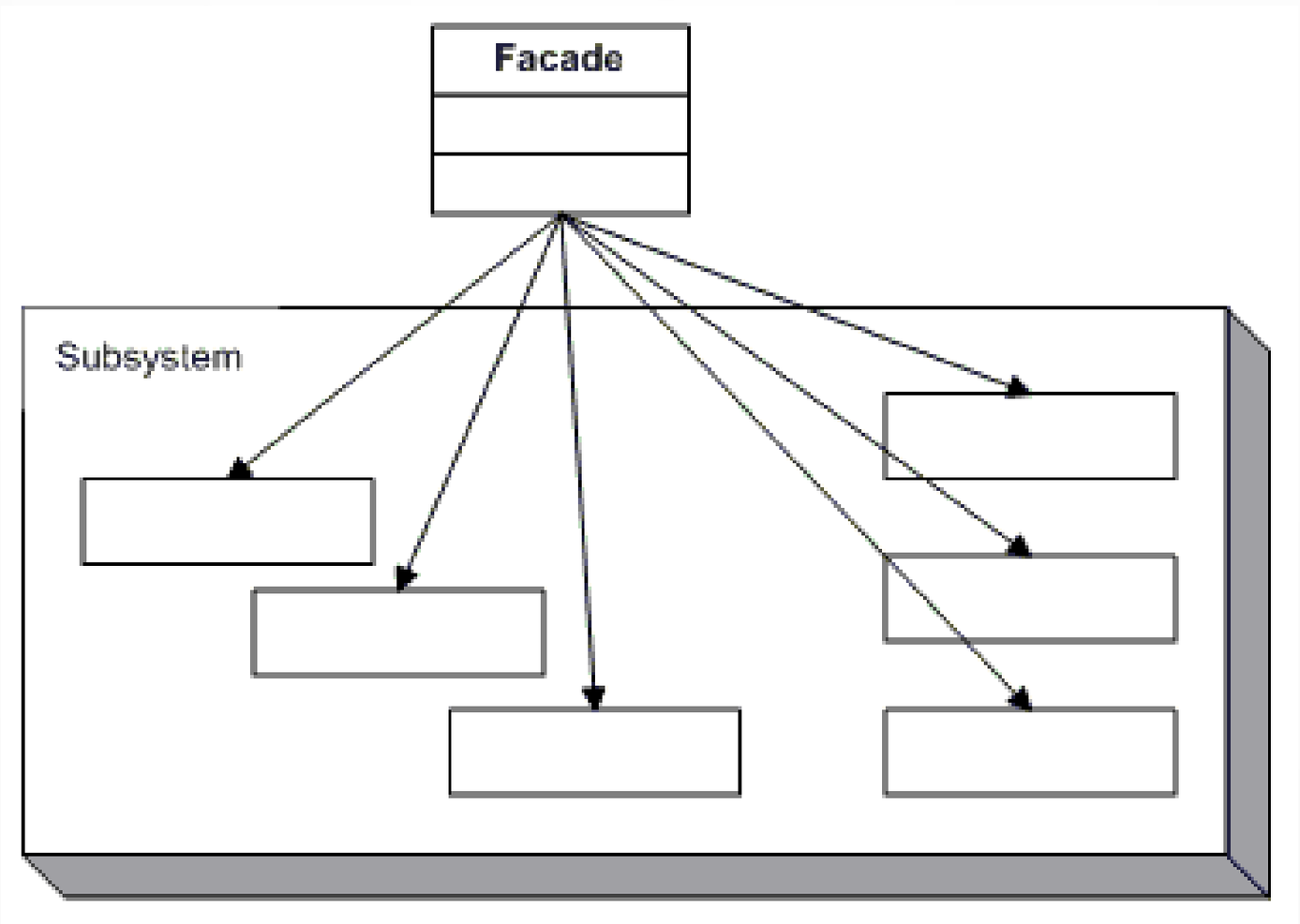
ШП Composite (компоновщик) Особенности

- Плюсы:
  - в программу легко добавлять новые примитивные или составные объекты, так как паттерн Composite использует общий базовый класс Component;
  - код имеет простую структуру — примитивные и составные объекты обрабатываются одинаковым образом;
  - позволяет легко обойти все узлы древовидной структуры.
- Минус:
  - неудобно осуществить запрет на добавление объектов определенных типов.

ШП Facade (фасад)

- Задача:
  - обеспечить унифицированный интерфейс с набором разрозненных реализаций или интерфейсов, например, с подсистемой,
  - нежелательно высокое связывание с этой подсистемой,
  - реализация подсистемы может измениться.
- Определить одну точку взаимодействия с подсистемой — фасадный объект, обеспечивающий общий интерфейс с подсистемой, и возложить на него обязанность по взаимодействию с её компонентами

ШП Facade (фасад)
Диаграмма компонентов



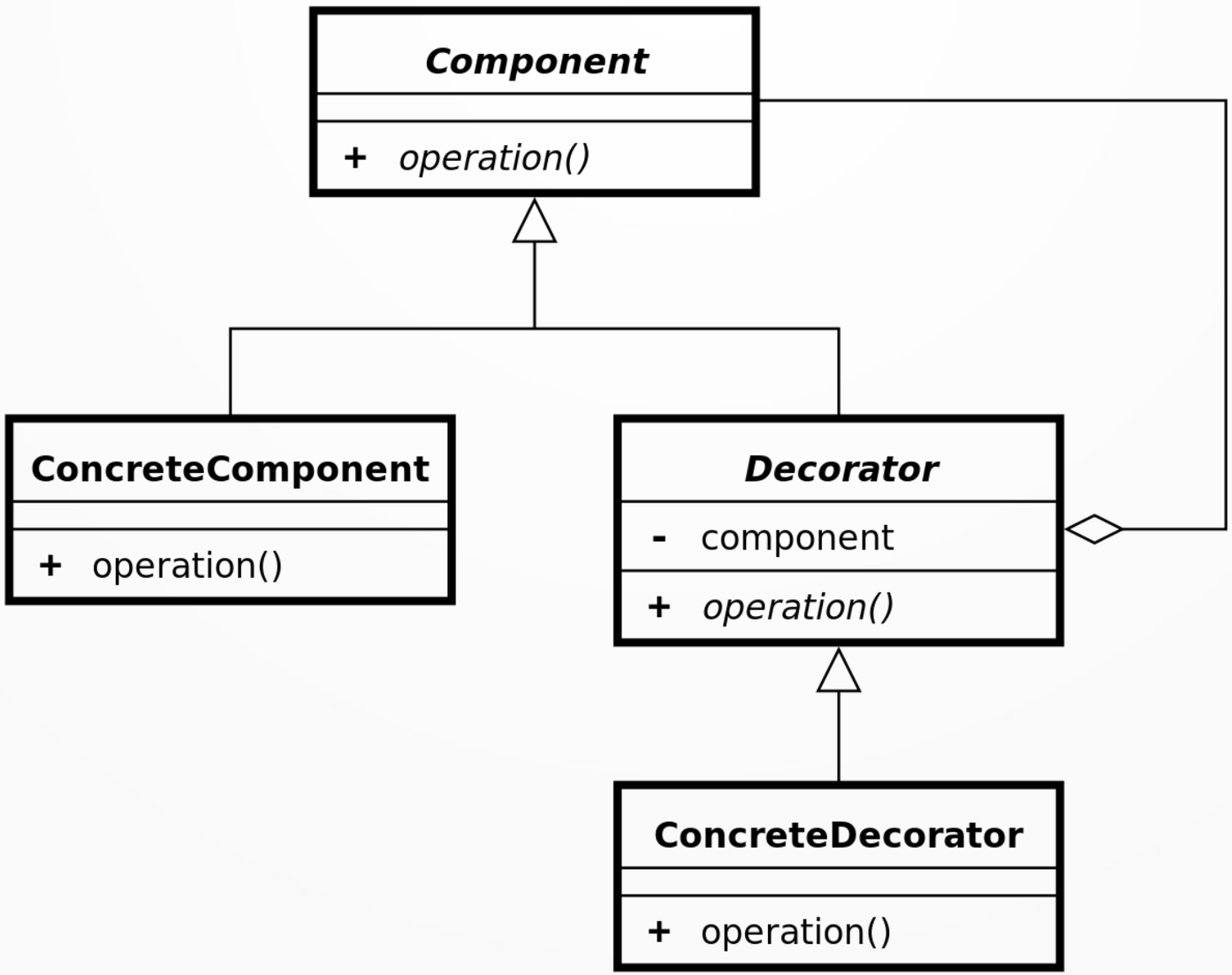
ШП Facade (фасад)
Особенности

- Фасад — это внешний объект, обеспечивающий единственную точку входа для служб подсистемы.
- Реализация других компонентов подсистемы закрыта и не видна внешним компонентам.
- Facade определяет новый интерфейс, в то время как Adapter использует уже имеющийся. Adapter делает работающими вместе два существующих интерфейса, не создавая новых.
- Abstract Factory может применяться как альтернатива Facade для сокрытия платформенно-зависимых классов
- Объекты "фасадов" часто являются Singleton, потому что требуется только один объект Facade.

ШП «Декоратор» (Decorator или Wrapper/Обёртка)

- **Задача**
  - Нужно расширить функциональность класса без использования наследования.
- **Decorator –** Класс, расширяющий функциональность другого класса без использования наследования.

ШП «Декоратор» (Decorator или Wrapper/Обёртка)



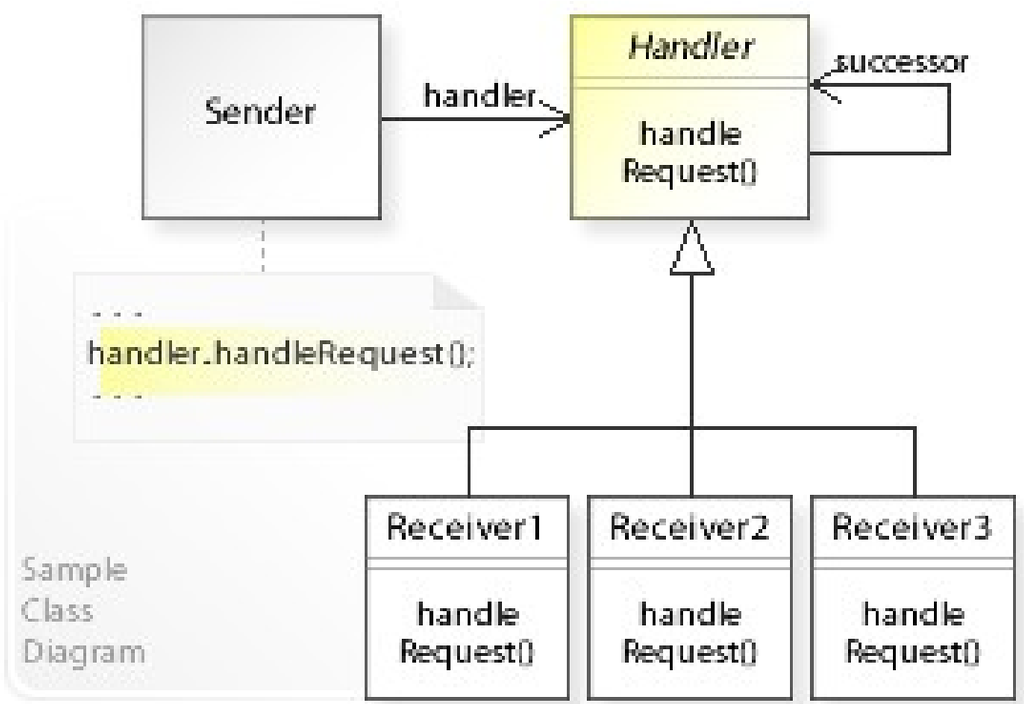
```cpp
#include <iostream>
#include <memory>

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(std::shared_ptr<IComponent> component): m_component(component) {}

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

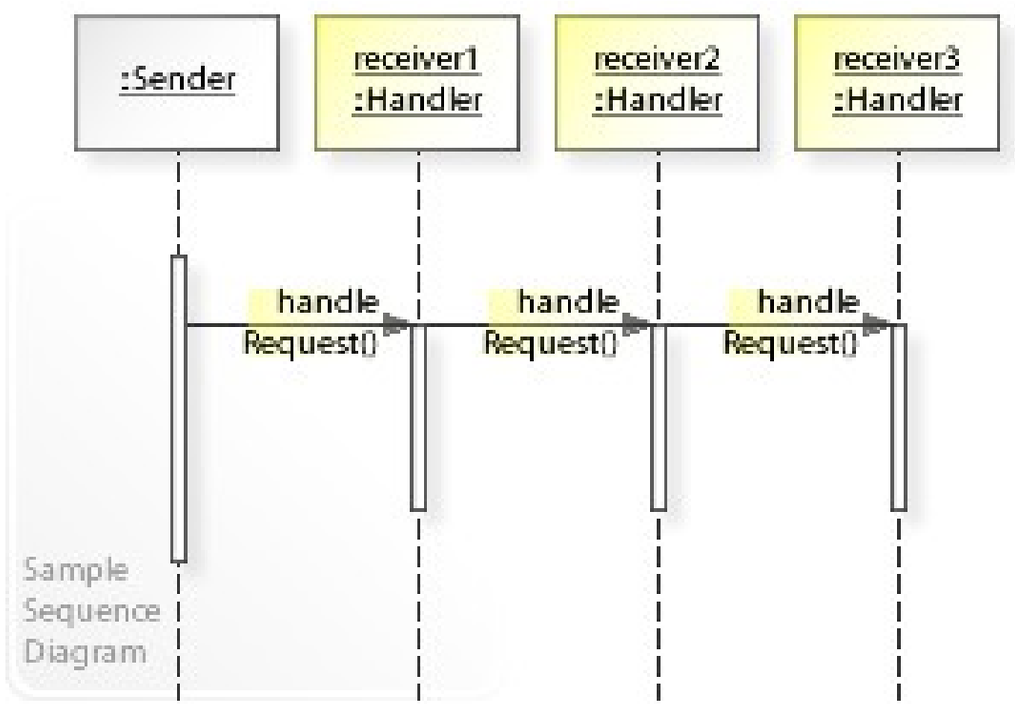
```cpp
class DecoratorTwo : public IComponent {
    std::shared_ptr<IComponent> m_component;

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

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

int main() {
    DecoratorTwo obj(std::make_shared<DecoratorOne>(std::make_shared<Component>()));
    obj.operation(); // prints "Hello, World!\n"

    return 0;
}
```

ШП «Цепочка ответственностей» (Chain of responsibility)

- **Задача**
  - в разрабатываемой системе имеется группа объектов, которые могут обрабатывать сообщения определенного типа;
  - все сообщения должны быть обработаны хотя бы одним объектом системы;
  - сообщения в системе обрабатываются по схеме «обработай сам либо перешли другому», то есть одни сообщения обрабатываются на том уровне, где они получены, а другие пересылаются объектам иного уровня.
- **Chain of responsibility (цепочка обязанностей)** – Предназначен для организации в системе уровней ответственности.

ШП «Цепочка ответственностей» (Chain of responsibility)





ШП «Команда» (Command, Action, Transaction)

- **Задача**
  - Создание структуры, в которой класс-отправитель и класс-получатель не зависят друг от друга напрямую.
  - Организация обратного вызова к классу, который включает в себя класс-отправитель.
- **Command** – Представляет действие. Объект команды заключает в себе само действие и его параметры.

ШП «Команда» (Command, Action, Transaction)

- Использование командных объектов упрощает построение общих компонентов, которые необходимо делегировать или выполнять вызовы методов в любое время без необходимости знать методы класса или параметров метода.
- Использование вызывающего объекта (invoker) позволяет вести учёт выполненных команд без необходимости знать клиенту об этой модели учёта (такой учёт может пригодиться, например, для реализации отмены и повтора команд).

ШП «Команда»
Примеры

- Кнопки пользовательского интерфейса и пункты меню
- Запись макросов
- Многоуровневая отмена операций (Undo)
- Сети
  - Можно отправить объекты команд по сети для выполнения на другой машине, например действие игрока в компьютерной игре.
- Индикаторы выполнения
- Пулы потоков
- и т.д.

ШП «Null Object»

- **Задача**
  - Объект требует взаимодействия с другими объектами. Null Object не устанавливает нового взаимодействия — он использует уже установленное взаимодействие.
  - Какие-то из взаимодействующих объектов должны бездействовать.
  - Требуется абстрагирование «общения» с объектами, имеющими NULL-значение.
- **Null Object** – Предотвращает нулевые указатели, предоставляя объект «по умолчанию».

```
class FuzeRdp : public Parser_interface, public inout::RdpBaseSugar
{
    std::string                     _path;              ///< Путь к файлу грамматики
    ast::FormationFlow_interface &  _flow; ///< Интерфейс для построения АСД

    InputStreamSupplier_interface & _input_stream_supplier;
    SyntaxDataCollector_interface & _syntax_data_collector; ///< Интерфейс сборщика информации о синтаксисе

    inout::uri_index_t              _current_file_index;
    bool                            _is_ready_for_parse;

public:
    FuzeRdp() = delete; ///< Пустой конструктор не поддерживается!

    /*!
     * \brief Конструктор класса начального разбора синтаксических правил языка
     * \param m             Интерфейсная ссылка на объект обеспечения вывода информации в вызываемую программу
     * \param path          Путь к грамматике
     * \param flow          Вспомогательный класс для построения АСД
     */
    FuzeRdp(inout::Reporter_abstract & m,
            const std::string & path,
            ast::FormationFlow_interface & flow);

    /*!
     * \brief Конструктор класса начального разбора синтаксических правил языка
     * \param m             Интерфейсная ссылка на объект обеспечения вывода информации в вызываемую программу
     * \param path          Путь к грамматике
     * \param flow          Вспомогательный класс для построения АСД
     * \param input_stream_supplier Поставщик входных потоков
     * \param syntax_data_collector Интерфейс сборщика информации о синтаксисе
     */
    FuzeRdp(inout::Reporter_abstract & m,
            const std::string & path,
            ast::FormationFlow_interface & flow,
            InputStreamSupplier_interface & input_stream_supplier,
            SyntaxDataCollector_interface & syntax_data_collector);
```

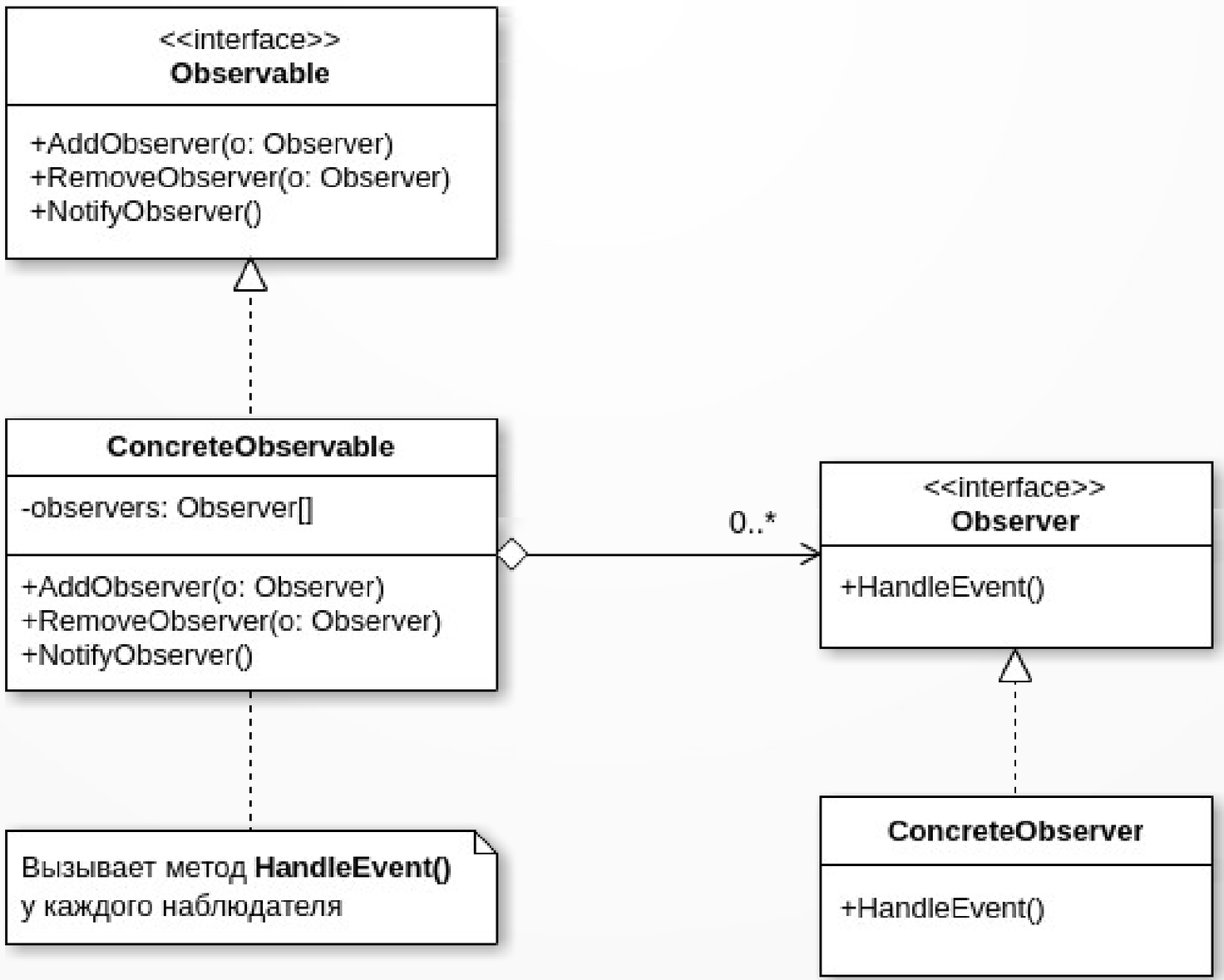
```
namespace {
    RegularInputStreamSupplier  regular_input_stream_supplier;
    SyntaxDataCollector_null    null_syntax_data_collector;
}

FuzeRdp::FuzeRdp(inout::Reporter_abstract & m,
                 const std::string & path,
                 ast::FormationFlow_interface & flow)
    : RdpBaseSugar(m, flow.tree().files())
    , _path(path)
    , _flow(flow)
    , _input_stream_supplier(regular_input_stream_supplier)
    , _syntax_data_collector(null_syntax_data_collector)
{
    initiateFileForParse();
}

FuzeRdp::FuzeRdp(inout::Reporter_abstract & m,
                 const std::string & path,
                 ast::FormationFlow_interface & flow,
                 InputStreamSupplier_interface & input_stream_supplier,
                 SyntaxDataCollector_interface & syntax_data_collector)
    : RdpBaseSugar(m, flow.tree().files())
    , _path(path)
    , _flow(flow)
    , _input_stream_supplier(input_stream_supplier)
    , _syntax_data_collector(syntax_data_collector)
{
    initiateFileForParse();
}
```

```cpp
#ifndef simodo_parser_SyntaxDataCollector_interface
#define simodo_parser_SyntaxDataCollector_interface

/*! \file SyntaxDataCollector_interface.h
    \brief Интерфейс сбора синтаксических данных в процессе синтаксического разбора
*/

#include "simodo/inout/token/Token.h"
#include "simodo/inout/reporter/Reporter_abstract.h"

namespace simodo::parser
{
    class SyntaxDataCollector_interface
    {
    public:
        virtual ~SyntaxDataCollector_interface() = default;

        virtual void collectToken(const inout::Token & token) = 0;
    };

    class SyntaxDataCollector_null : public SyntaxDataCollector_interface
    {
    public:
        virtual void collectToken(const inout::Token & ) override {}
    };

}

#endif // simodo_parser_SyntaxDataCollector_interface
```

ШП «Наблюдатель» (Observer)

- **Задача**
  - существует как минимум один объект, рассылающий сообщения;
  - имеется не менее одного получателя сообщений, причём их количество и состав могут изменяться во время работы приложения;
  - позволяет избежать сильного зацепления взаимодействующих классов.
- **Observer (наблюдатель)** – Определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом событии.

ШП «Наблюдатель» (Observer)



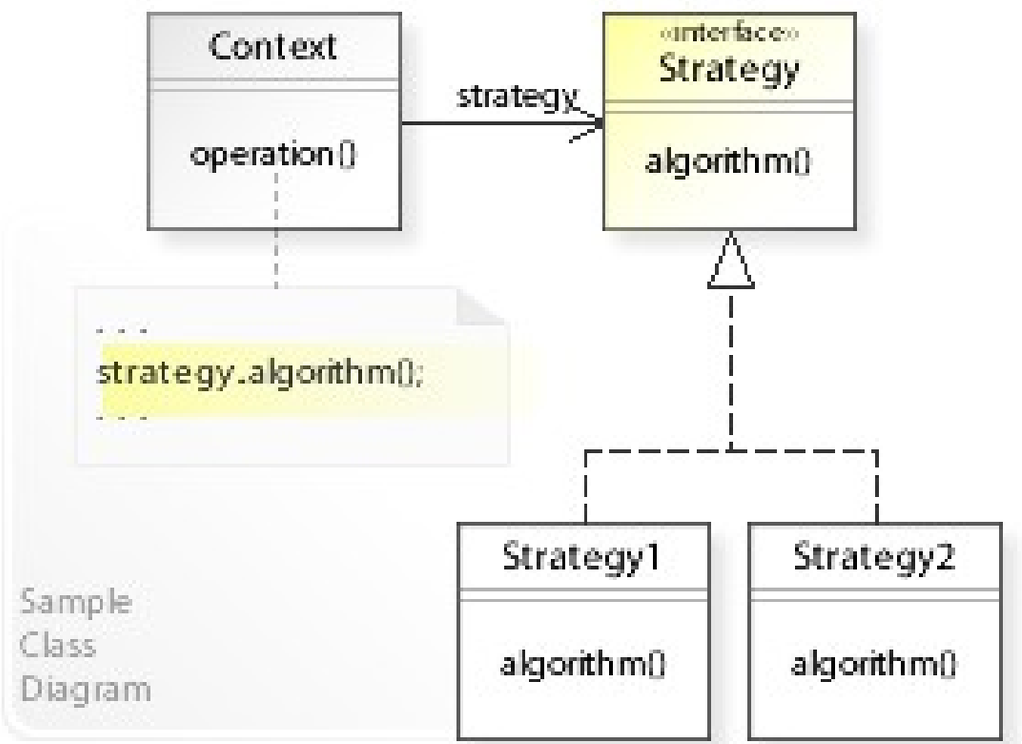
```
class SupervisedString;
class IObserver
{
public:
    virtual void handleEvent(const SupervisedString&) = 0;
};

class SupervisedString // Observable class
{
    string _str;
    list<IObserver*> _observers;

    void _Notify()
    {
        for(auto& observer: _observers)
        {
            observer->handleEvent(*this);
        }
    }

public:
    void add(IObserver& ref)
    {
        _observers.push_back(&ref);
    }

    void remove(IObserver& ref)
    {
        _observers.remove(&ref);
    }

    const string& get() const
    {
        return _str;
    }

    void reset(string str)
    {
        _str = str;
        _Notify();
    }
};
```

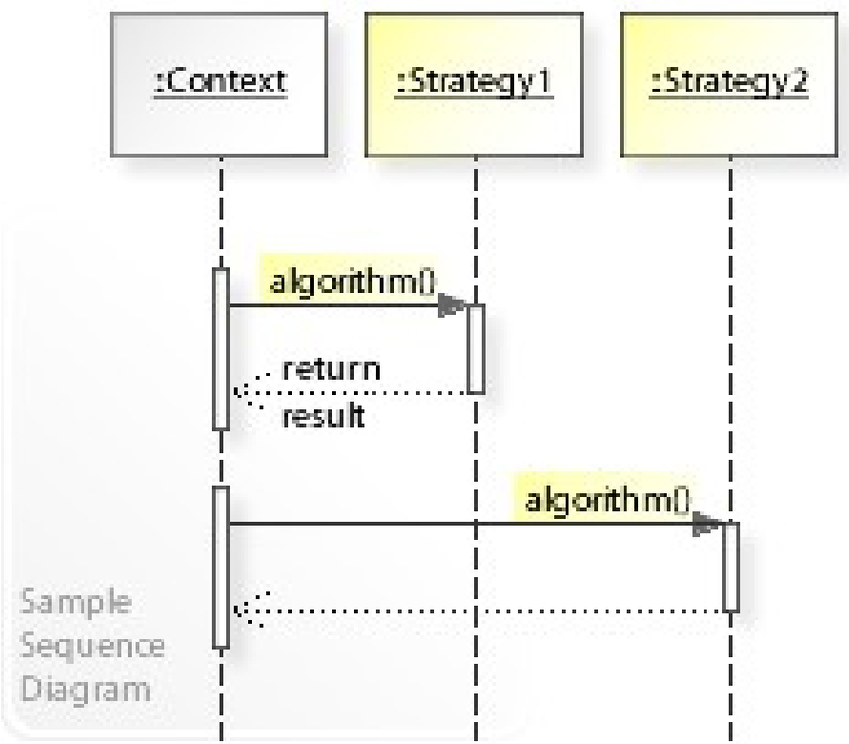
```
class Reflector: public IObserver // Prints the observed string into cout
{
public:
    virtual void handleEvent(const SupervisedString& ref)
    {
        cout << ref.get() << endl;
    }
};

class Counter: public IObserver // Prints the length of observed string into cout
{
public:
  virtual void handleEvent(const SupervisedString& ref)
  {
      cout << "length = " << ref.get().length() << endl;
  }
};

int main()
{
    SupervisedString str;
    Reflector refl;
    Counter cnt;

    str.add(refl);
    str.reset("Hello, World!");
    cout << endl;

    str.remove(refl);
    str.add(cnt);
    str.reset("World, Hello!");
    cout << endl;

    return 0;
}
```

ШП «Стратегия» (Strategy)

- **Задача**
  - По типу клиента (или по типу обрабатываемых данных) выбрать подходящий алгоритм, который следует применить.
- **Strategy** – Предназначен для определения семейства алгоритмов, инкапсуляции каждого из них и обеспечения их взаимозаменяемости.

ШП «Стратегия» (Strategy)





```cpp
#include <iostream>

class Strategy
{
public:
    virtual ~Strategy() {}
    virtual void use() = 0;
};

class Strategy_1: public Strategy
{
public:
    void use(){
        std::cout << "Strategy_1" << std::endl;
    }
};

class Strategy_2: public Strategy
{
public:
    void use(){
        std::cout << "Strategy_2" << std::endl;
    }
};

class Strategy_3: public Strategy
{
public:
    void use(){
        std::cout << "Strategy_3" << std::endl;
    }
};
```

```cpp
class Context
{
protected:
    Strategy* operation;

public:
    virtual ~Context() {}
    virtual void useStrategy() = 0;
    virtual void setStrategy(Strategy* v) = 0;
};

class Client: public Context
{
public:
    void useStrategy()
    {
        operation->use();
    }

    void setStrategy(Strategy* o)
    {
        operation = o;
    }
};

int main(int /*argc*/, char* /*argv*/[])
{
    Client customClient;
    Strategy_1 str1;
    Strategy_2 str2;
    Strategy_3 str3;

    customClient.setStrategy(&str1);
    customClient.useStrategy();
    customClient.setStrategy(&str2);
    customClient.useStrategy();
    customClient.setStrategy(&str3);
    customClient.useStrategy();

    return 0;
}
```

Вопросы?