

# ПРИНЦИПЫ ООП

---

Язык UML, принципы объектно-ориентированного проектирования и  
паттерны проектирования

К. Владимиров, Syntacore, 2022  
mail-to: [konstantin.vladimirov@gmail.com](mailto:konstantin.vladimirov@gmail.com)

➤ Проектирование и UML

❑ Принципы SOLID

❑ Правила хорошего кода

❑ Паттерны проектирования

# Контексты и интерфейсы

## Интерфейс (C-style): matrix.h

```
struct M;  
M* create_diag(size_t);  
M* prod(const M*, const M*);  
double det(const M*);  
void destroy(M*);  
// .....
```

## Контекст (C-style): matrix.c

```
struct M {  
    double *contents;  
    size_t x, y;  
};  
#define Msz sizeof(M);  
M* create_diag(size_t w) {  
    M* ret = malloc(Msz);  
    // .....
```

# Контексты и интерфейсы

## Интерфейс (C++ style): imatrix.h

```
struct IM {  
    virtual IM& clone(const IM&);  
    virtual ~IM() = 0;  
    // .....
```

## Контекст (C++ style): matrix.hpp

```
template <typename T>  
class M : public IM {  
    T *contents;  
    size_t x, y;  
  
public:  
    M(M& rhs);  
    M& clone(const IM&) override;  
    // все реализации в том же файле
```

# Контексты и инварианты

## Контекст (C++ style): matrix.hpp

```
template <typename T>
class M : public IM {
    T *contents;
    size_t x, y;
public:
    M(M& rhs);
    M& clone(const IM&) override;
    // ....
```

## Инварианты

- Указатель contents валиден если  $x \neq 0$
- Если  $x \neq 0$  то всегда  $y \neq 0$
- Для contents аллоцирована память размером  $x * y * \text{sizeof}(T)$
- После клонирования матрица равна исходной
- Ещё?

# Базовые понятия

- Контекст инкапсулирует данные и охраняет инварианты.
- Контекст реализует интерфейс (для типов в C++ через наследование интерфейса).
- Производный контекст расширяет базовый (для типов в C++ через наследование реализации).
- Если контексты это типы, производный контекст связан с базовым дополнительными отношениями (частное/общее, быть частью и подобными).
- Если несколько типов реализуют общий интерфейс, вызовы их методов через этот интерфейс полиморфны.

# Обсуждение: проектирование

- Проектирование сложной системы классов это человеческая деятельность
- Что является артефактом этой деятельности?
- Как можно было бы хотя бы частично формализовать этот процесс?

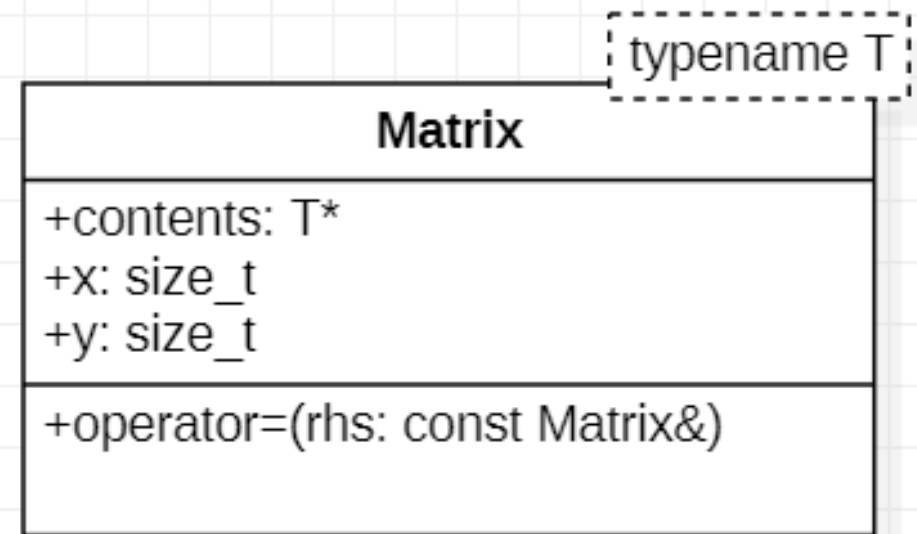
# Обсуждение: язык моделирования

- Проектирование это моделирование отношений между типами
- В каких отношениях могут быть друг с другом классы в C++?
- Примеры отношений: "A наследует от B" или "C является полем в D"
- Назовите все какие сможете вообразить



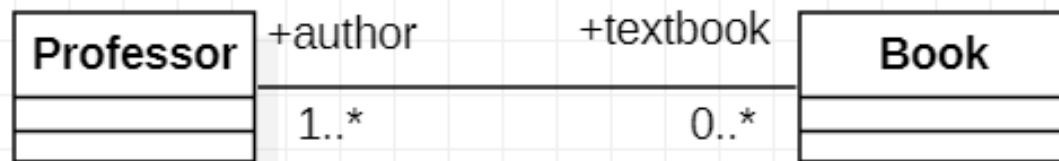
# Отношения между классами и UML

- UML это специальный язык, который моделирует классы и отношения между классами (отношения будут далее)
- Класс в UML определяется через своё имя, поля и методы
- По традиции имя идёт в первом квадрате, поля во втором а методы в третьем
- Формат полей "поле : тип" (несколько контринтуитивно для C++)
- UML поддерживает также тонны других атрибутов, например шаблонные параметры



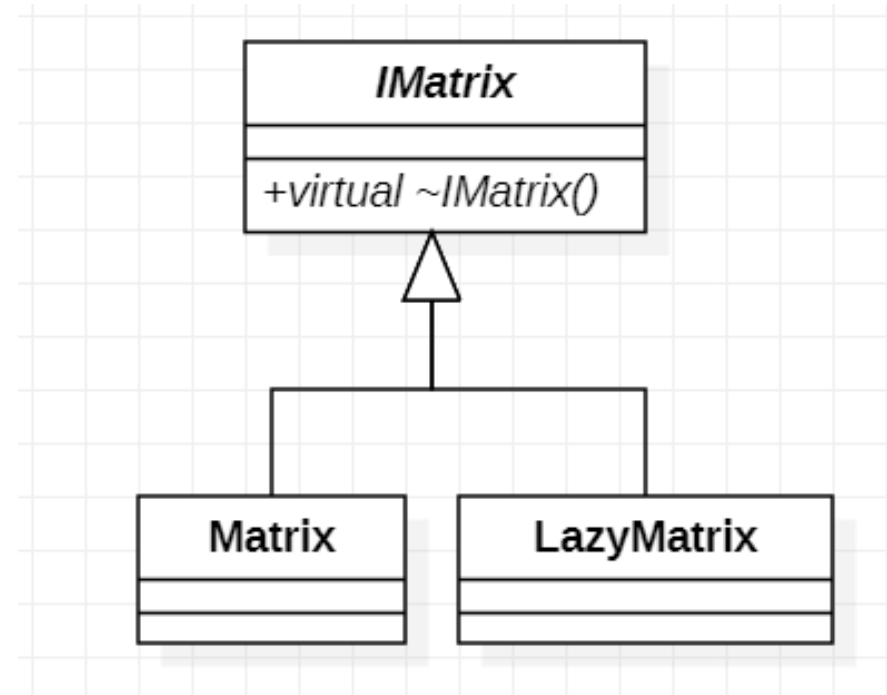
# Отношения между классами и UML

- Ассоциация: сущности каким-то образом связаны друг с другом
- Например появляются вместе (внутри одной функции)



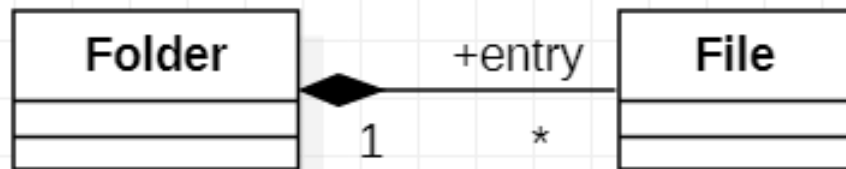
- Здесь также видно, что у каждой связи можно указать роли и множественность.

- Генерализация: отношение частное/общее (для C++ это открытое наследование)



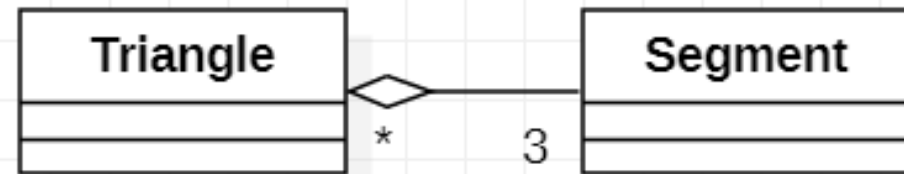
# Отношения между классами и UML

- Композиция означает, что сущность В является частью сущности А



- Здесь файл принадлежит только одной папке и связан с ней временем жизни

- Аггрегация: сущность А владеет сущностью В, но кроме А у В может быть много владельцев



- Здесь треугольник состоит из отрезков, но каждый из отрезков может участвовать во многих треугольниках

# Обсуждение

- UML это средство описания, которым можно описать любую систему, в том числе сколь угодно плохую.
- Software имеет английский корень soft, означающий нечто, что легко изменять.
- Но часто вместо куска пластилина у нас под руками оказывается странная засохшая субстанция с обломками гвоздей и лезвий внутри.
- Первый шаг к хорошему коду это **легко изменяемый** код.

- ❑ Проектирование и UML

- Принципы SOLID

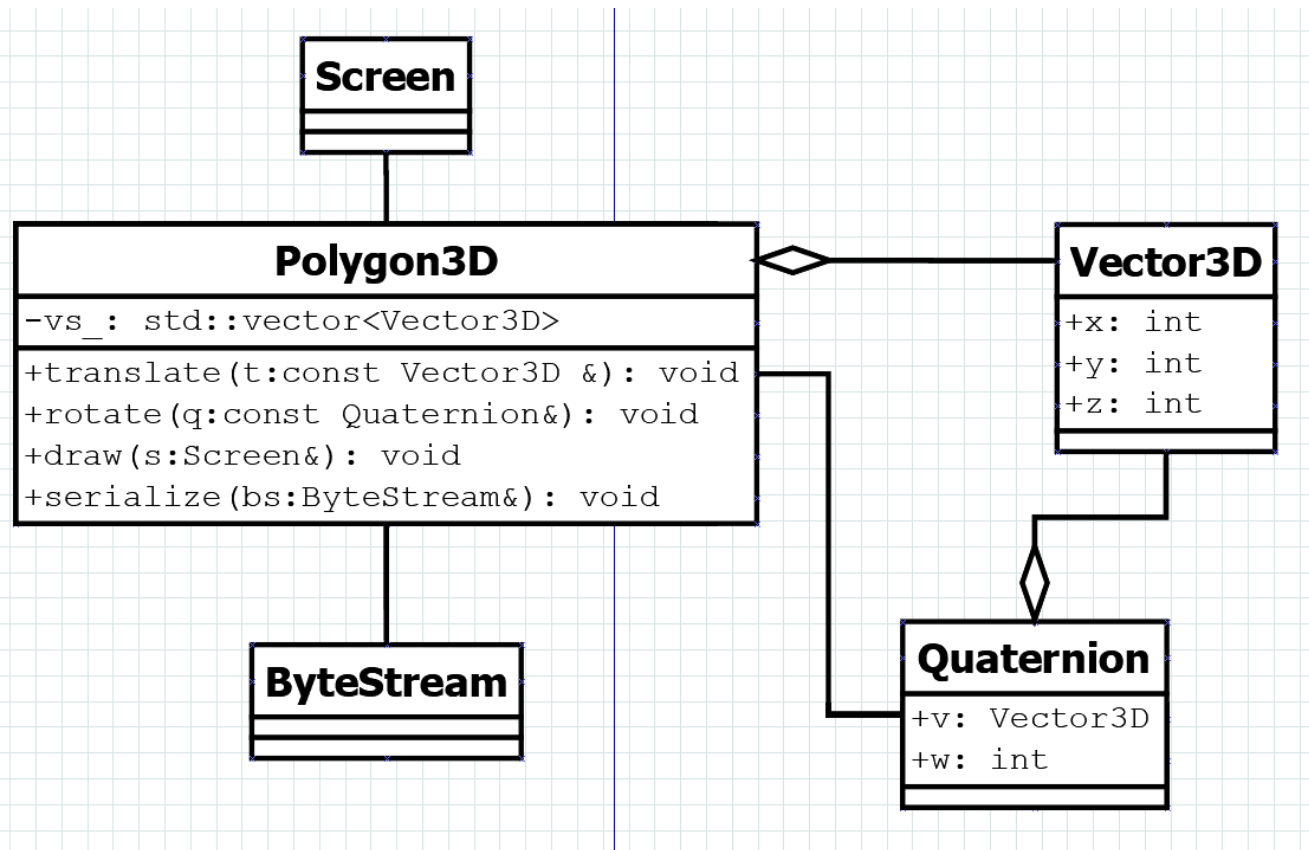
- ❑ Правила хорошего кода

- ❑ Паттерны проектирования

# Принципы SOLID

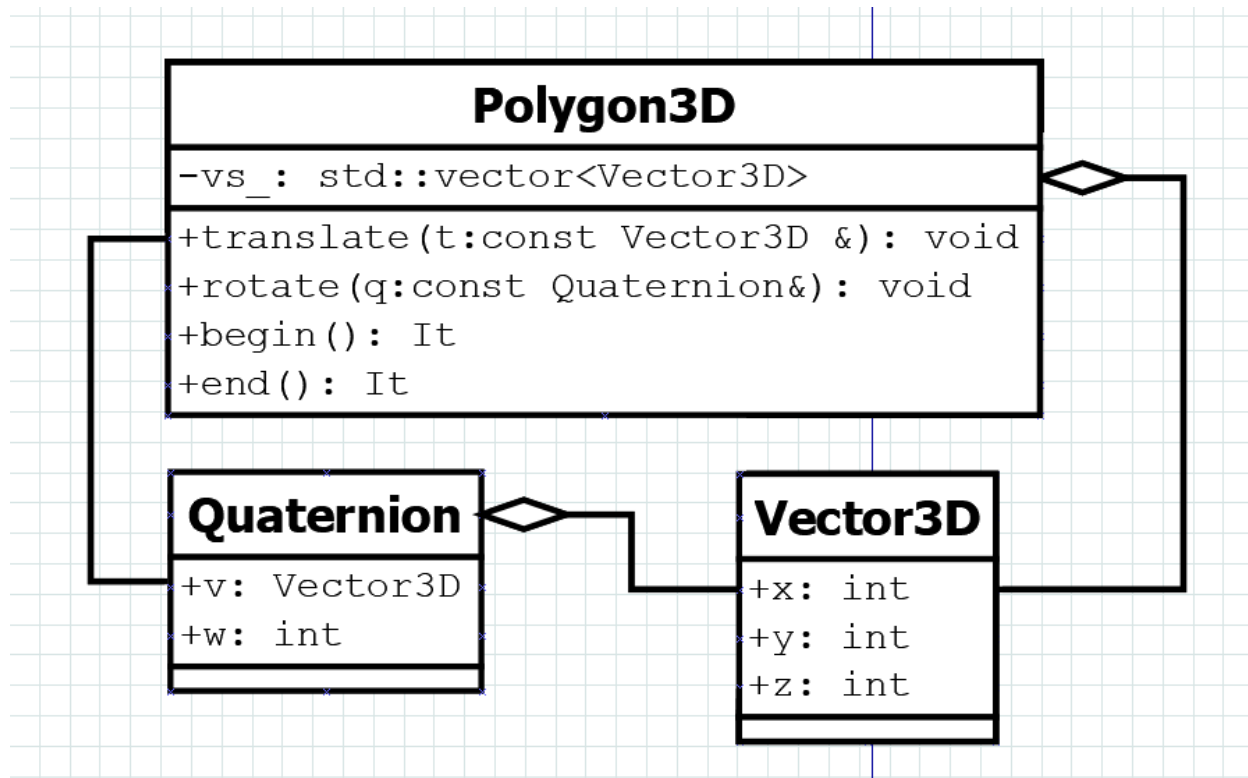
- **S**RP – single responsibility principle
  - каждый контекст должен иметь одну ответственность
- **O**CP – open-close principle
  - каждый контекст должен быть закрыт для изменения и открыт для расширения
- **L**SP – Liskov substitution principle
  - частный класс должен иметь возможность свободно заменять общий
- **I**SP – interface segregation principle
  - Тип не должен зависеть от тех интерфейсов, которые он не использует
- **D**IP – dependency inversion principle
  - Высокоуровневые классы не должны зависеть от низкоуровневых

# Пример плохого проектирования (SRP)



- В каком случае мы тут должны будем изменять полигон?
- Что в этом плохого?
- Есть ли нечто плохое в зависимости от вектора и от кватернионов?
- "A class should have only one reason to change" (Robert C. Martin)

# Принцип единственной ответственности



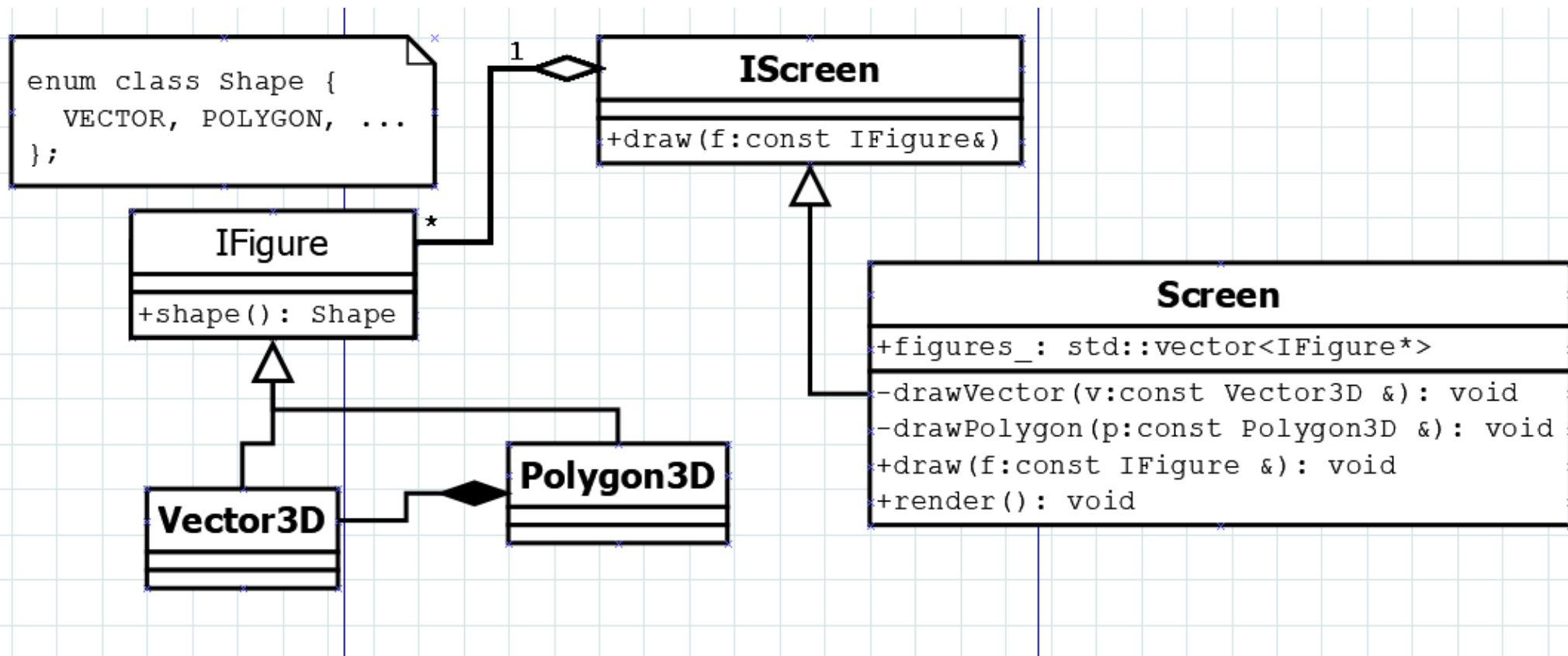
- Теперь единственная обязанность это геометрия
- Для вывода есть итераторы
- В итоге внешние функции могут обращаться к элементам но не к состоянию полигона
- "We want to design components that are self-contained: independent and with single well-defined purpose" (Andrew Hunt, David Thomas)



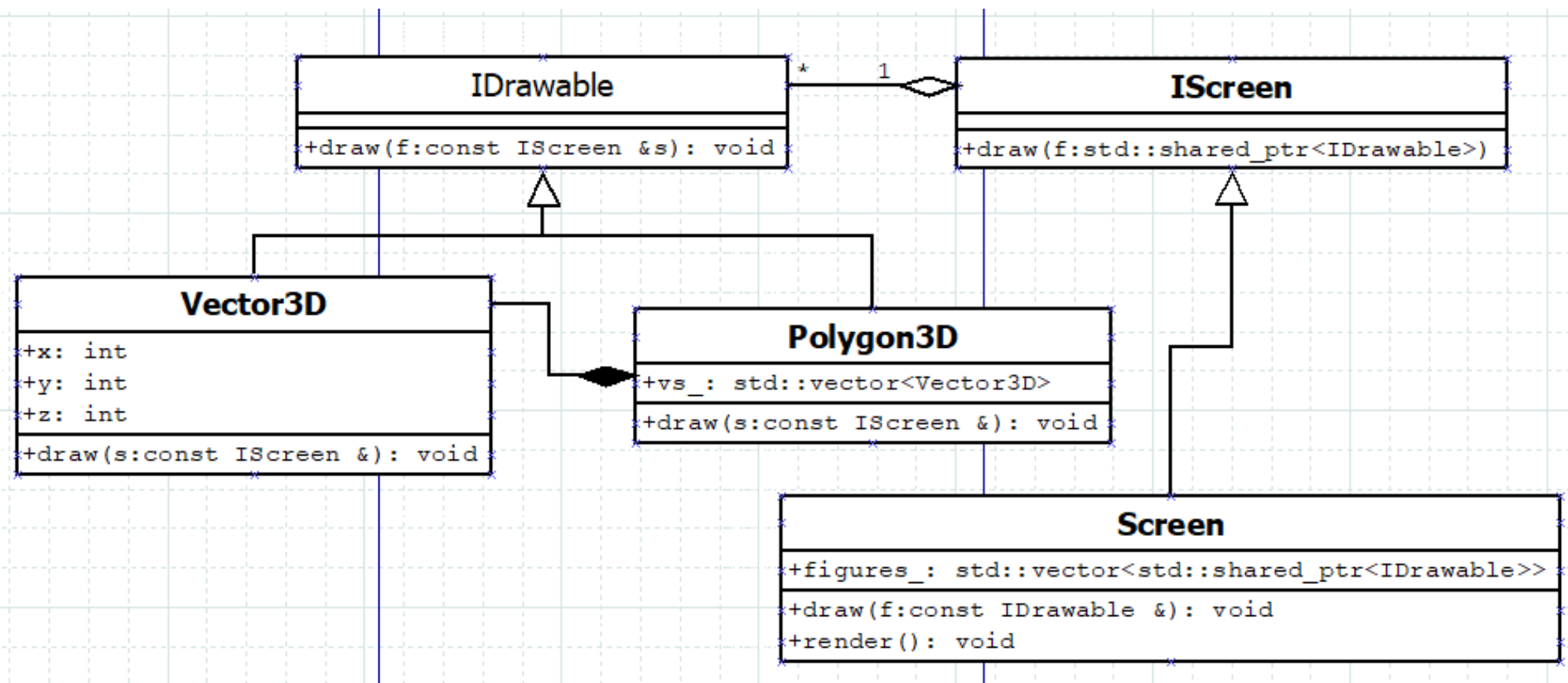
# Гайдлайн: СВЯЗНОСТЬ

- Ваши сущности должны быть внутренне связаны (cohesive) и внешне разделены.
- Разделяйте всё, что может быть разделено без создания жёстких внешних связей. Пример: отделение алгоритмов от контейнеров.
- "Cohesion is a measure of the strength of association of the elements inside a module. A highly cohesive module is a collection of statements and data items that should be treated as a whole because they are so closely related."  
(Tom DeMarco)

# Пример плохого проектирования (ОСР)



# Принцип открытости и закрытости



# Обсуждение

- Такое чувство, что ОСР в таком наивном виде противоречит SRP.
- Мы добавили виртуальную функцию draw в полигон, но мы несколькими слайдами раньше договорились этого **не** делать.
- "Inheritance is the base class of Evil" (Sean Parent)
- Посмотрите на код справа.
- Чего мы хотели бы?

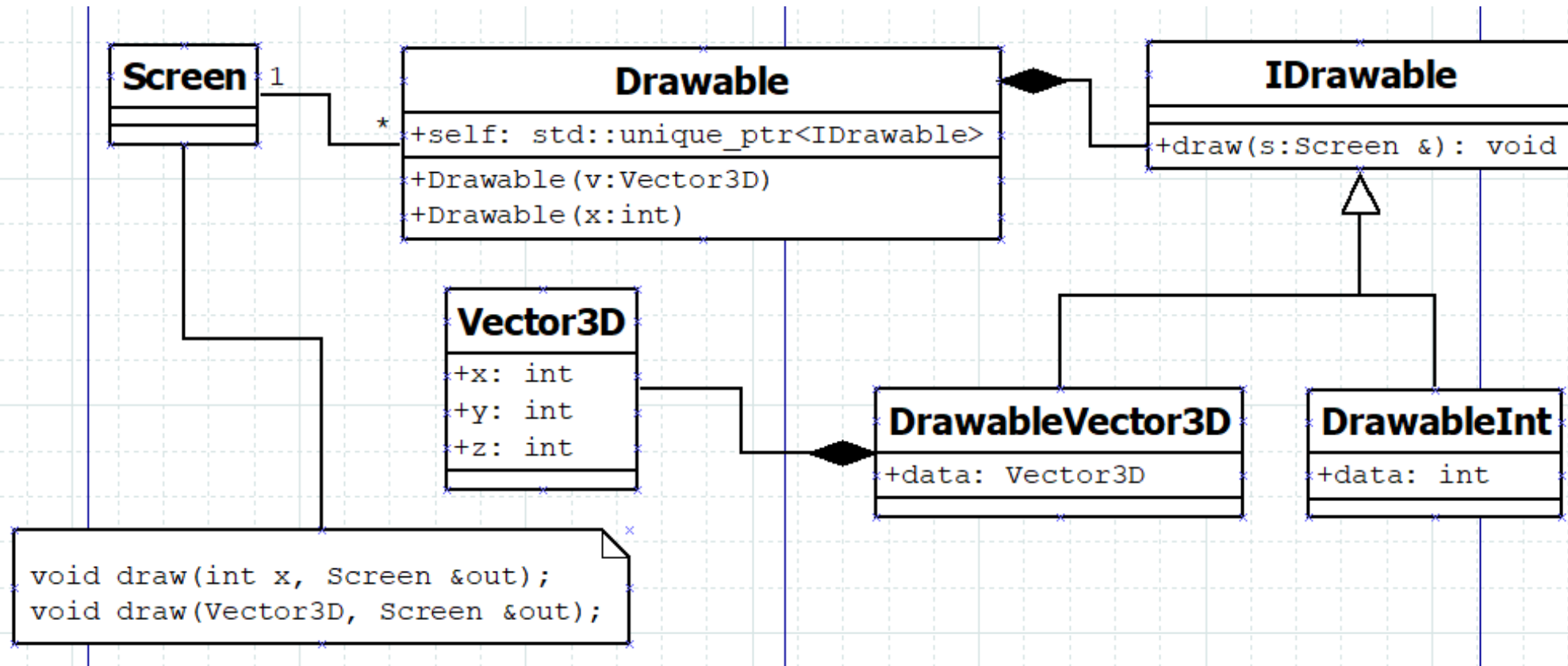
```
using document_t = std::vector<int>;  
  
// документ хранит объекты  
// семантика значения  
// no incidental data structures  
document.push_back(1);  
document.push_back(2);  
document.push_back(3);  
  
draw(document, std::cout);
```

# Обсуждение

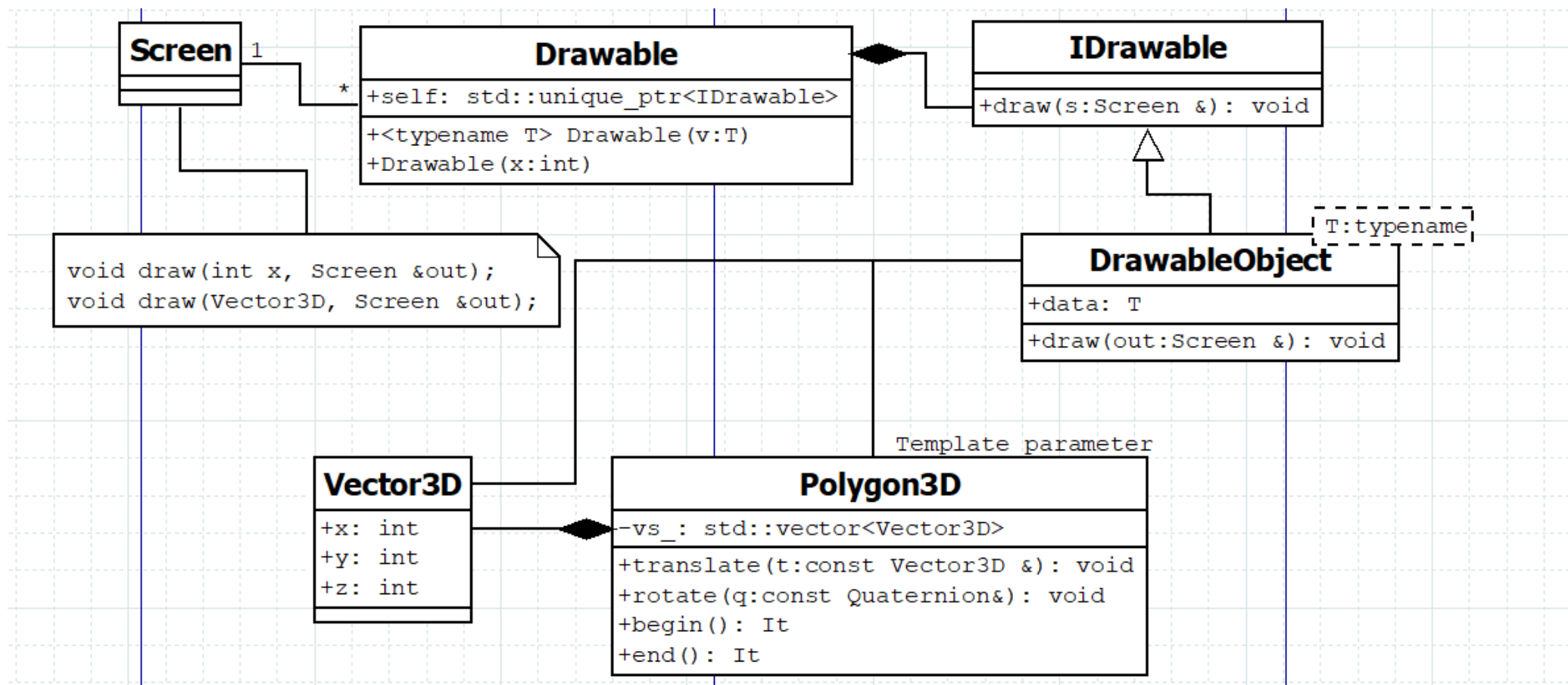
- Такое чувство, что ОСР в таком наивном виде противоречит SRP.
- Мы добавили виртуальную функцию draw в полигон, но мы несколькими слайдами раньше договорились этого **не** делать.
- "Inheritance is the base class of Evil" (Sean Parent)
- Посмотрите на код справа.
- Чего мы хотели бы?

```
using document_t = std::vector<??>;  
  
// документ хранит объекты  
// семантика значения  
// no incidental data structures  
document.push_back(circle);  
document.push_back(polygon);  
document.push_back(vector);  
  
draw(document, std::cout);  
  
// мы хотели бы хранить и полиморфно  
// отображать разнородные объекты
```

# Модель и концепция



# Parent reversal: вводим шаблоны



# Обсуждение

- Техники наподобие Parent Reversal позволяют помирить OCP и SRP
- Теперь мы расширяем добавляя свободные функции, полиморфные, как множество перегрузки.
- Динамический полиморфизм при этом остаётся деталью реализации.
- Шаблонный полиморфизм используется чтобы позволить обобщённое программирование



# Пример плохого проектирования (LSP)

- Все ли видят в чём тут основная проблема?

```
bool intersect(Polygon2D& l, Polygon2D& r); // 2D intersection
```

```
class Polygon2D {  
    std::vector<double> xcoord, ycoord;  
    // .... everything else ....  
};
```

```
class Polygon3D : public Polygon2D {  
    std::vector<double> zcoord;  
    // .... everything else ....  
}
```

# Принцип подстановки Лисков

- Более общие классы должны быть более общими и по составу и по поведению.

```
class Polygon3D : public Polygon2D;
```

- Это читается как: **трёхмерный полигон может быть использован во всех контекстах, где нам нужен двумерный полигон**. Если это некорректно, наследовать нельзя.
- Предусловия алгоритмов не могут быть усилены производным классом.
- Постусловия алгоритмов не могут быть ослаблены производным классом.
- Важной концепцией для LSP является ковариантность.

# Ковариантность

- Мы говорим, что изменение типа **ковариантно к генерализации**, если выполняется условие:

если A обобщает B, то A' обобщает B'

- Собственно указатели ковариантны к генерализации если трактовать  $A' = A^*$

```
class Rectangle : public Shape { /* ... */ };
```

```
void draw(Shape* shapes, size_t size);
```

```
Rectangle rects[5];
```

```
draw(rects, 5); // ok, Rectangle* is Shape*
```

# Обсуждение

- Динамический полиморфизм коварен.

```
void draw(Shape* shapes, size_t size);
```

```
Rectangle rects[5];
```

```
draw(rects, 5); // грамматически ok, Rectangle* is Shape*
```

- Как вы думаете нет ли здесь скрытых проблем?

# Инвариантность

- Мы говорим, что изменение типа ковариантно к генерализации, если выполняется условие:

если A обобщает B, то A' обобщает B'

- При этом шаблоны вообще-то **инвариантны к генерализации**

```
class Rectangle : public Shape { /* ... */ };
```

```
void draw(std::vector<Shape> shapes);
```

```
std::vector<Rectangle> rects(5);
```

```
draw(rects); // fail, vector<Rectangle> is not vector<Shape>
```

# Обсуждение

- Можно поставить обратный вопрос: а почему, собственно, указатели не инвариантны?

```
template <typename T> using Pointer = T*; // казалось бы
```

```
void draw(Pointer<Shape> shapes, size_t size);
```

```
Pointer<Rectangle> rects = new Rectangle[5];
```

```
draw(rects, 5); // ok, но чем Pointer<Rectangle>  
                // лучше чем std::vector<Rectangle>?
```

- Подсказка: ковариантны только одинарные указатели
- Таким образом, ковариантность указателей и ссылок к обобщению это приятное исключение для LSP, а не правило.

# Контравариантность

- Мы говорим, что изменение типа **контравариантно к генерализации**, если выполняется условие:

если A обобщает B, то B ' обобщает A '

- Контравариантны возвращаемые значения методов.

# Обсуждение

- Именно ковариантность указателей и ссылок и их не подверженность срезке делают их отличными кандидатами в C++
- Но их использование приводит к неявным (incidental) структурам данных и убивает value-семантику.



# Пример плохого проектирования (ISP)

```
struct IWorker {  
    virtual void work() = 0;  
    virtual void eat() = 0;  
    // .....  
};  
  
class Robot : public IWorker {  
    void work() override;  
    void eat() override {  
        // do nothing  
    }  
};
```

```
class Manager {  
    IWorker *subdue;  
  
public:  
    void manage () {  
        subdue->work();  
    }  
};
```

- Здесь менеджер зависит от интерфейса eat. В итоге его должны реализовать роботы

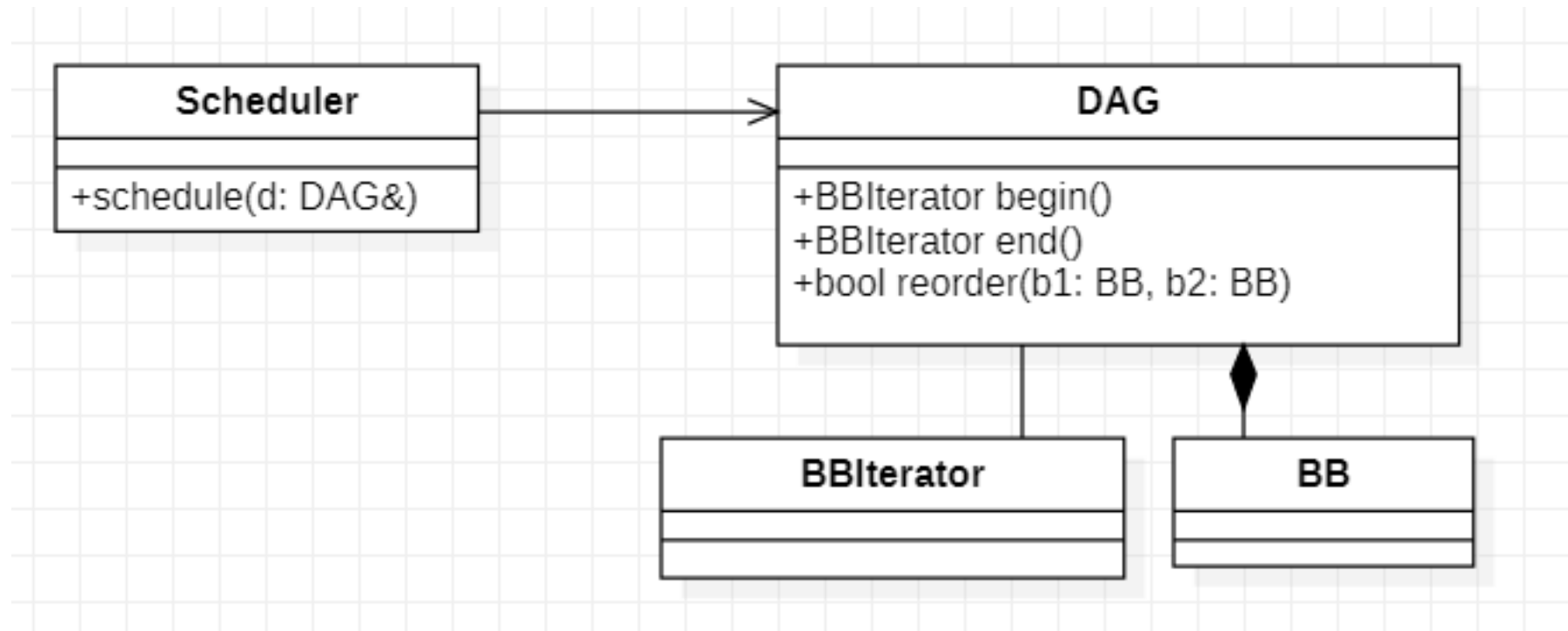
# Принцип разделения интерфейса

- Более общие классы должны быть более общими

```
struct IWorkable {  
    virtual void work() = 0;  
    // .....  
};  
  
class Robot: public IWorkable {  
    void work() override;  
};
```

- Такое чувство, что это SRP restated

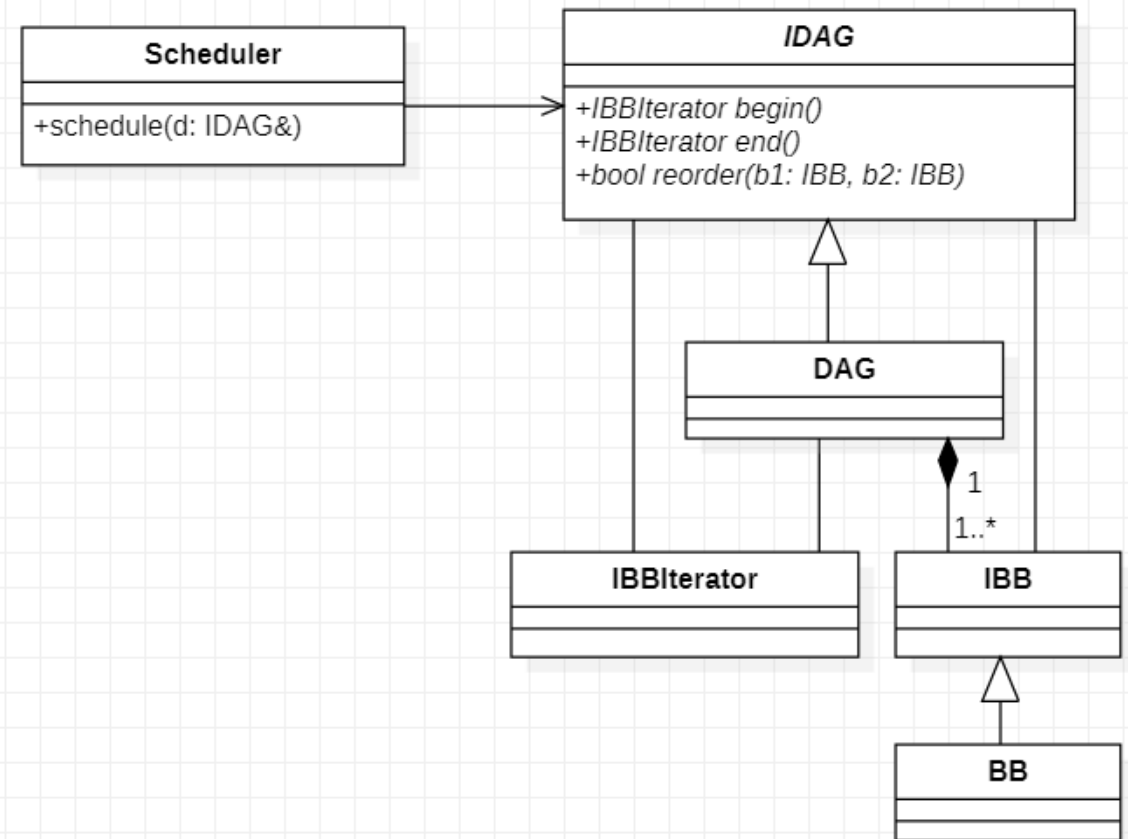
# Пример плохого проектирования (DIP)



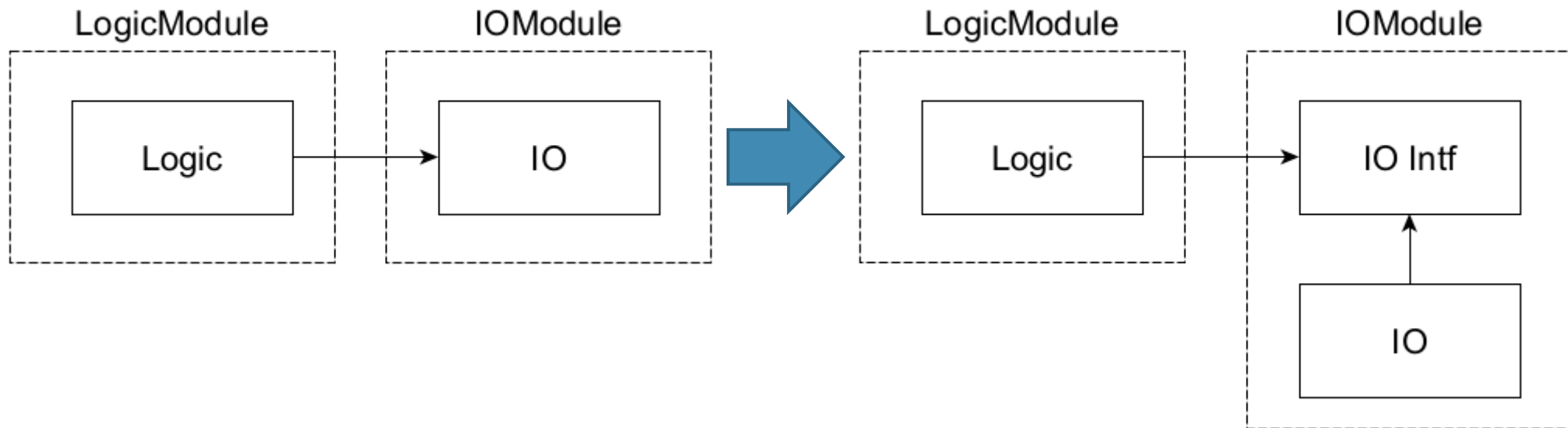
*"Dependency is the key problem in software development at all scales"*  
(Kent Beck)

# Принцип инверсии зависимостей

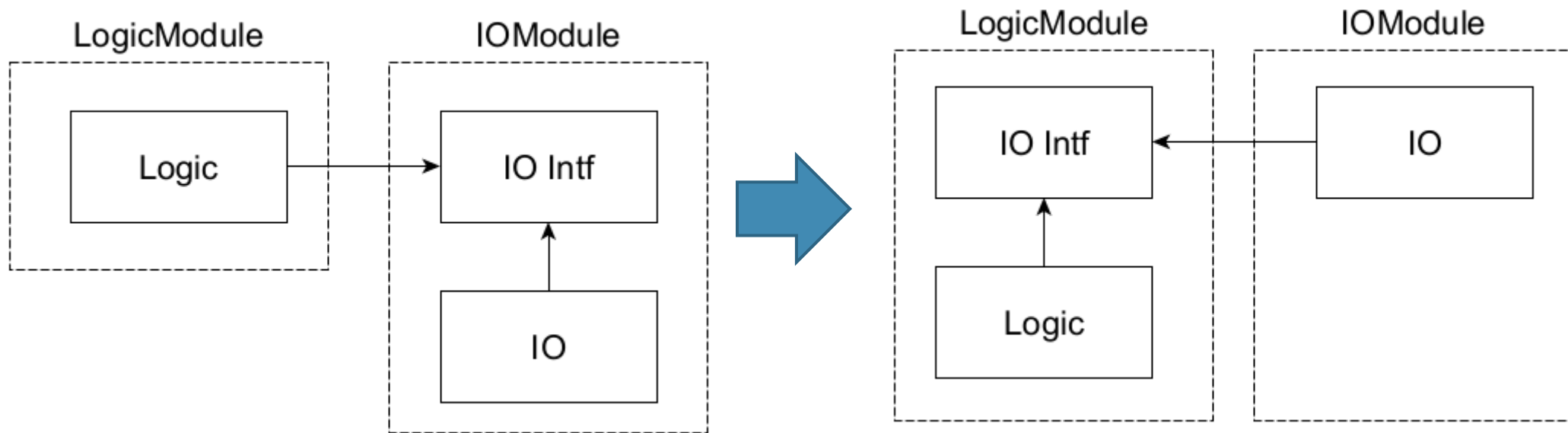
- Высокоуровневые классы не зависят от низкоуровневых
- Вместо этого и те и другие зависят от абстракций
- Scheduler знает только об интерфейсе, следовательно то, что за этим интерфейсом легко заменить



# Обсуждение: почему inversion?



# Обсуждение: почему inversion?



- ❑ Проектирование и UML
- ❑ Принципы SOLID
- Правила хорошего кода
- ❑ Паттерны проектирования

# Гуманитарная составляющая

- Де Марко и Листер писали, что программист в среднем занимается не научной или технической деятельностью, а деятельностью социальной
- Это на сто процентов верно для бухгалтерии, веб-программирования и т.п.
- Но даже для компиляторостроения, высоконагруженных систем и всего такого интересного соотношение ~ 80/20 в пользу гуманитарных задач
- Программный код больше похож на чертёж здания, чем на доказательство теоремы. Поэтому говорят о "качестве", "архитектуре", "проекте"
- Поговорим о качестве. Что такое хороший код?



# Хороший код

- Объективные критерии качества есть, но они очевидно не о том
  - скорость работы
  - время до поставки пользователю
  - количество найденных дефектов на строчку
  - искусственные критерии вроде цикломатической сложности и т.д. (увы, но все эти требования может легко выполнить чудовищная адская индусская лапша)
- Субъективные критерии ("когда я лично назову код хорошим")
  - читаемость
  - расширяемость
  - разумный выбор алгоритмов и абстракций
- Любой человек защищается. Главное свойство плохого кода: **его написал не я**

# Хороший код

- Многие принципы хорошего кода с первого взгляда спорны, но они формировались годами и написаны кровью
- Таковы принципы SOLID для ООП
- Таковы ещё два важных принципа которые применимы вообще везде
- Law of Demeter или Principle of least information
  - Контекст не должен давать пользователю заглядывать в более низкие уровни абстракции напрямую
- Principle of least astonishment
  - То что программист видит в коде не должно его удивлять и запутывать

# Пример плохого проектирования

- Здесь явно что-то идёт не так

```
class Options {  
    Directory current_  
    // ....  
public:  
    Directory &getDir() const; // returns current_  
    // ....  
};
```

```
Options opts(argc, argv);
```

```
string path = opts.getDir().getPath();
```

# Закон "Деметры"

- Уберём раскрытие пользователю интерфейса напрямую

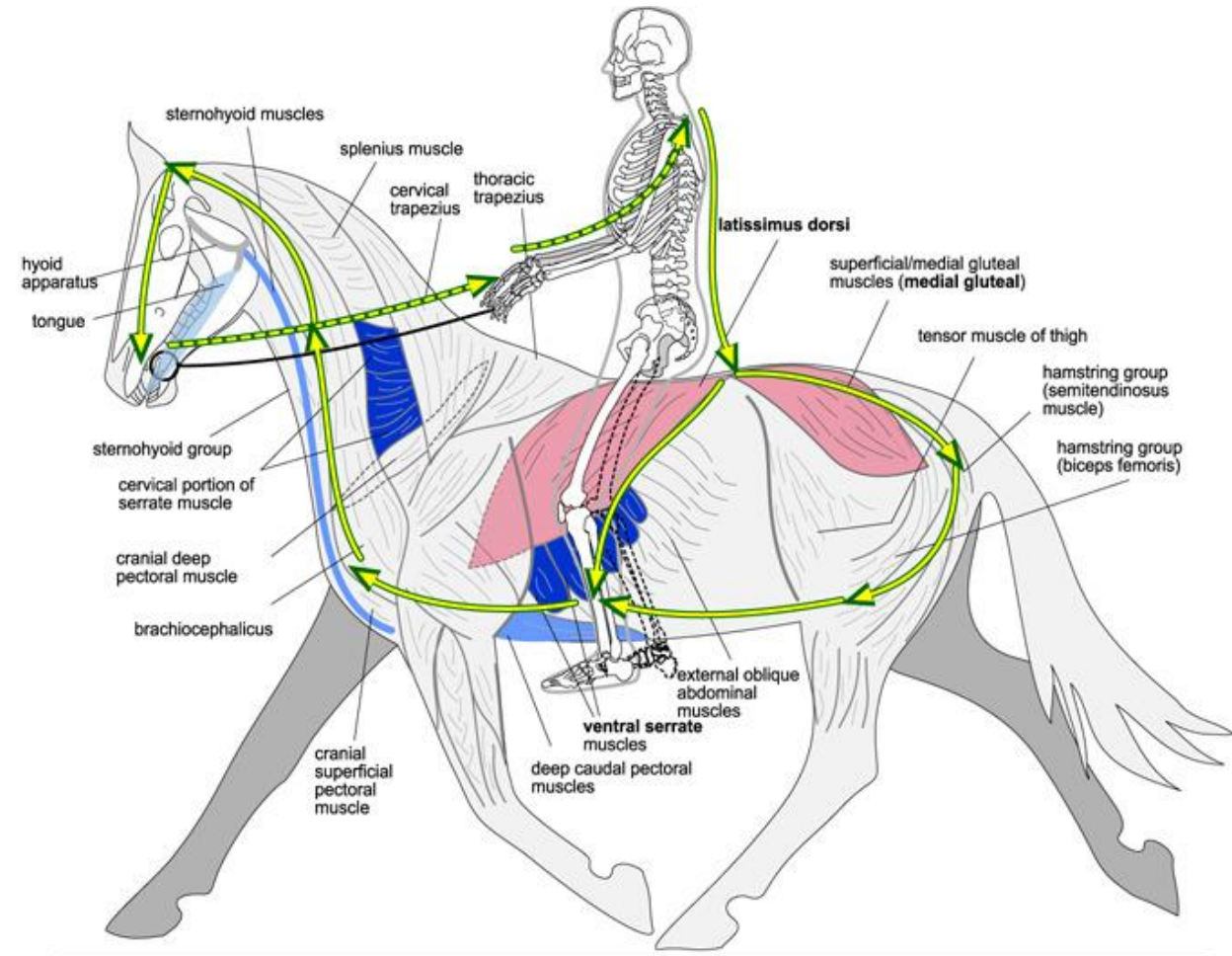
```
class Options {  
    Directory current_  
    // ....  
public:  
    string getPath() const; // returns current_.getPath()  
    // ....  
};
```

```
Options opts(argc, argv);
```

```
string path = opts.getPath();
```

# Аллегория закона "Деметры"

- Всадник должен управлять лошадью, но не ногами лошади
- Было бы странно, если бы всадник получил интерфейс к нервам, позволяющим двигать ногами лошади напрямую
- Но именно это регулярно происходит в плохо спроектированных системах



# Пример плохого проектирования

- Допустим для удобства мы спроектировали множество перегрузки так

```
// parses "010" as 8, "0x10" as 16, "10" as 10  
int strtoint(string s);
```

```
// respects user radix  
int strtoint(string s, int radix);
```

- На какие проблемы может наткнуться программист невнимательно читавший документацию?
- Всегда ли программисты внимательно читают документацию?

# POLA: убираем удивительное

- Для наименьшего удивления мы можем устроить функцию так

```
// radix = 10 if not specified  
int strtoint(string s, int radix = 10);
```

- Теперь при неправильном использовании будет разумная ошибка
- Вторую можно оставить как

```
// parses "010" as 8, "0x10" as 16, "10" as 10  
int smart_strtoint(string s);
```

- ❑ Проектирование и UML
- ❑ Принципы SOLID
- ❑ Правила хорошего кода
- Паттерны проектирования



# Идея паттернов проектирования

- Паттерны проектирования были придуманы Гаммой, Влиссидесом и прочими
  - Чтобы программисты могли общаться о проектировании не вдаваясь в детали
  - Чтобы выделить и закрепить проверенные и надёжные проектные решения, часть из которых они опубликовали в своей книге [*GOF*]
- Идея прижилась и сейчас обзорное знание классических паттернов это часть общей культуры программиста
- Классические паттерны делятся на порождающие, структурные и поведенческие

# Порождающие паттерны: обзор

- **Фабричный метод** – статический метод, выполняющий функции "виртуального конструктора"
- **Прототип** – то же, но для "виртуального конструктора копирования"
- **Абстрактная фабрика** – базовый тип для создания в его наследниках групп ассоциированных объектов
- **Синглтон** – объект с приватным конструктором и статическим методом создания, будет разобран далее
- **Строитель** – кусочное создание объекта для большей гибкости

# Порождающие паттерны: синглтон

- Иногда некий объект идеологически единственный на всю программу

```
// отображение на экран
class ViewPort {
    ViewPort();

public:
    // ....
    static ViewPort *queryViewPort();
};
```

- Такой паттерн называется синглтон и он наиболее известен среди прочих
- Многие считают его ничем не лучше глобальной переменной

# Порождающие паттерны: строитель

- В инфраструктуре LLVM мы хотим работать с любыми даже самыми причудливыми ассемблерами

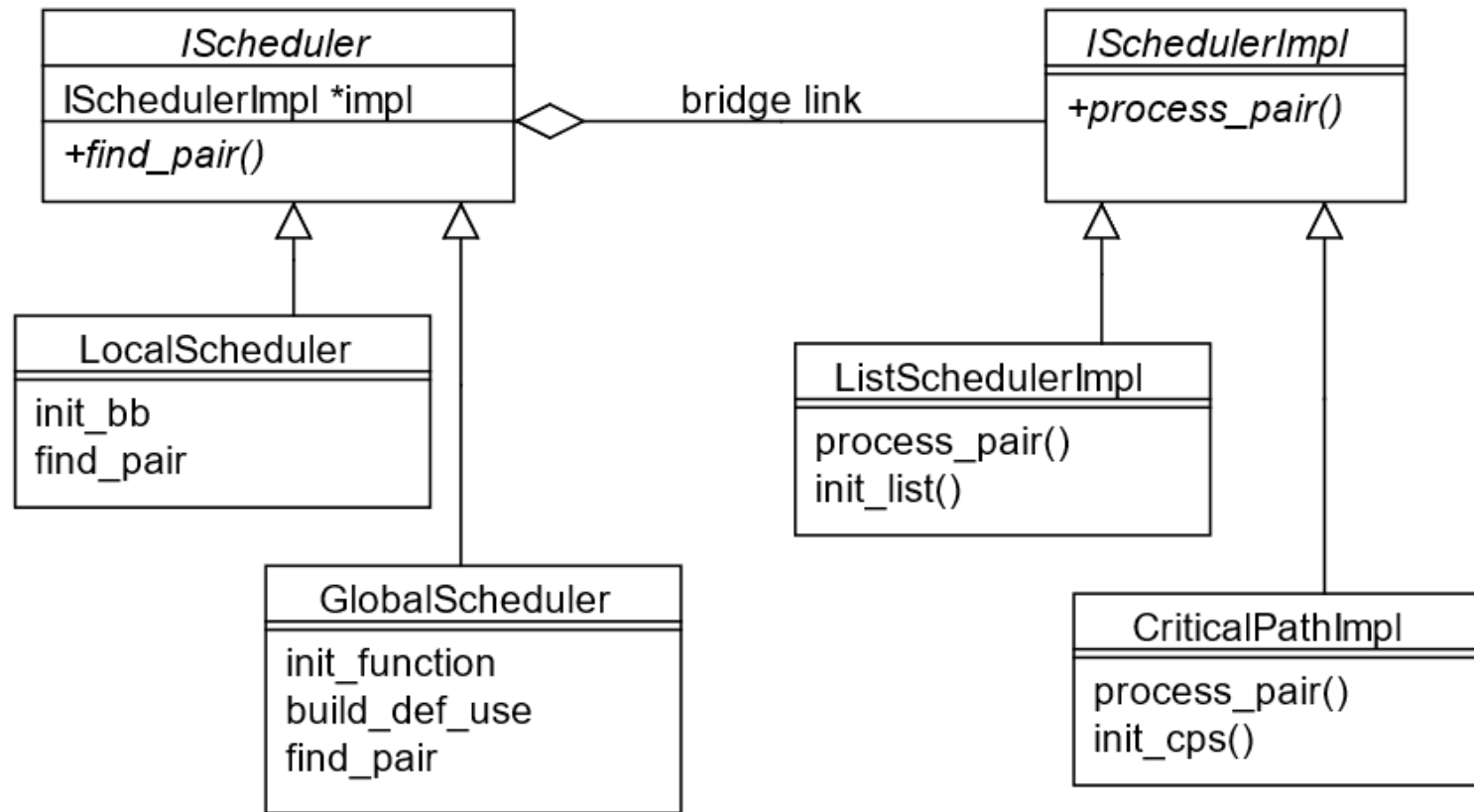
```
// we want ADD R0, R1, 1
MachineInstrBuilder NewMIB(ADD);
NewMIB.addReg(R0);
NewMIB.addReg(R1);
NewMIB.addImm(1);
MachineInstr *NewMI = NewMIB.get();
```

- Здесь MachineInstrBuilder предоставляет методы для гибкого абстрагирования от конкретного синтаксиса и способ создать любую мыслимую инструкцию

# Структурные паттерны: обзор

- **Адаптер** – изменяет интерфейс под требования пользователя
- **Декоратор** – расширяет интерфейс, не изменяя контекст
- **Фасад** – облегченный интерфейс для сложного контекста
- **Приспособленец (flyweight)** – пул идентичных объектов из которых пользователю либо возвращается существующий либо создаётся новый
- **Мост** – развязывает семейства конкретных классов через барьеры разделяемых абстракций

# Структурные паттерны: мост



# Поведенческие паттерны: обзор (ч. 1)

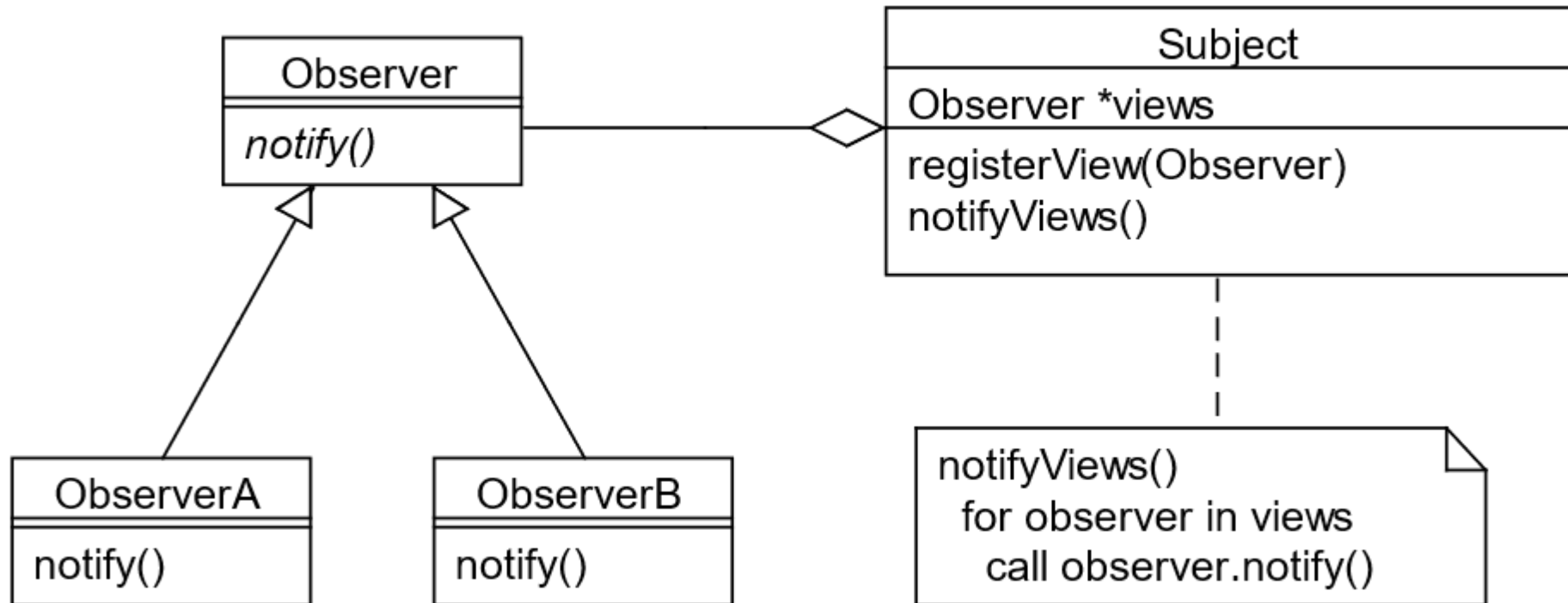
- **Команда** – объект инкапсулирующий одно действие и его параметры
- **Цепочка возможностей** – серия возможных объектов-обработчиков команды (например обработка и перевыброс исключений)
- **Интерпретатор** – DSL, встроенный в систему
- **Итератор** – объект для последовательного доступа к объекту, но без раскрытия структуры объекта
- **Посредник** – абстрагирует взаимодействия объектов, которые могут не знать даже интерфейс друг друга, обмениваясь через посредника
- **Хранитель** – сериализатор, встроенный в систему

# Поведенческие паттерны: обзор (ч. 2)

- Наблюдатель – устанавливает оповещение одного объекта об изменениях в другом
- Состояние – состояние конечного автомата. Имеет поведение и переход в другие состояния.
- Стратегия – общий интерфейс, определяющий методы, совместно используемые объектом для решения соответствующих задач
- Шаблонный метод – см. идиому NVI. Невиртуальная часть NVI это и есть шаблонный метод.
- Посетитель – операция, которая выполняется над объектами других классов



# Паттерн наблюдатель



# Пример модели: целые числа

```
IntVal subj;  
DIVObs divObs1(&subj, 4); // наблюдает (subj / 4)  
DIVObs divObs2(&subj, 3); // наблюдает (subj / 3)  
MODObs modObs(&subj, 3); // наблюдает (subj % 3)  
  
subj.setVal(14);  
  
cout << divObs1.observed() << " "  
      << divObs2.observed() << " "  
      << modObs.observed() << " " << std::endl;  
  
subj.setVal(18); // оповещает всех наблюдателей
```

- Домашняя наработка: написать такие классы IntVal, DIVObs и MODObs, которые бы позволили наблюдение

# Литература

- [CC11] ISO/IEC 14882 – "Information technology – Programming languages – C++", 2011
- [BS] Bjarne Stroustrup – The C++ Programming Language (4th Edition) , 2013
- [MDP] Robert Martin – Design Principles and Design Patterns, 2000
- [MDP] Robert Martin – Design Principles and Design Patterns, 2000
- [KB] Kent Beck – TDD by example, 2000
- [GOF] Gamma, Helm, Johnson, Vlissides – Design Patterns: Elements of Reusable Object-oriented Software, 2003
- [SM] Martin Reddy – API design for C++, 2011
- [DB] Steve McConnell – Code Complete: A Practical Handbook of Software Construction, 1993
- Breaking Dependencies: The SOLID Principles - Klaus Iglberger - CppCon 2020

# Бонус: антипаттерны

- **Детонатор** – паттерн, который ждёт в вашем коде и готов в любой момент разнести все к чертям
  - Хороший пример: отсутствие проверки на нулевой указатель
- **Бригада** – контейнерный класс для кривого и кособокого кода, методы в котором были отвергнуты разработчиками остальных классов
  - Вместе они – БРИГАДА
- **Сыр** – паттерн сыр полон дыр. Кстати, чем старше сыр, тем крепче запах
- **Посетитель из ада** – выход на единицу за границы массива, случайным образом затирающий значение важного флага дальше по стеку
- **Липучка** – очень плохой код, который вы назначены поддерживать до конца работы в компании (а то и жизни)
  - Примета: коготок в липучке увяз – всей птичке пропасть.