ЛЕКЦИЯ 21
ПРОЕКТИРОВАНИЕ

АЛГОРИТМИЗАЦИЯ И
ПРОГРАММИРОВАНИЕ

ЛЕКТОР ФУРМАВНИН С.А.

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;
// ...
```

Констекст (C++-style): matrix.hpp

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

Инварианты

- Указатель contents валиден, если $x \neq 0$
- Если $x \neq 0$, то всегда $y \neq 0$
- Для contents аллоцирована память размером $x \cdot y \cdot sizeof(T)$
- После клонирования матрица математически 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, то производный контекст связан с базовым дополнительными отношениями (частное/общее, быть частью и др.)
- Если несколько типов реализуют общий интерфейс, вызовы их методов через этот интерфейс полиморфны

ОБСУЖДЕНИЕ: ПРОЕКТИРОВАНИЕ

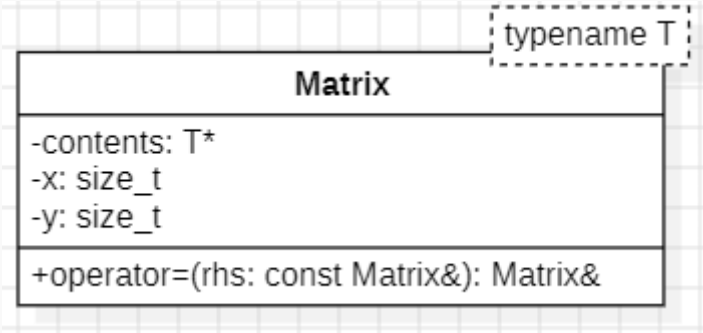
- Проектирование сложной системы классов – это человеческая деятельность
- Что является артефактом этой деятельности?
- Как можно было бы хотя бы частично формализовать этот процесс?

ОБСУЖДЕНИЕ: ЯЗЫК МОДЕЛИРОВАНИЯ

- Проектирование – это моделирований 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 прямоугольнике, поля – во втором, а методы – в третьем.
- Формат полей «поле : тип» (несколько контринтутивно для C++, но похоже на Python)
- UML поддерживает также тонны других атрибутов, например шаблонные параметры



https://www.uml-diagrams.org

ОТНОШЕНИЯ МЕЖДУ КЛАССАМИ И UML

Ассоциация показывает, что объекты одной сущности (класса) связаны с объектами другой сущности(отношение типа: «Я знаю о…»).

Агрегация – это разновидность ассоциации при отношении между целым и его частями (отношение типа: «Я знаю о… и без этого могу существовать»). Одно отношение агрегации не может включать более двух классов (контейнер и содержимое).

ОТНОШЕНИЯ МЕЖДУ КЛАССАМИ И UML

Композиция – более строгий вариант агрегации. Композиция имеет жёсткую зависимость времени существования экземпляров класса-контейнера и экземпляров содержащихся классов. Если контейнер будет уничтожен, то всё его содержимое будет также уничтожено. (Отношение типа: «Я знаю о… и без этого не могу существовать»).

ОТНОШЕНИЯ МЕЖДУ КЛАССАМИ И UML

Обобщение – процесс передачи открытого и защищённого состояния и поведения от одного класса другому (реализовано через парадигму ООП наследование).

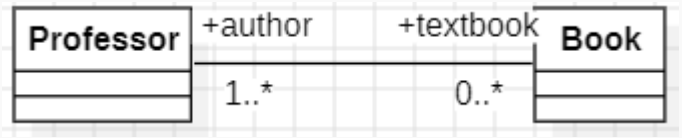
Реализация - напоминает наследование, но это более похоже на контрактный подход (использование интерфейсов).

ОТНОШЕНИЯ МЕЖДУ КЛАССАМИ И UML

Зависимость – это слабая форма отношения использования, при котором изменение в спецификации одного влечёт за собой изменение другого, причем обратное не обязательно.

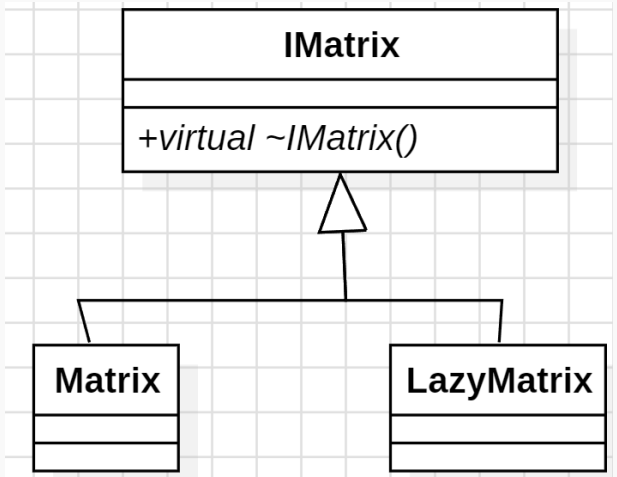
ОТНОШЕНИЯ МЕЖДУ КЛАССАМИ И UML

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



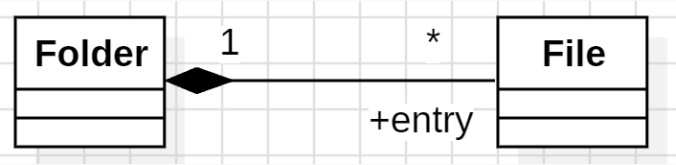
- Здесь также видно, что у каждой связи можно указать роли и множественность

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



ОТНОШЕНИЯ МЕЖДУ КЛАССАМИ И UML

- Композиция означает, что сущность B является частью сущности A



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

- Агрегация: сущность A владеет сущностью B, но кроме A у B 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**

ПРИНЦИПЫ SOLID

SRP – single responcibility principle

- каждый контекст должен иметь одну ответственность

OCP – open-close principle

- каждый контекст должен быть закрыт для модификации и открыт для расширения

LSP – Liskov substitution principle

- частный класс может иметь возможность заменять общий

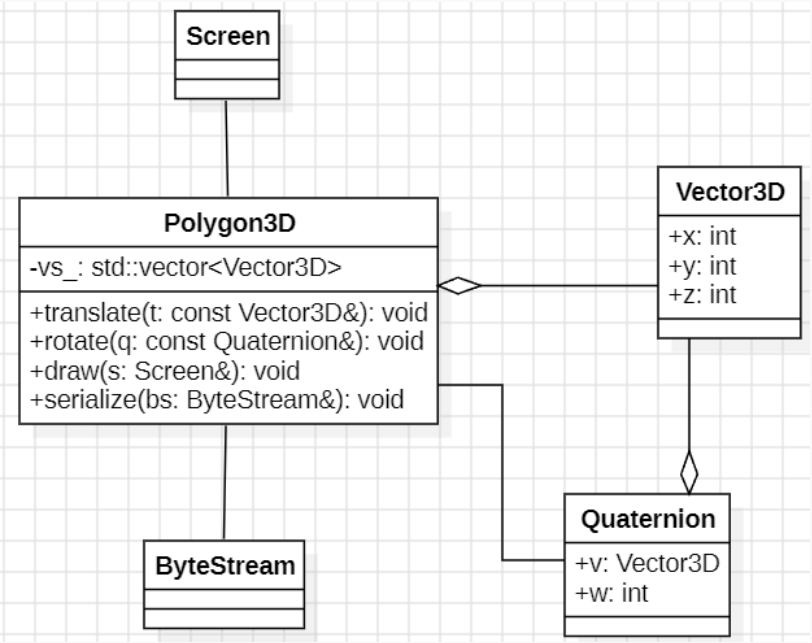
ISP – interface separation principle

- Тип не должен зависеть от тех интерфейсов, которые он не использует

DIP – dependency inversion principle

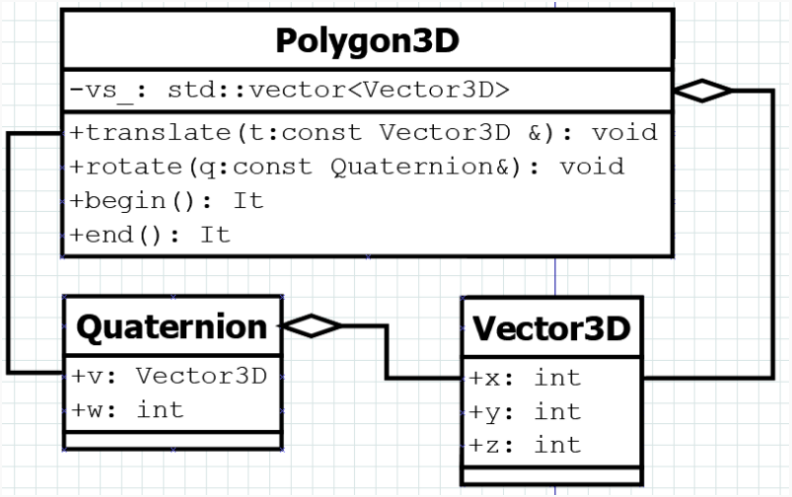
- Высокоуровневые классы не должны зависеть от низкоуровневых

ПРИМЕР ПЛОХОГО ПРОЕКТИРОВАНИЯ (SRP)



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

ПРИНЦИП ЕДИНСТВЕННОЙ ОТВЕТСТВЕННОСТИ

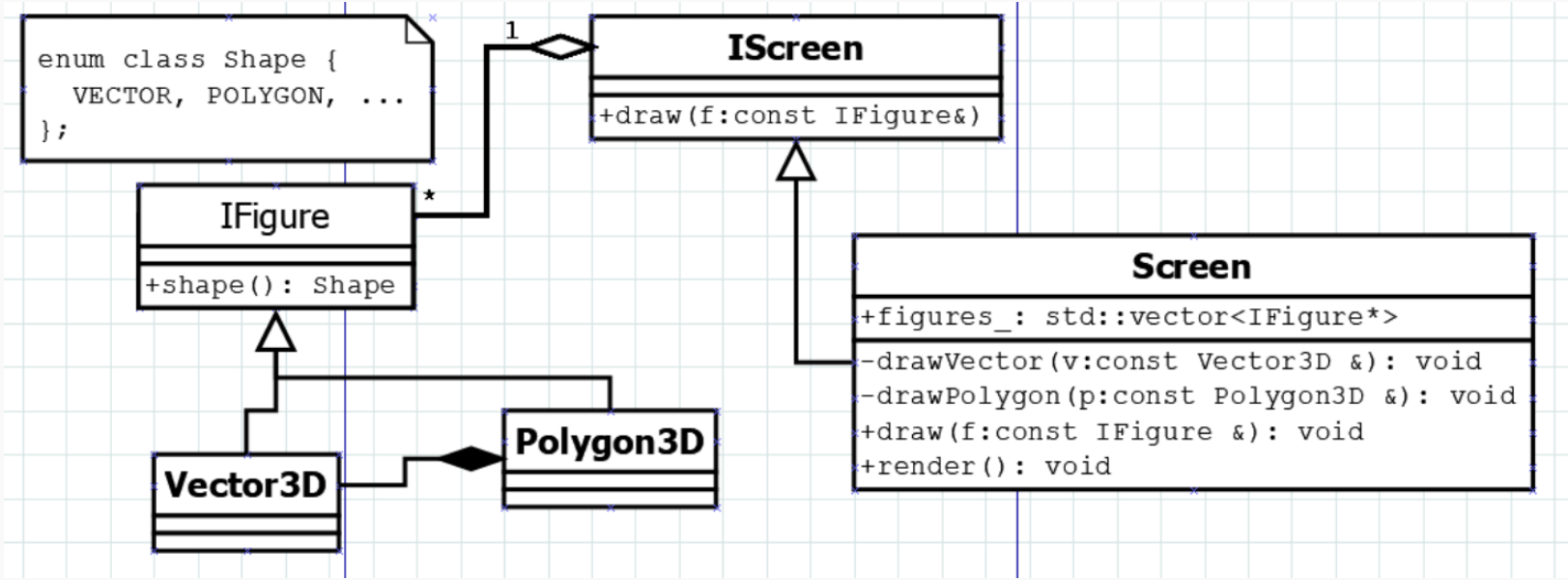


- Теперь единственная обязанность это геометрия
- Для вывода есть итераторы
- В итоге внешние функции могут обращаться к элементам но не к состоянию полигона
- «We want to design components that are self-contained: independent and with single well-defined purpose» (Andrew Hunt, David Thomas)

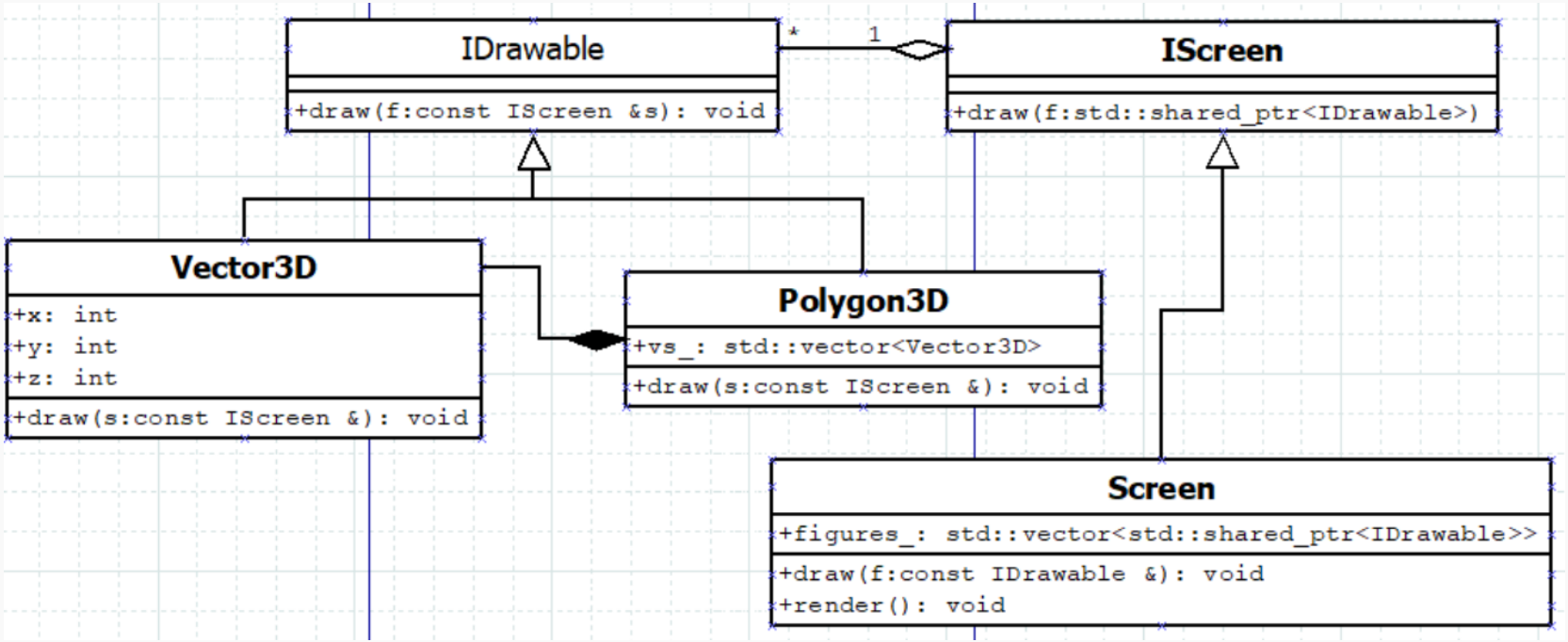
ГАЙДЛАЙН: СВЯЗНОСТЬ

- 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 (OCP)



ПРИНЦИП ОТКРЫТОСТИ И ЗАКРЫТОСТИ



ОБСУЖДЕНИЕ

- Такое чувство, что OCP 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);
```

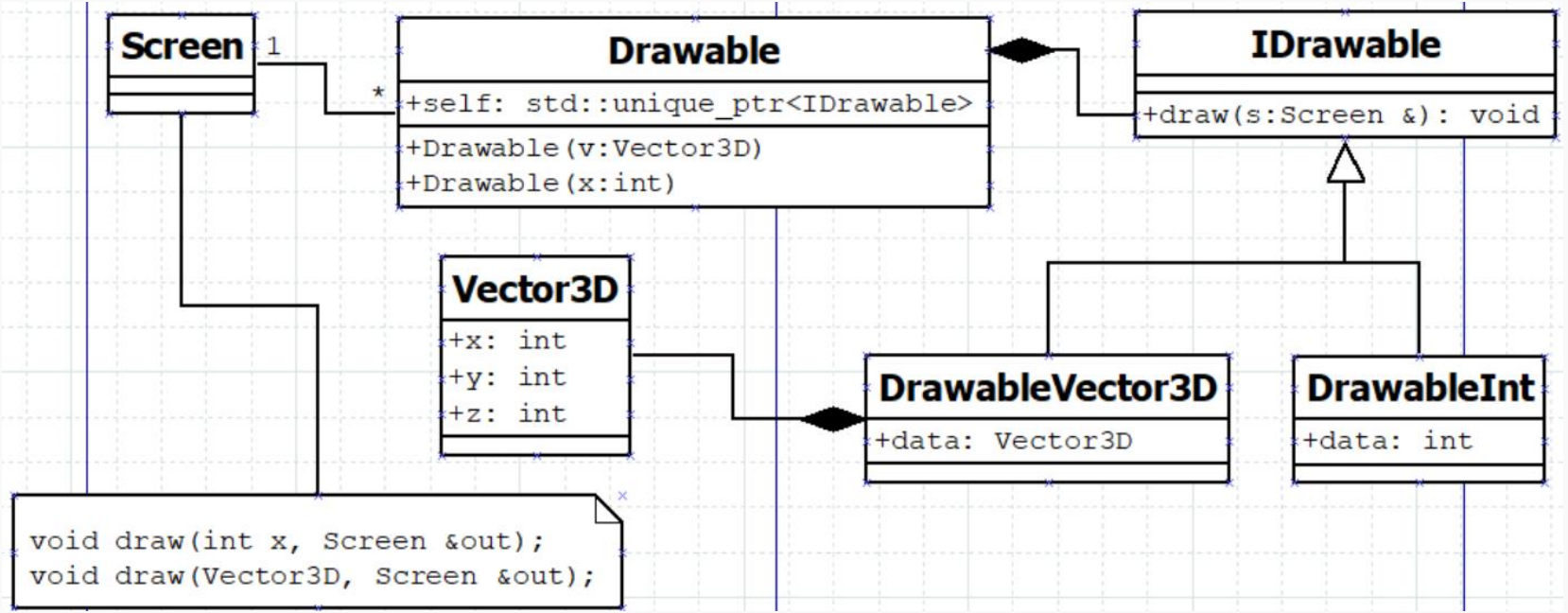
ОБСУЖДЕНИЕ

- Такое чувство, что OCP 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?

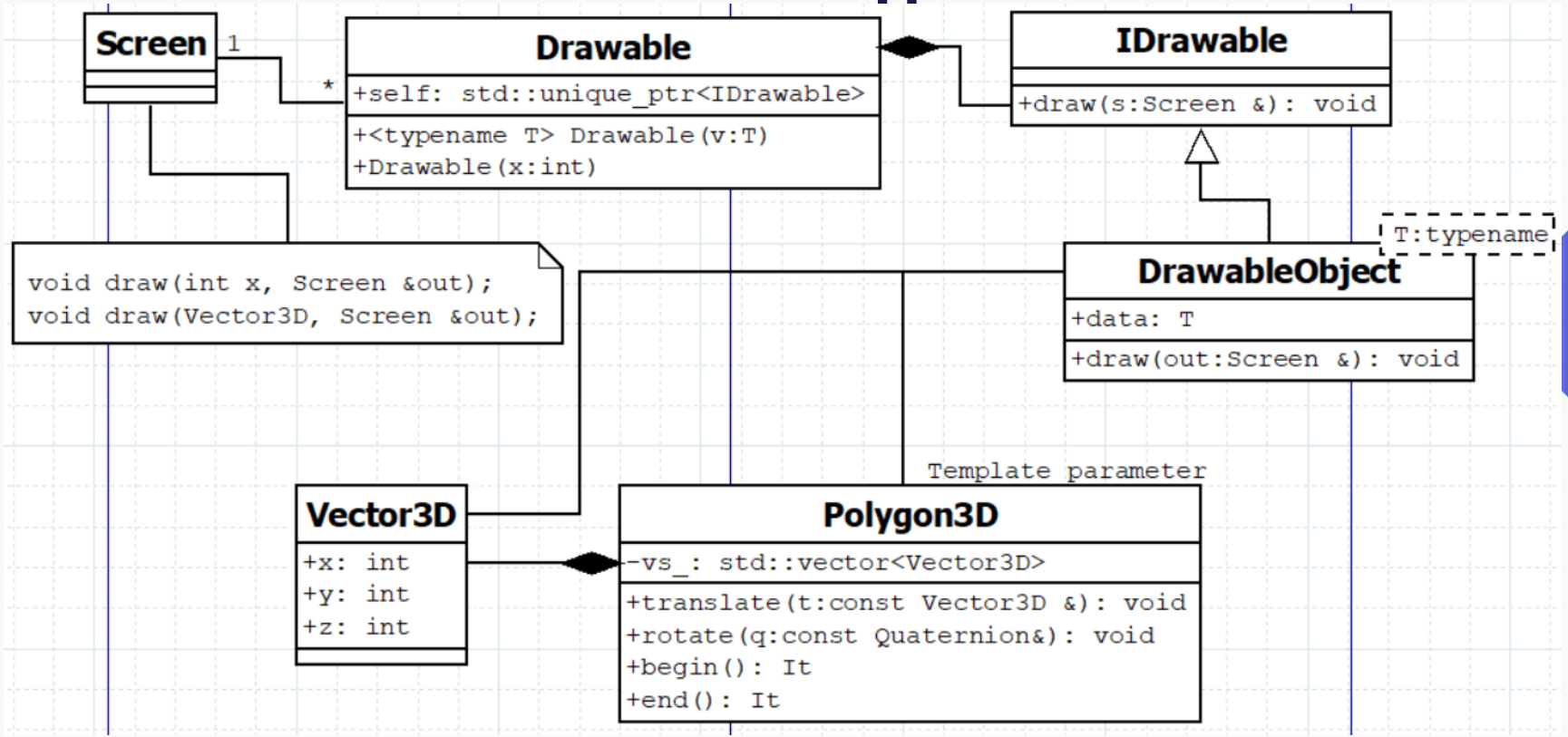
```
using document_t = std::vector<int>;
// документ хранит объекты
// семантика значения
// no incidental data structures
document.push_back(1);
document.push_back(2);
document.push_back(3);
draw(document, std::cout);

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?

ИНВАРИНТНОСТЬ

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.

КОНТРВАРИАНТНОСТЬ

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

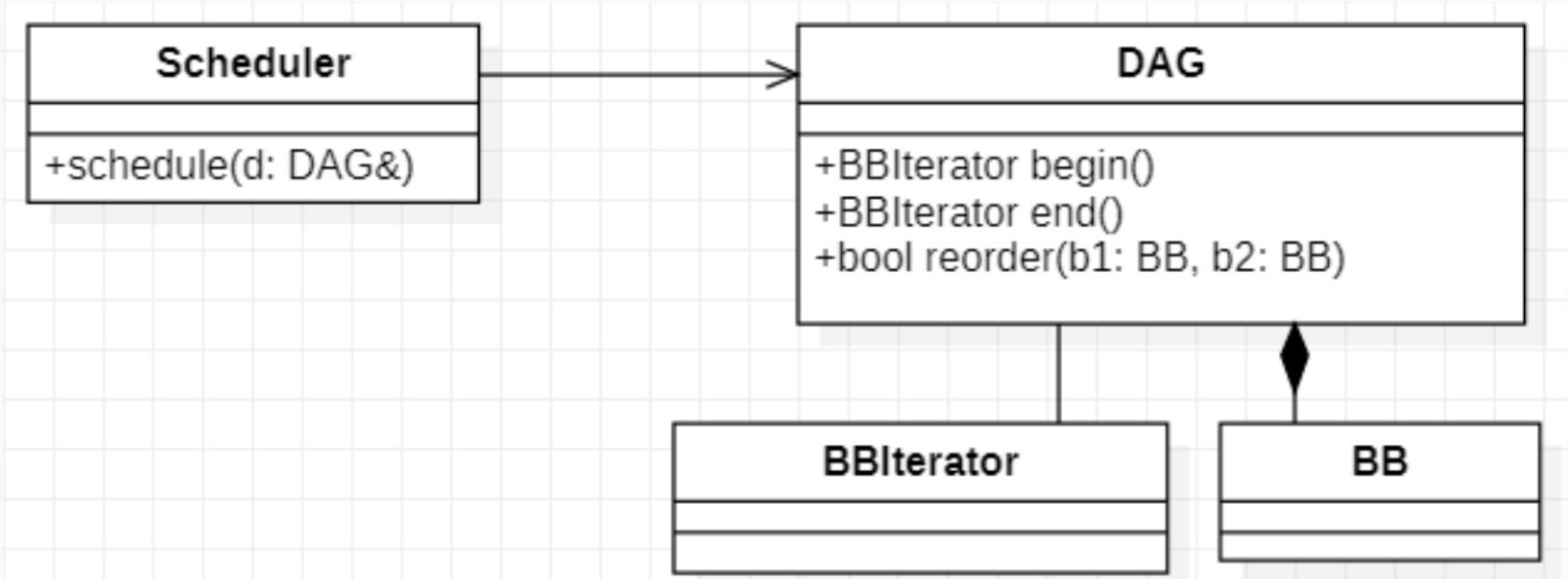
ПРИНЦИП РАЗДЕЛЕНИЯ ИНТЕРФЕЙСА

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

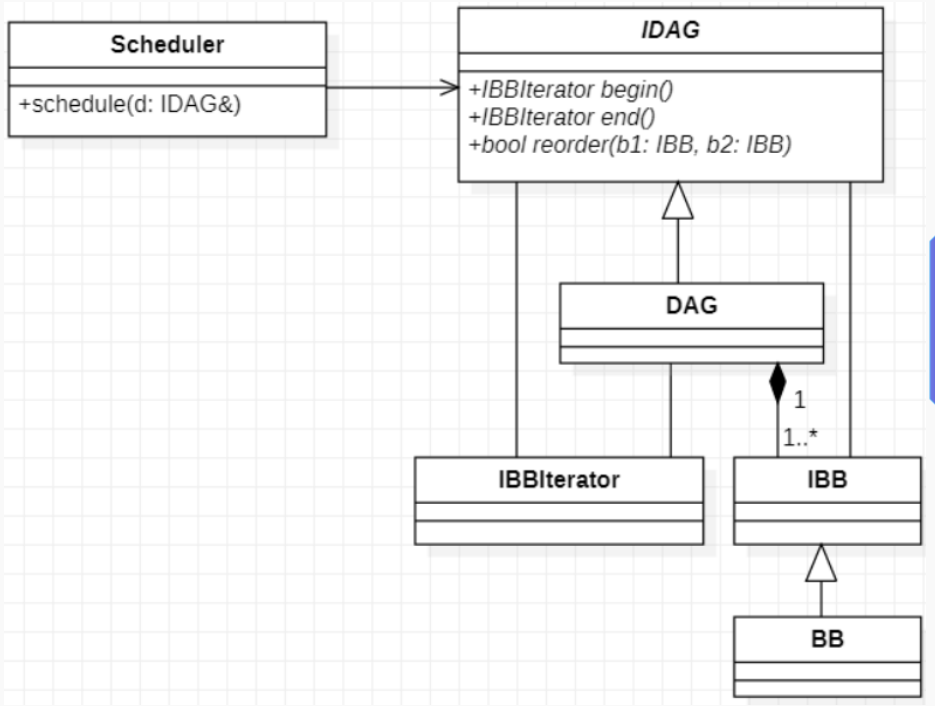
ПРИМЕР ПЛОХОГО ПРОЕКТИРОВАНИЯ (DIP)



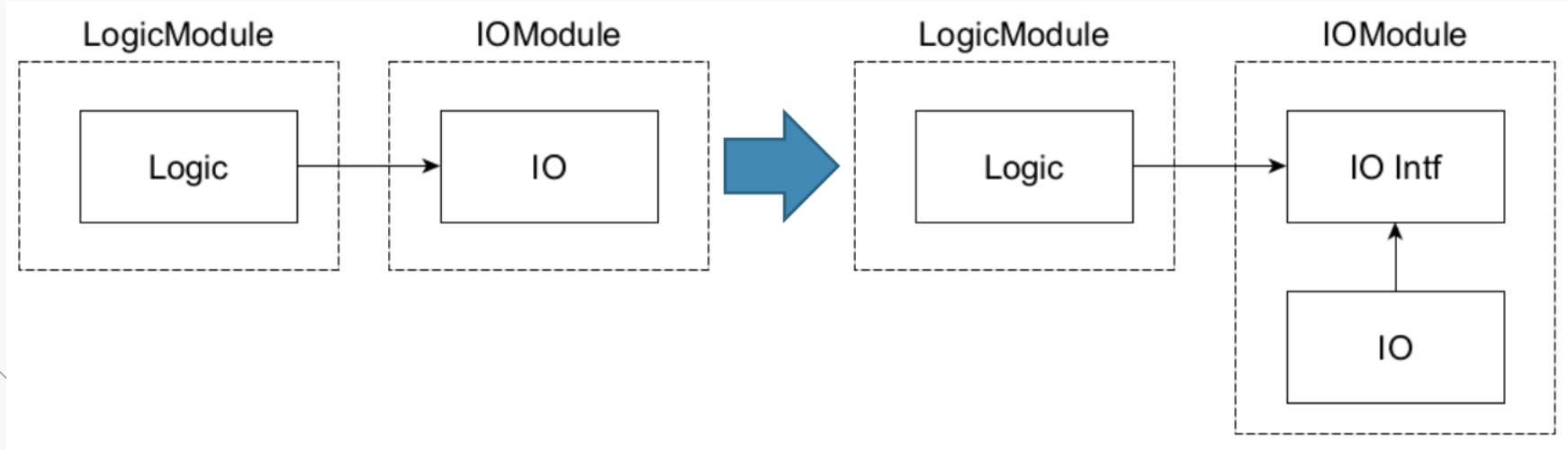
"Dependency is the key problem in software development at all scales"
(Kent Beck)

ПРИНЦИП ИНВЕРСИИ ЗАВИСИМОСТЕЙ

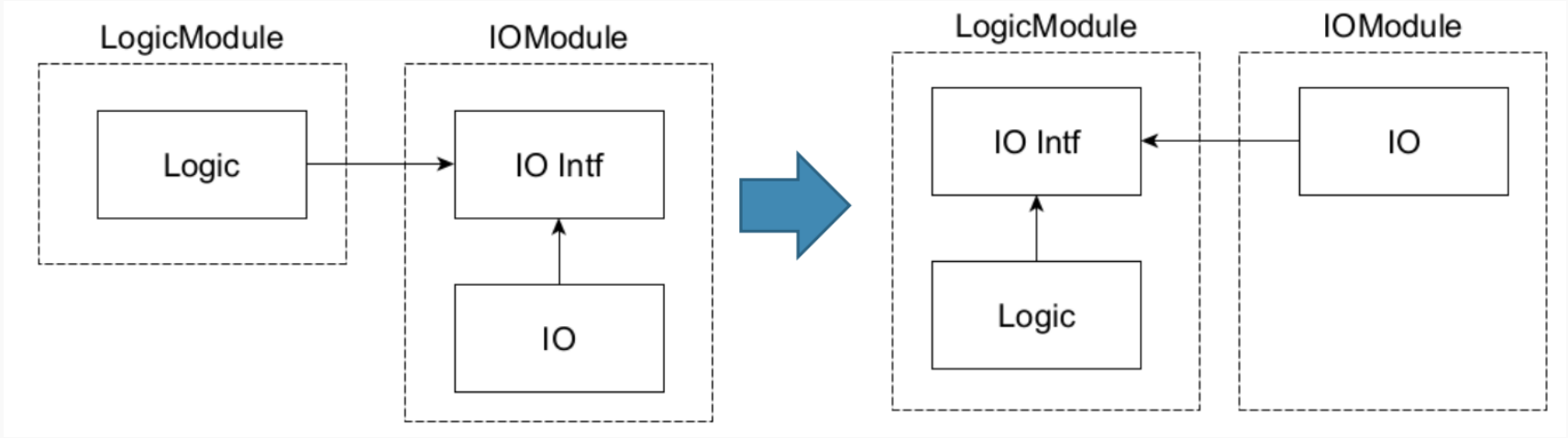
- Высокоуровневые классы не зависят от низкоуровневых
- Вместо этого и те и другие зависят от абстракций
- Scheduler знает только об интерфейсе, следовательно то, что за этим интерфейсом легко заменить



ОБСУЖДЕНИЕ: ПОЧЕМУ INVERSION?



ОБСУЖДЕНИЕ: ПОЧЕМУ INVERSION?



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»

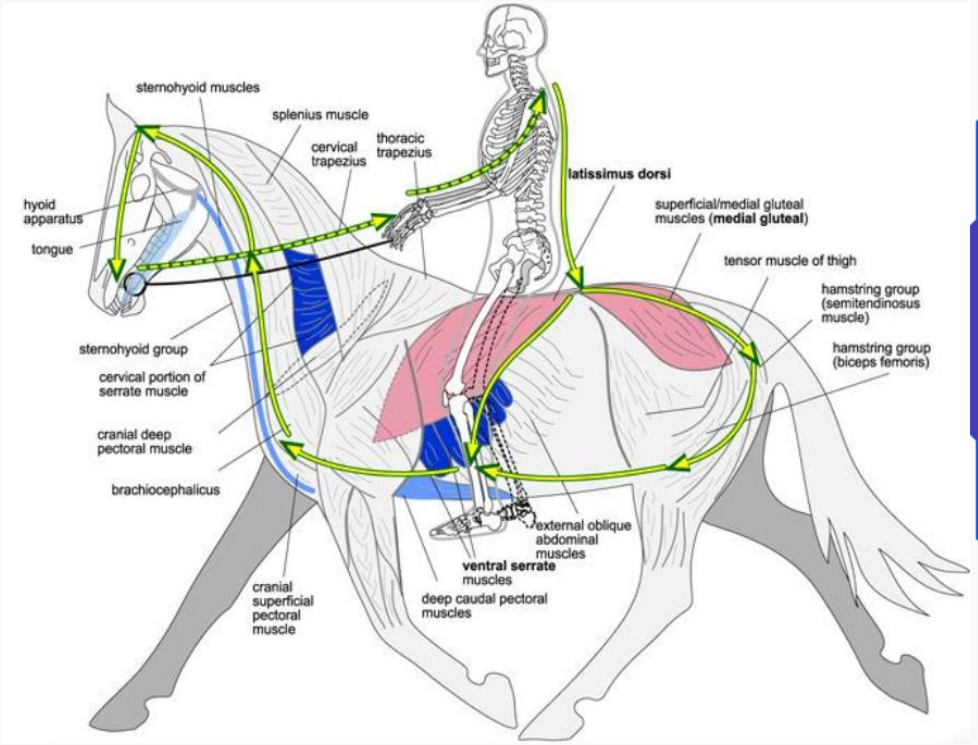
Уберём раскрытие пользователю интерфейса напрямую

```
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);
```

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
- 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

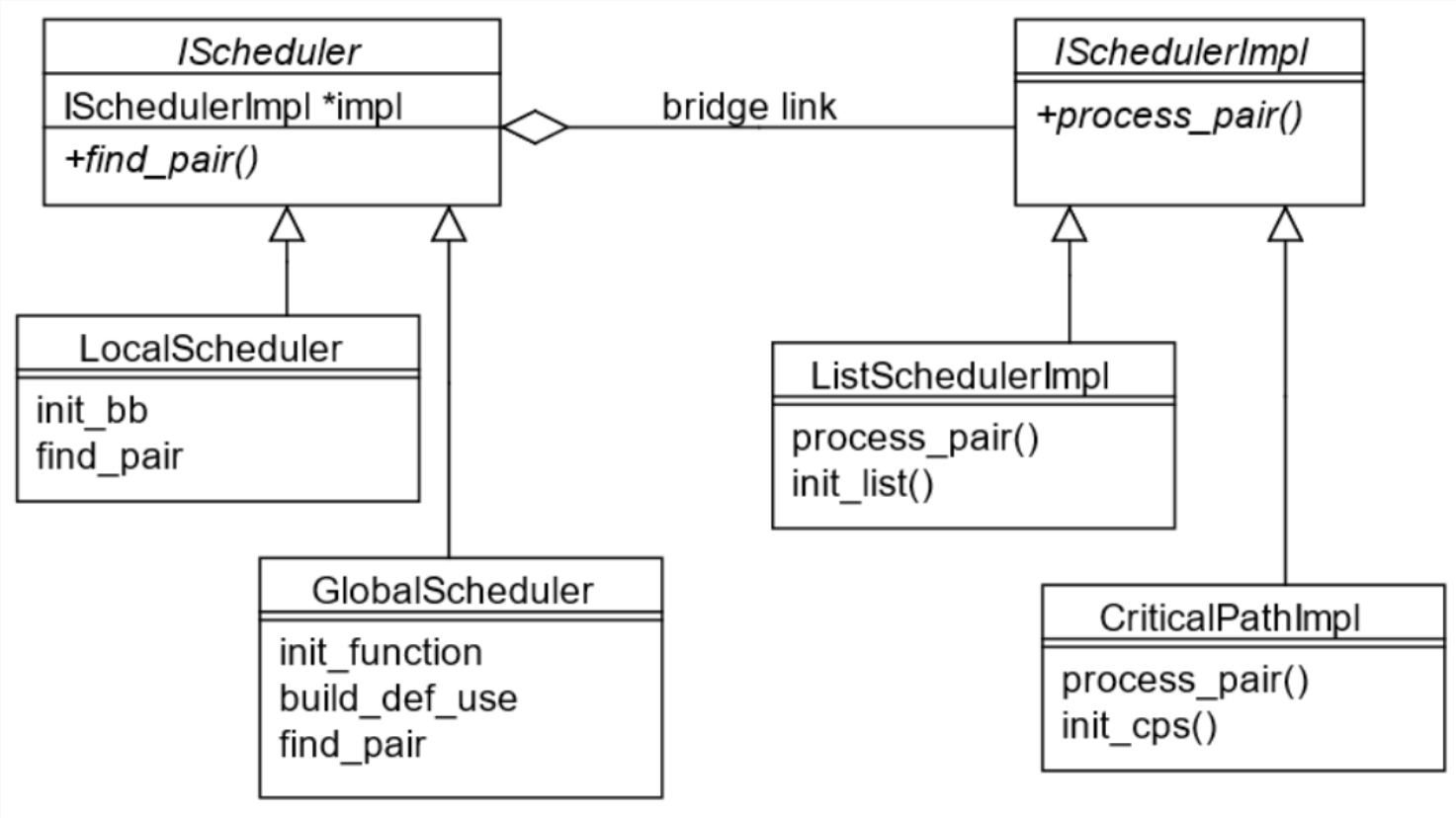
```
// we want ADD R0, R1, 1
MachineInstrBuilder NewMIB(ADD);
NewMIB.addReg(R0);
NewMIB.addReg(R1);
NewMIB.addImm(1);
MachineInstr *NewMI = NewMIB.get();
```

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

- **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

- **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

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 косого кода, методы в котором были отвергнуты разработчиками остальных классов
  - Вместе они – БРИГАДА
- **Сыр** – паттерн сыр полон дыр. Кстати, чем старше сыр, тем крепче запашок
- **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.

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

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