Прикладное программирование

Лекция №8.

Рассмотрим задачу

Условие задачи:

В рамках решения глобальной задачи по развертыванию системы «Умный дом» Вам необходимо написать программу для управления терморегулятором, который должен уметь контролировать и регулировать температуру в помещении согласно показаниям с датчиков.

Из чего состоит терморегулятор?

- Контроллер
- Система датчиков
- Нагревательный элемент
- Программное обеспечение

Что нам для этого нужно сделать?

1. Необходимо построить алгоритм, по которому будет выполняться наша задача.
2. Продумать, какие классы нам необходимы для решения.
3. Написать программу
4. Запустить ПО на контроллере.
5. Интегрировать разработку в готовую систему.

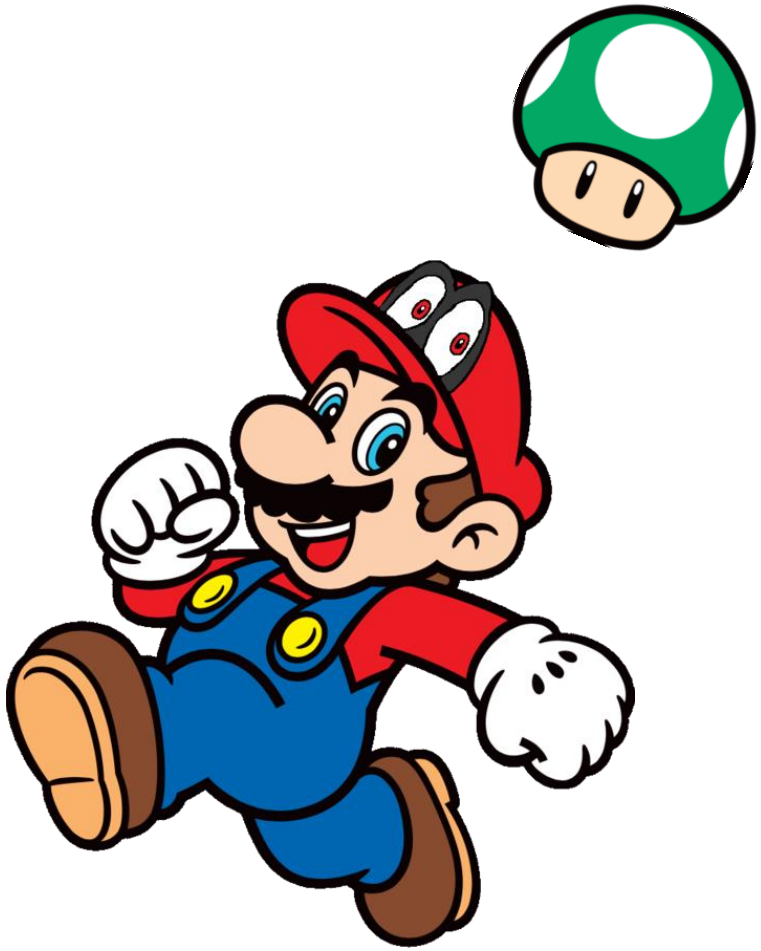
Приступаем к решению

1. Давайте обсудим алгоритм.
2. Создадим контроллер. Класс Controller.
3. Добавим датчик. Класс Sensor.
4. Добавим нагревательный элемент. Класс Heater.

Готово!

Можно программировать? Нет.

Level up, Mario!



Проектирование

На этапе проектирования архитектором или опытным разработчиком:

- Производится **анализ поставленной задачи**;
- Создается **проектная документация**, включающая текстовые описания модулей и компонентов программы;
- Создаются **диаграммы,** описывающие процессы внутри программы;
- Проектируются **модели** будущей программы.

Чаще всего все диаграммы и модели разрабатываются с помощью языка **UML**.

Что такое UML?

UML — графический язык для **визуализации**, **описания** параметров, **конструирования** и **документирования** различных систем.

Как создать UML-диаграмму?

1. Руками ☺
2. С помощью специальных **CASE-**средств:
   1. **Rational Rose** (http://www-01.ibm.com/software/rational/)
   2. **Enterprise Architect** (http://www.sparxsystems.com.au/)
   3. **Plantuml** (http://plantuml.com/ru/)

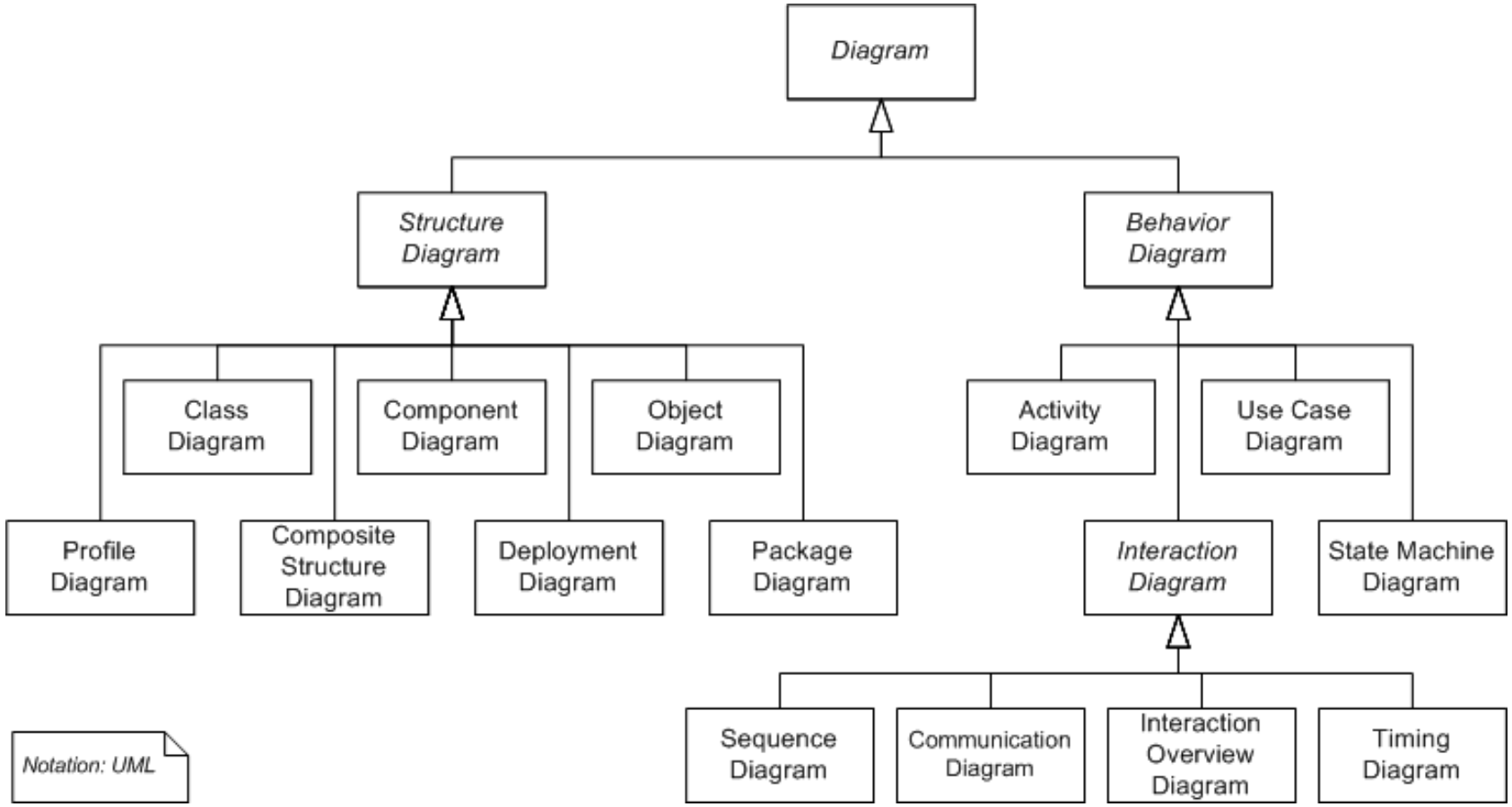
Чем хороши UML-диаграммы?

На основе технологии **UML** строится **единая информационная модель.**

Представленные **CASE-средства** способны генерировать код на различных объектно-ориентированных языках, а так же обладают очень полезной функцией **реверсивного инжиниринга**.

*(**Реверсивный инжиниринг** позволяет создать **графическую модель** из имеющегося программного кода и комментариев к нему.)*

Software Diagrams. Classification



Any Understandable Diagram & Terms are Good ☺

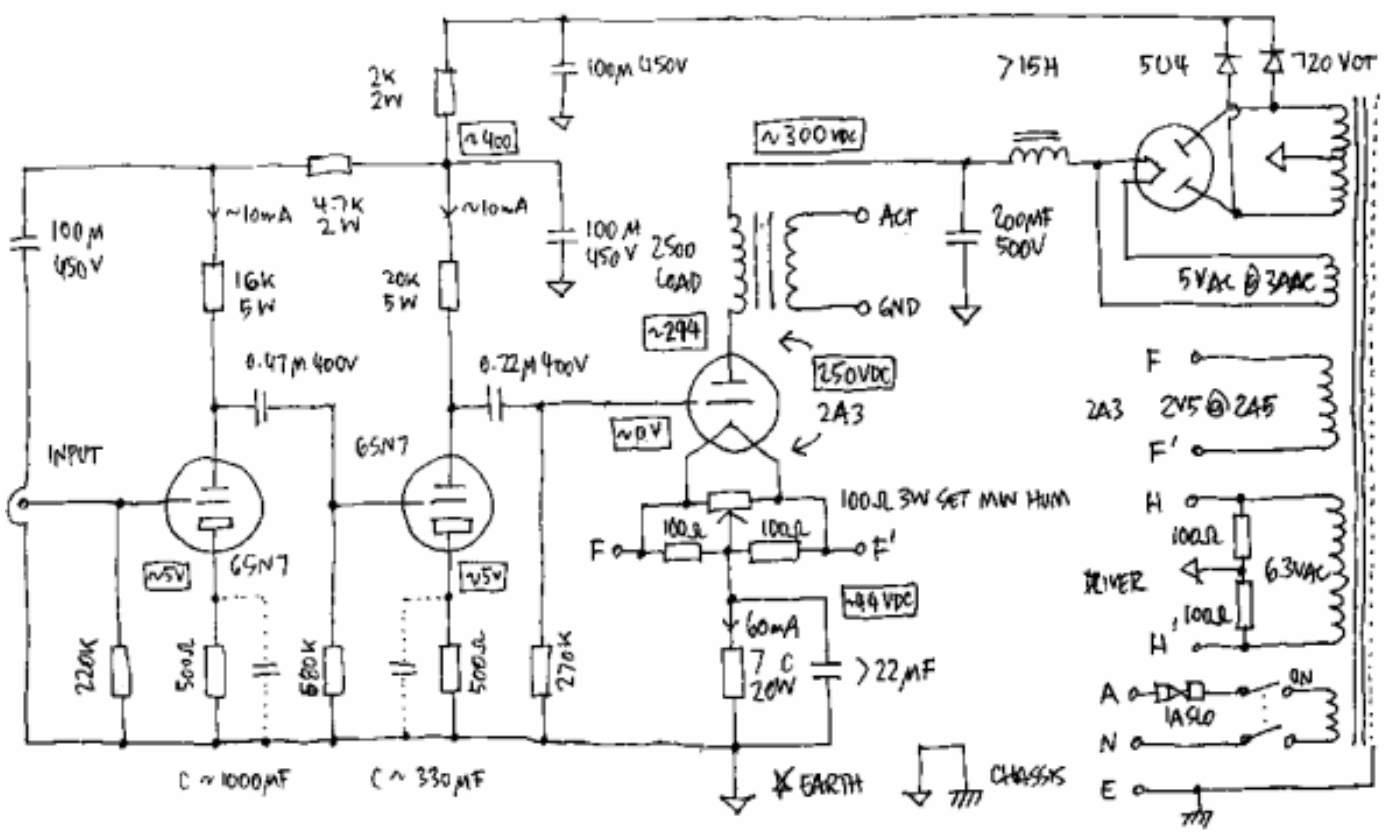


Диаграмма классов (class diagram)

Зачем?

Диаграмма классов служит для представления статической структуры модели системы в терминологии классов объектно-ориентированного программирования.

Что может?

Диаграмма классов может отражать различные **взаимосвязи между отдельными сущностями предметной области**, такими как **объекты** и **подсистемы**, а также описывает их **внутреннюю структуру** (поля, методы) и **типы отношений** (наследование, реализация интерфейсов, и т.д.).

Диаграмма классов (class diagram)

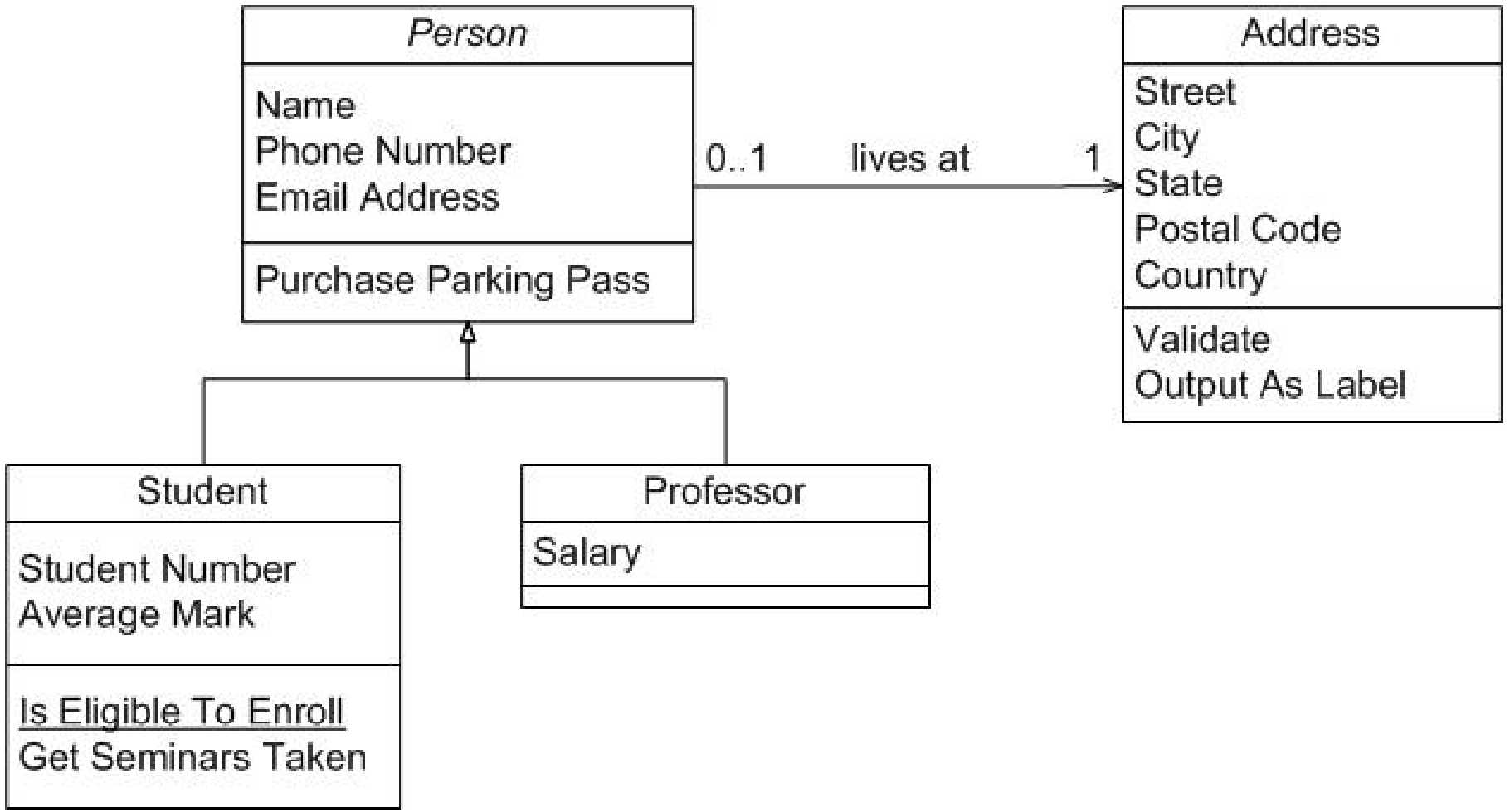
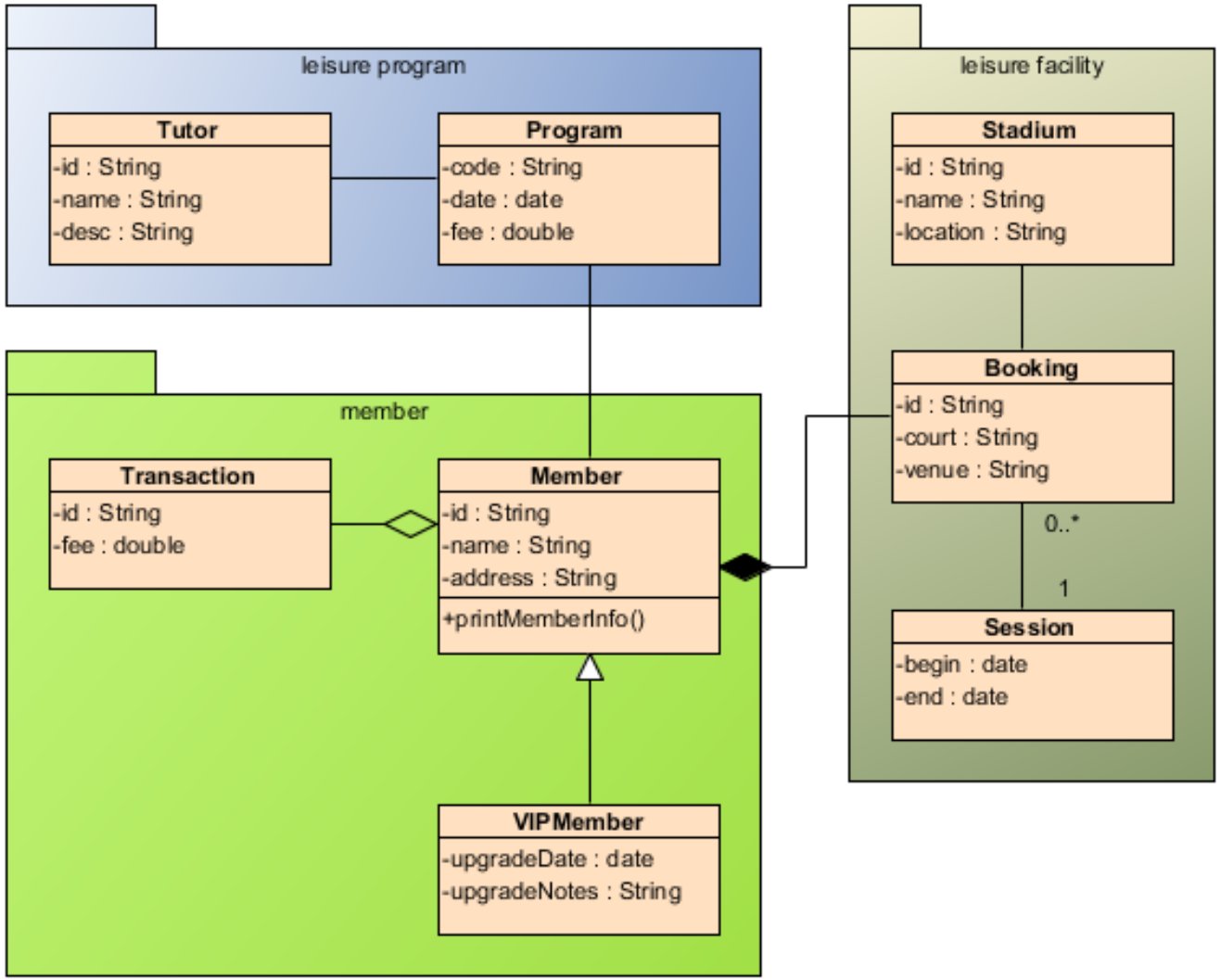
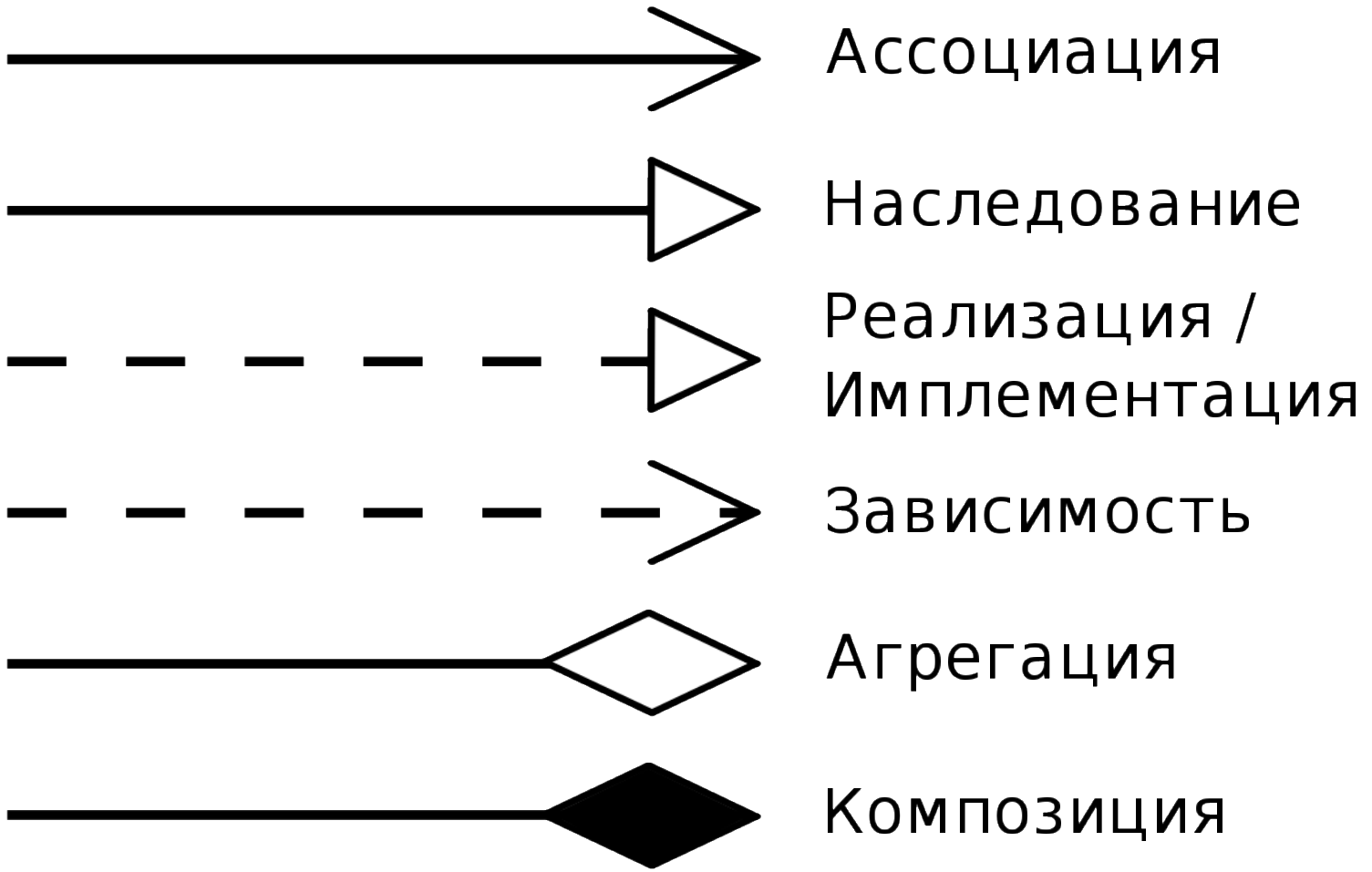


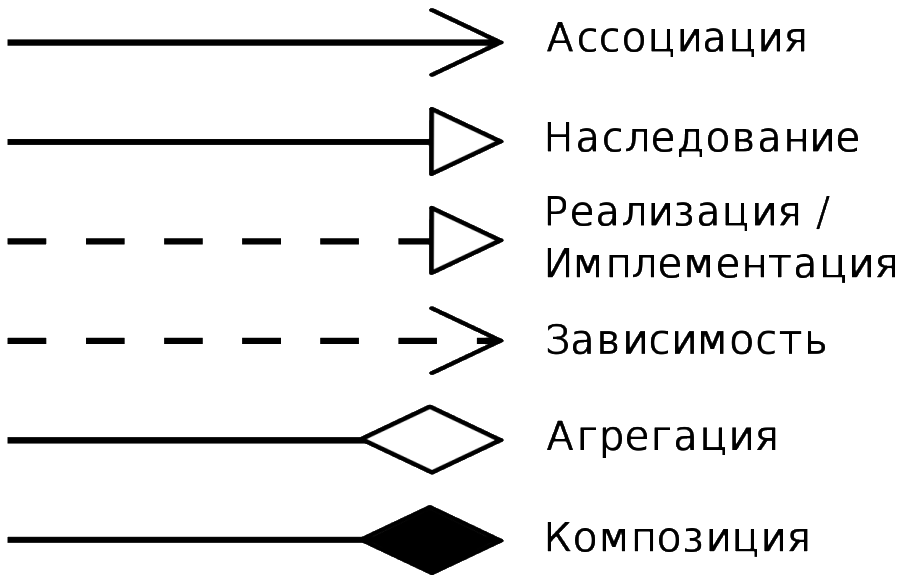
Диаграмма классов (class diagram)



Отношения между классами



Отношения между классами



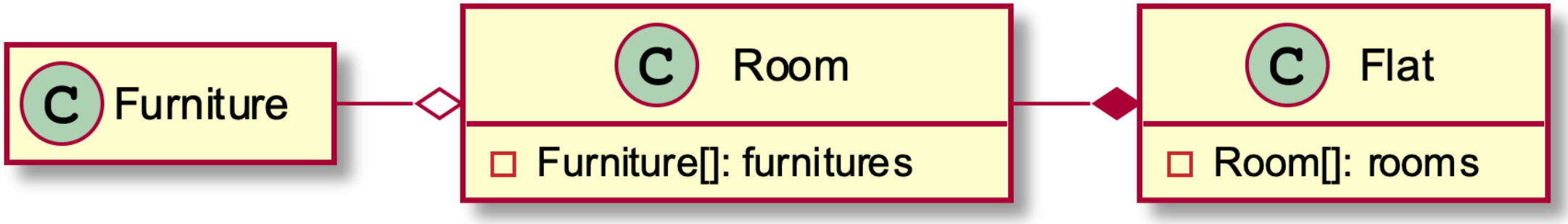
- **Агрегация** (aggregation) — связь **«часть»–«целое»**, в котором **«часть»** может существовать отдельно от **«целого»**. Ромб указывается со стороны **«целого»**.

- **Композиция** (composition) — подвид агрегации, в которой **«части»** не могут существовать отдельно от **«целого»**.

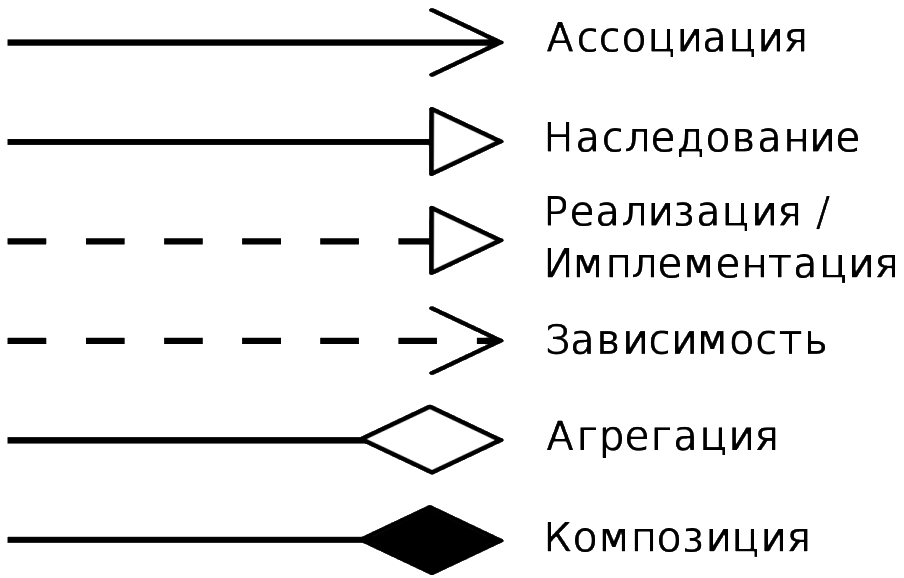
Композиция VS Агрегация

Комната (Room) является частью **квартиры (Flat)**, следовательно здесь подходит композиция, потому что комната без квартиры существовать не может.

А **мебель (Furniture)** не является неотъемлемой частью **квартиры (Flat)**, но в то же время, квартира содержит мебель, поэтому следует использовать агрегацию.



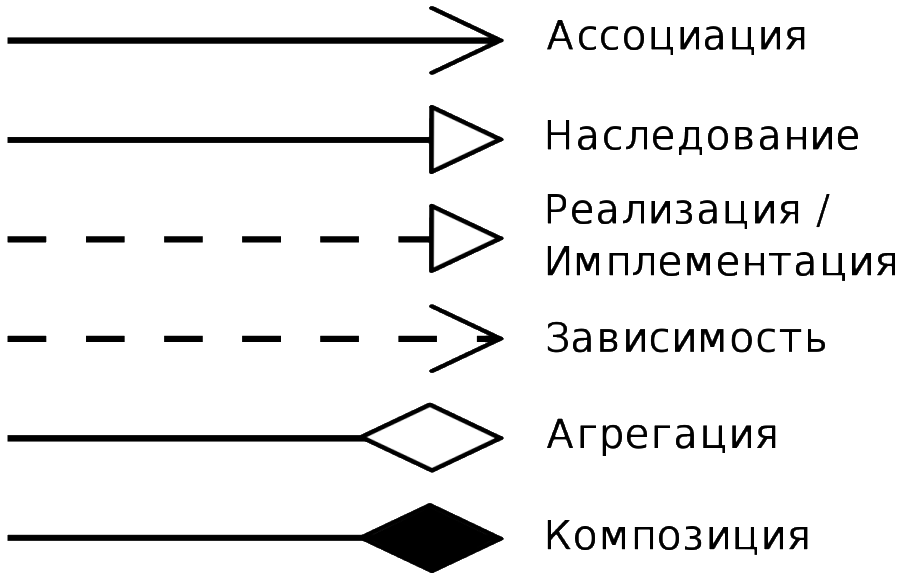
Отношения между классами



- **Ассоциация** (association) – тип отношения, где объекты одного класса связаны с объектами другого класса так, что можно логически перемещаться от объектов одного класса к другому. Является общим случаем композиции и агрегации.

- **Реализация** (implementation) — отношение **«часть»-«целое»** между двумя элементами модели, в котором один элемент (***клиент***) реализует поведение, заданное другим (***поставщиком***).

Отношения между классами



- **Зависимость** (dependency) — тип отношений, где изменение в одной сущности (независимой) может влиять на состояние или поведение другой сущности (зависимой). Со стороны стрелки указывается независимая сущность.

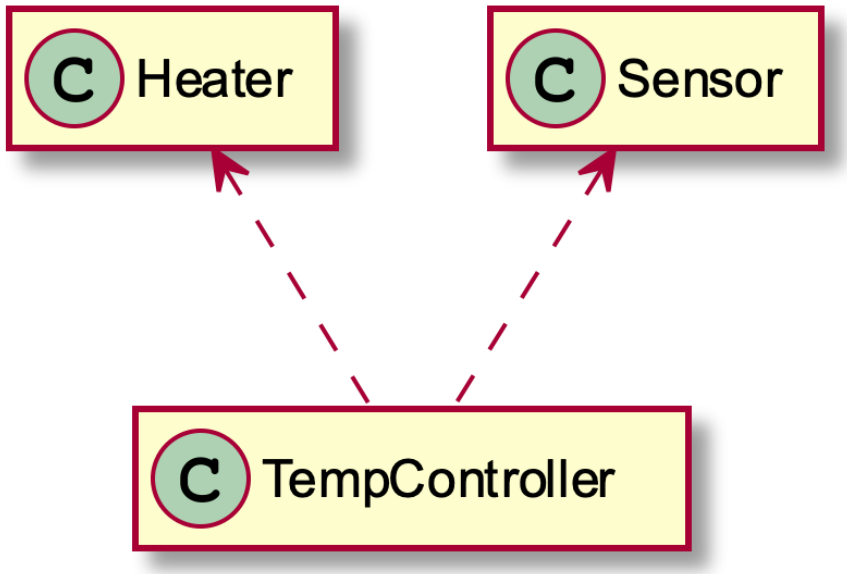
- **Обобщение** (generalization) — отношение, которое описывает наследование или реализацию интерфейса. Со стороны стрелки находится суперкласс или интерфейс.

Когда заказчик сказал, что этап проектирования можно пропустить

I’ll kill them, Winston. I’ll kill them all.

Вернемся к задаче

Построим диаграмму классов для задачи с терморегулятором.



Какие преимущества дают диаграммы?

- Вам не нужно «погружаться» в N классов и детально разбираться, какие задачи выполняет каждый из них.
- На диаграмме все классы представлены в одном месте прямо у Вас перед глазами.
- Вы легко можете определить и отследить все отношения и зависимости всех классов, интерфейсов и прочих компонентов в вашей программе.
- Можно найти все несоответствия, неточности и даже спрогнозировать потенциальные проблемы при дальнейшем развитии программы еще на «бумажном этапе». Таким образом можно избежать большого числа проблем и лишних затрат ресурсов и времени.

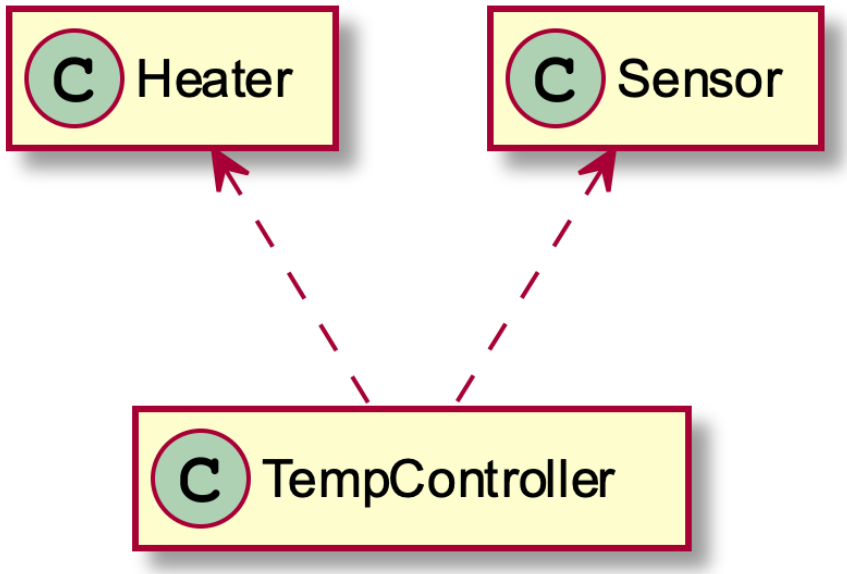
Вернемся к задаче

Чего не хватает для полноценного решения задачи?

- Устройство? – есть.
- Алгоритм? – есть.
- Классы? – есть.
- Диаграмма? – есть.
- Что еще нужно?

Вернемся к задаче

Диаграмма классов для задачи с терморегулятором.



НО!

Обновляем решение задачи

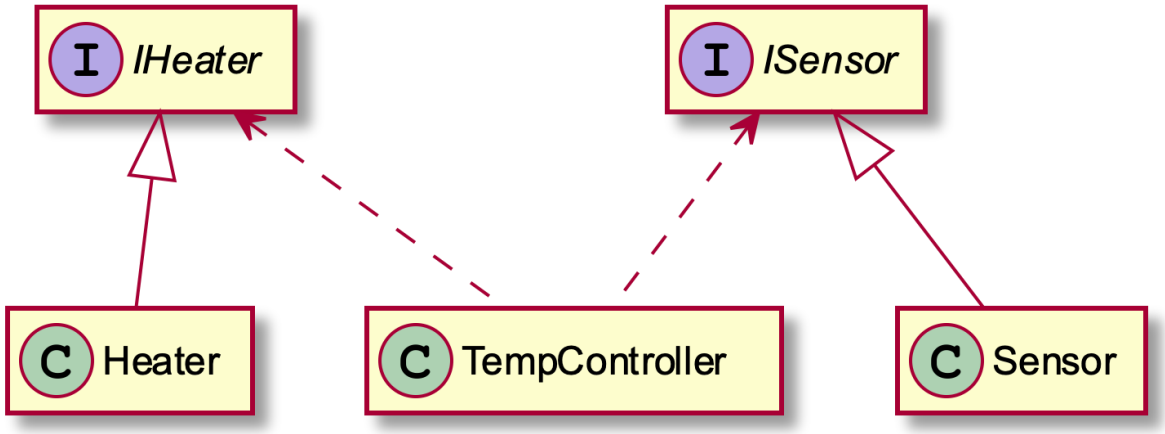
Ответим на несколько вопросов:

1. А как мы планируем тестировать работоспособность программы на реальном устройстве?

Обновляем решение задачи

Ответим на несколько вопросов:

1. А как мы планируем тестировать работоспособность программы на реальном устройстве?



Обновляем решение задачи

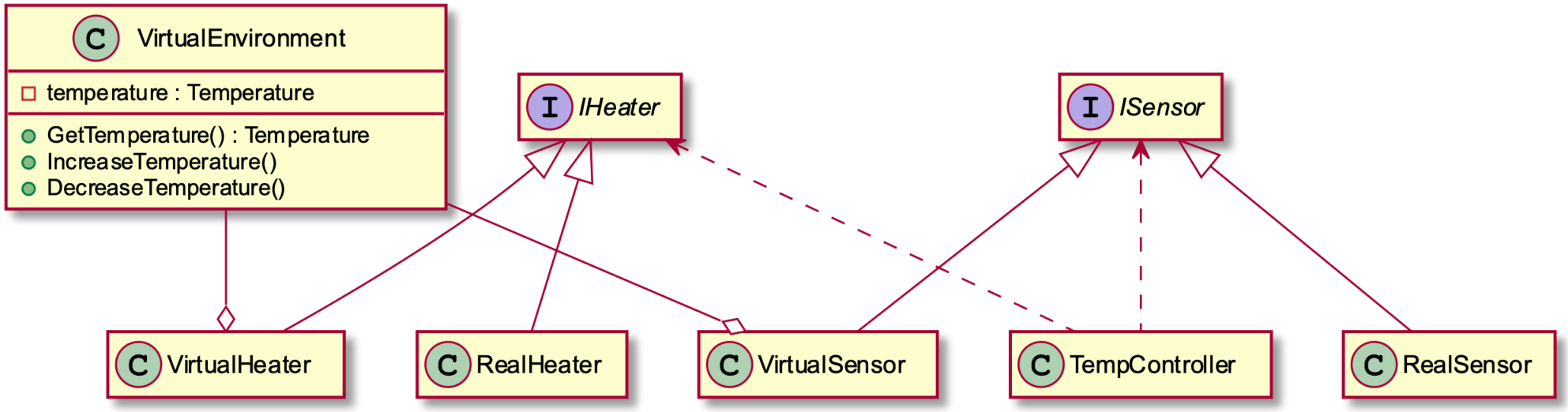
Ответим на несколько вопросов:

2. Где же гарантия, что мы подаем на вход парные значения?

Обновляем решение задачи

Ответим на несколько вопросов:

2. Где же гарантия, что мы подаем на вход парные значения?

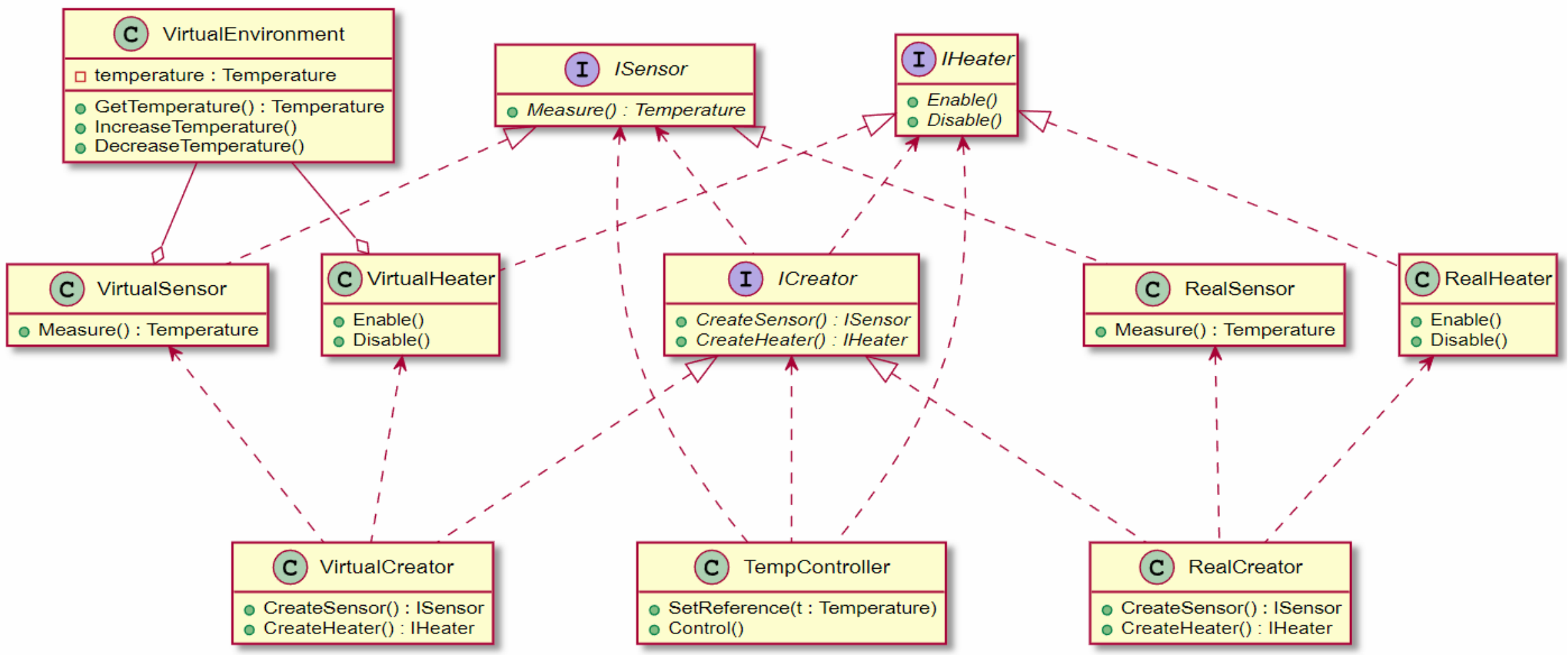


Обновляем решение задачи

Ответим на несколько вопросов:

3. А как гарантировать, что мы подаем на вход именно пару «виртуальный + виртуальный»?
4. И как спрятать реализацию **Environment** – тестовой среды окружения?

Обновляем решение задачи



Результаты

- Мы понимаем, как работает терморегулятор.
- Мы уверены, что он правильно будет работать со всеми датчиками.
- Мы знаем, что он будет правильно выполнять свою задачу.
- Мы даже знаем, как будем выполнять тестирование и калибровку готового устройства.

Эффект «WOW» или учимся думать головой

Как Вы можете заметить, мы уже решили поставленную задачу.

При этом, мы еще не написали **ни одной строчки кода!**

Осталось все только воплотить в жизни!

Как показывает практика, большинство подобных задач, не говоря о значительно более сложных и комплексных задачах, можно решить **еще на этапе проектирования!**

Почему так важен этап проектирования?

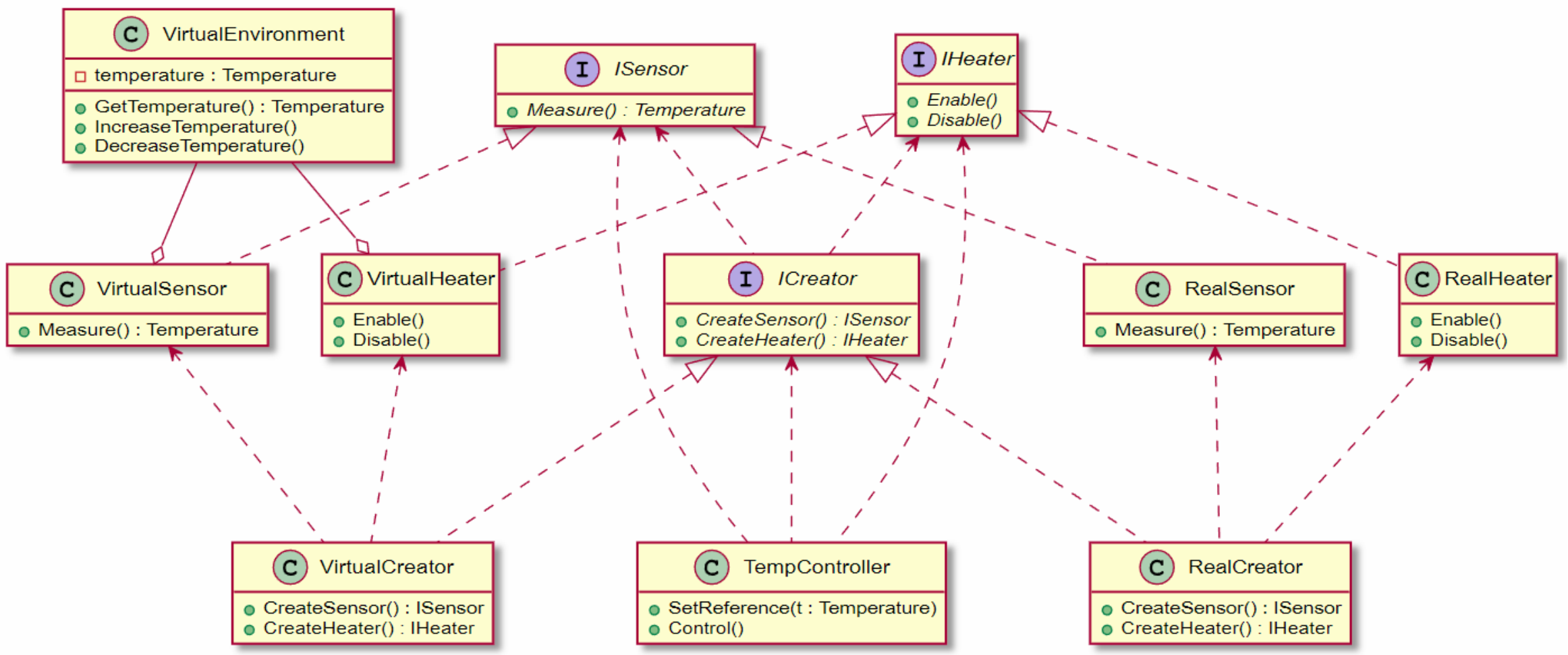
Проектирование необходимо, чтобы бороться со сложностью!

Главной целью проектирования является определение внутренних свойств системы и детализации её внешних свойств на уровне требований к ПО.

Проектирование на примере классов

- **На уровне анализа** класс содержит в себе только приблизительное описание системы и работает как логическая концепция предметной области или программного продукта.
- **На уровне проектирования** класс отражает основные проектные решения касательно распределения информации и планируемой функциональности, объединяя в себе сведения о состоянии и операциях.
- **На уровне реализации** класс дорабатывается до вида, в котором его можно использовать для решения задачи на языке программирования.

Вернемся к задаче о терморегуляторе



Абстрактная фабрика. Коротко.

Абстрактная фабрика — это такой шаблон, который часто используется как конкретное решение, когда необходимо создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.

Come with me, if you want to ..!

Что такое шаблоны проектирования?

Шаблон проектирования — это часто встречающееся решение определённой проблемы при проектировании архитектуры программ.

В отличие от готовых функций или библиотек, шаблон **нельзя просто взять и скопировать в программу**.

Паттерн представляет собой **общую концепцию решения** определенной проблемы, которую необходимо адаптировать под задачи программы.

Паттерн VS Алгоритм

Паттерны часто путают с алгоритмами, т.к. оба понятия описывают типовые решения известных проблем.

- **Алгоритм** — это **чёткий набор действий.**
- **Паттерн** — это **высокоуровневое описание решения**, реализация которого может отличаться в двух разных программах.

Следует помнить:

Если паттерн применять **неправильно** или к **неподходящей задаче**, то он может принести множество проблем. Но **правильно** примененный паттерн поможет решить задачу **легко** и **просто**.

Из чего состоят шаблоны?

Описания паттернов очень формальны и состоят из:

- **Проблема**, которую решает паттерн;
- **Мотивации к решению** проблемы способом, который предлагает паттерн;
- **Структуры классов**, составляющих решение;
- **Пример** на одном из языков программирования;
- **Особенности реализации** в различных контекстах;
- **Связи** с другими паттернами.

Паттерны – это больно. Зачем?

- **Проверенные решения.** Вы тратите меньше времени, используя готовые решения, вместо повторного «изобретения велосипеда». До некоторых решений вы смогли бы додуматься и сами, но многие могут быть для вас открытием.
- **Стандартизация кода.** Вы сделаете меньше просчётов при проектировании, используя типовые унифицированные решения, так как все скрытые проблемы в них уже давно найдены и решены.
- **Общая терминология.** Вы произносите название паттерна, вместо того, чтобы час объяснять другим программистам, что вы придумали и какие классы для этого нужны.

Классификация паттернов

Паттерны отличаются по **уровню сложности**, **детализации** и **охвату проектируемой системы**.

Самые **низкоуровневые** и **простые** паттерны — ***идиомы***.

Они не универсальны, поскольку применимы только в рамках одного конкретного языка программирования.

Самые **универсальные** — ***архитектурные паттерны***, которые можно реализовать практически на любом языке. Они нужны для проектирования **всей программы**, а не отдельных её элементов.

Классификация по предназначению

- **Fundamental** (Фундаментальные)
- **Creational** (Порождающие)
- **Structural** (Структурные)
- **Behavioral** (Поведенческие)

- **Anti-patterns** (Анти-паттерны)

Классификация по предназначению (основные)

- **Порождающие паттерны (Creational)** решают задачу гибкого создания объектов без внесения в программу лишних зависимостей, т.о. предоставляют механизмы инициализации, позволяя создавать объекты удобным способом.
- **Структурные паттерны (Structural)** определяют отношения между классами и объектами, позволяя им работать совместно.
- **Поведенческие паттерны (Behavioral)** заботятся об эффективной коммуникации между объектами.

Somebody, please, get this man the ~~gun~~ pattern!

Шаблоны проектирования

Паттерны создания объектов

Creational паттерны работают с механизмами создания объектов.

- **Singleton** (Одиночка) - обеспечивает существование в системе ровно одного экземпляра некоторого класса;
- **Factory Method** (Фабрика) - делегирует процесс создания объектов классам-наследникам;
- **Abstract Factory** (Абстрактная фабрика) - описывает сущность для создания целых семейств взаимосвязанных объектов;
- **Prototype** (Прототип) - клонирует объекты на основании некоторого базового объекта;
- **Builder** (Строитель) - отделяет процесс создания комплексного объекта от его представления.

Структурные паттерны

Структурные паттерны (structural) описывают создание более сложных объектов, либо упрощают работу с другими объектами системы.

- **Adapter** (Адаптер) - на основании некоторого класса создает необходимый клиенту интерфейс;
- **Facade** (Фасад) - описывает унифицированный интерфейс для облегчения работы с набором подсистем;
- **Composite** (Компоновщик) - работает с базовыми и составными объектами единым образом;

Структурные паттерны

Структурные паттерны (structural) описывают создание более сложных объектов, либо упрощают работу с другими объектами системы.

- **Decorator** (Декоратор) - динамически добавляет новую функциональность некоторому объекту, сохраняя его интерфейс;
- **Proxy** (Заместитель) - создает объект, который перехватывает вызовы к другому объекту;
- **Bridge** (Мост) - разделяет абстракцию от интерфейса, позволяя им меняться независимо;
- **Flyweight** (Легковес) - эффективно работает с огромным количеством схожих объектов.

Поведенческие паттерны проектирования

Поведенческие шаблоны (behavioral) определяют эффективные способы взаимодействия различных объектов в системе.

- **Strategy** (Стратегия) - описывает набор взаимозаменяемых алгоритмов с единым интерфейсом;
- **Iterator** (Итератор) - обеспечивает доступ к коллекциям объектов без раскрытия внутреннего устройства этих коллекций;
- **Observer** (Наблюдатель) - создает объект для отслеживания изменений в подсистеме и нотификации других подсистем;
- **Memento** (Хранитель) - сохраняет внутреннее состояние объекта для последующего использования без нарушения инкапсуляции;

Поведенческие паттерны проектирования

Поведенческие шаблоны (behavioral) определяют эффективные способы взаимодействия различных объектов в системе.

- **Command** (Команда) - описывает объект, представляющий собой некоторое действие, которое можно выполнить в необходимый момент;
- **Interpreter** (Интерпретатор) - определяет способ вычисления выражений некоторого языка;
- **Mediator** (Посредник) - создает объект, которые регулирует взаимодействие между набором подсистем;
- **State** (Состояние) - позволяет объекту менять свое поведение при изменении его внутреннего состояния;

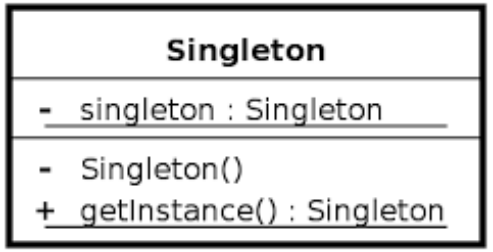
Поведенческие паттерны проектирования

Поведенческие шаблоны (behavioral) определяют эффективные способы взаимодействия различных объектов в системе.

- **Template method** (Шаблонный метод) - описывает алгоритм, возлагая реализацию некоторых частей алгоритма на подклассы;
- **Visitor** (Посетитель) - отделяет алгоритм от структуры, с которыми алгоритм работает;
- **Chain of responsibility** (Цепочка обязанностей) - пропускает некоторый запрос через набор обработчиков событий, до тех пор пока запрос не будет обработан.

Вы готовы «поднять» паттерны?

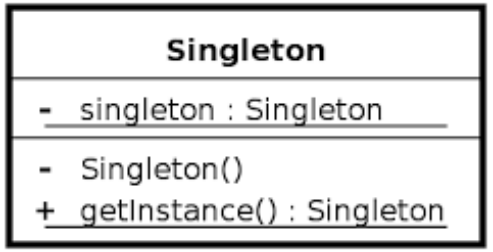
Creational: Singleton



Особенности:

1. Гарантирует наличие единственного экземпляра класса.
2. Предоставляет глобальную точку доступа.
3. Реализует отложенную инициализацию объекта-одиночки.

Creational: Singleton



Недостатки:

1. Нарушение принципа Single Responsibility (класс отвечает и за некоторый процесс, и за собственное время жизни);
   - Проблемы с расширяемостью;
   - Копирование кода при создании других синглтонов;
2. Скрытые зависимости (по сигнатуре функции не понятно, что она использует синглтон);
3. Проблемы тестируемостью (результаты тестов могут зависеть от порядка их выполнения).

Creational: Singleton – Применение

1. Когда в программе должен быть единственный экземпляр какого-то класса, доступный всем клиентам (например, общий доступ к базе данных из разных частей программы).

Singleton скрывает от клиентов все способы создания нового объекта, кроме специального метода. Этот метод либо создаёт объект, либо отдаёт существующий объект, если он уже был создан.

Creational: Singleton – Применение

2. Когда необходимо иметь больше контроля над глобальными переменными.

В отличие от глобальных переменных, **Singleton** гарантирует, что никакой другой код не заменит созданный экземпляр класса, поэтому вы всегда уверены в наличии лишь одного объекта-одиночки.

Тем не менее, в любой момент можно расширить это ограничение и позволить любое количество объектов-одиночек, поменяв код в одном месте (**метод getInstance**).

!!! Метод getInstance() создает только один экземпляр класса.

Creational: Singleton – Применение

Часто этот шаблон применяется при работе с конфигурацией программы или системы, с которой Вы работаете.

- Конфигурацию программ удобно хранить в файлах.
- Создадим текстовый файл **«props.txt»**, в котором будет храниться информация типа "ключ=значение"

Creational: Singleton – Применение

```
1
2  public final class Singleton {
3      private static Singleton _instance = null;
4
5      private Singleton() {}
6
7      public static synchronized Singleton getInstance() {
8          if (_instance == null)
9              _instance = new Singleton();
10         return _instance;
11     }
12 }
13
14
```

Creational: Singleton – Применение

```
17  public class Configuration {
18      private static Configuration _instance = null;
19
20      private Properties props = null;
21
22      private Configuration() {
23          props = new Properties();
24          try {
25              FileInputStream fis = new FileInputStream(new File("props.txt"));
26              props.load(fis);
27          }
28          catch (Exception e) {
29              // обработайте ошибку чтения конфигурации
30          }
31      }
32  }
33
```

Creational: Singleton – Применение

```
// Синглтон
public synchronized static Configuration getInstance() {
    if (_instance == null)
        _instance = new Configuration();
    return _instance;
}

// получить значение свойства по имени
public synchronized String getProperty(String key) {
    String value = null;
    if (props.containsKey(key))
        value = (String) props.get(key);
    else {
        // сообщите о том, что свойство не найдено
    }
    return value;
}
```

Creational: Singleton – Применение

```
54  // теперь для работы с конфигурацией можно использовать конструкцию вида:
55  String propValue = Configuration.getInstance().getProperty(propKey)
```

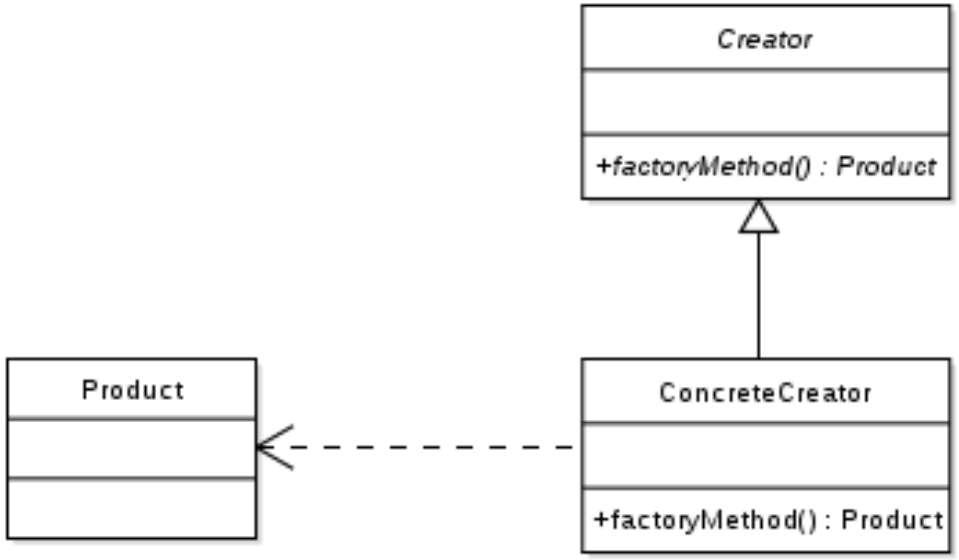
Задача

Вы пишете кросс-платформенное ПО, например, музыкальный плеер, который должен уметь работать на OS Windows, Linus и Mac OS.

Как Вы будете решать эту задачу?

Creational: Factory Method

Фабрика — это порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.



Creational: Factory Method

Преимущества:

- Избавляет класс от привязки к конкретным классам продуктов.
- Выделяет код производства продуктов в одно место, упрощая поддержку кода.
- Упрощает добавление новых продуктов в программу.
- Реализует *принцип открытости/закрытости*.

Недостатки:

- Может привести к созданию больших **параллельных иерархий классов**, так как для каждого класса продукта необходимо создать свой подкласс создателя.

Creational: Factory Method - Применение

```
2  class Factory {
3      public OS getCurrentOS(String inputos) {
4          OS os = null;
5          if (inputos.equals("windows")) {
6              os = new windowsOS();
7          } else if (inputos.equals("linux")) {
8              os = new linuxOS();
9          } else if (inputos.equals("mac")) {
10             os = new macOS();
11         }
12         return os;
13     }
14 }
15
```

Creational: Factory Method - Применение

```
16 interface OS {
17     void getOS();
18 }
19
20 class windowsOS implements OS {
21     public void getOS () {
22         System.out.println("применить для Windows");
23     }
24 }
25
26 class linuxOS implements OS {
27     public void getOS () {
28         System.out.println("применить для Linux");
29     }
30 }
31
32 class macOS implements OS {
33     public void getOS () {
34         System.out.println("применить для MacOS");
35     }
36 }
```

Creational: Factory Method - Применение

```
38 // тест Фабрики
39 public class FactoryTest {
40     public static void main(String[] args){
41         String input = "Linux";
42         Factory factory = new Factory();
43         OS os = factory.getCurrentOS(input);
44         os.getOS();
45     }
46 }
47
```

Creational: Factory Method - Применение

1. Когда заранее неизвестны типы и зависимости объектов, с которыми должен работать ваш код.

Фабричный метод отделяет код производства продуктов от остального кода, который эти продукты использует.

2. Когда нужно дать возможность пользователям расширять части фреймворка или библиотеки.

Пользователи могут расширять классы вашего фреймворка через наследование. Но как сделать так, чтобы фреймворк создавал объекты из этих новых классов, а не из стандартных?

Creational: Factory Method - Применение

3. Когда нужно сэкономить системные ресурсы, повторно используя уже созданные объекты, вместо порождения новых.

Такая проблема обычно возникает при работе с тяжёлыми ресурсоёмкими объектами, такими, как подключение к базе данных, файловой системе и т. д.

Creational: Abstract Factory

Абстрактная фабрика — это порождающий паттерн проектирования, который позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.

Creational: Abstract Factory

Преимущества:

- Гарантирует сочетаемость создаваемых продуктов.
- Избавляет клиентский код от привязки к конкретным классам продуктов.
- Выделяет код производства продуктов в одно место, упрощая поддержку кода.
- Упрощает добавление новых продуктов в программу.
- Реализует *принцип открытости/закрытости*.

Недостатки:

- Усложняет код программы из-за введения множества дополнительных классов.
- Требует наличия всех типов продуктов в каждой вариации.

Кто хочет купить автомобиль?

```
2  interface Lada {
3      long getLadaPrice();
4  }
5
6  interface Ferrari {
7      long getFerrariPrice();
8  }
9
10 interface Porshe {
11     long getPorshePrice();
12 }
13
14 interface InteAbsFactory {
15     Lada getLada();
16     Ferrari getFerrari();
17     Porshe getPorshe();
18 }
19
20
```

Creational: Abstract Factory – Применение

```
23 // Первая Фабрика
24 class ByLadaImpl implements Lada {
25     public long getLadaPrice() {
26         return 1000;
27     }
28 }
29
30 class ByFerrariImpl implements Ferrari {
31     public long getFerrariPrice() {
32         return 3000;
33     }
34 }
35
36 class ByPorsheImpl implements Porshe {
37     public long getPorshePrice() {
38         return 2000;
39     }
40 }
41
```

Creational: Abstract Factory – Применение

```
44 class ByCarPriceAbsFactory implements InteAbsFactory {
45     public Lada getLada() {
46         return new ByLadaImpl();
47     }
48     public Ferrari getFerrari() {
49         return new ByFerrariImpl();
50     }
51     public Porshe getPorshe() {
52         return new ByPorsheImpl();
53     }
54 }
55
```

Creational: Abstract Factory – Применение

```
59 // Вторая Фабрика
60 class RuLadaImpl implements Lada {
61     public long getLadaPrice() {
62         return 10000;
63     }
64 }
65
66 class RuFerrariImpl implements Ferrari {
67     public long getFerrariPrice() {
68         return 30000;
69     }
70 }
71
72 class RuPorsheImpl implements Porshe {
73     public long getPorshePrice() {
74         return 20000;
75     }
76 }
77
78
```

Creational: Abstract Factory – Применение

```
82 class RuCarPriceAbsFactory implements InteAbsFactory {
83     public Lada getLada() {
84         return new RuLadaImpl();
85     }
86     public Ferrari getFerrari() {
87         return new RuFerrariImpl();
88     }
89     public Porshe getPorshe() {
90         return new RuPorsheImpl();
91     }
92 }
93
94
```

Creational: Abstract Factory – Применение

```
97  // тест Абстрактной фабрики
98  public class AbstractFactoryTest {
99      public static void main(String[] args) {
100         String country = "BY";
101         InteAbsFactory ifactory = null;
102
103         if (country.equals("BY")) {
104             ifactory = new ByCarPriceAbsFactory();
105         } else if (country.equals("RU")) {
106             ifactory = new RuCarPriceAbsFactory();
107         }
108
109         Lada lada = ifactory.getLada();
110         System.out.println(lada.getLadaPrice());
111     }
112 }
113
114
```

Creational: Abstract Factory – Применение

1. Когда бизнес-логика программы должна работать с разными видами связанных друг с другом продуктов, не завися от конкретных классов продуктов.

Абстрактная фабрика скрывает от клиентского кода подробности того, как и какие конкретно объекты будут созданы. Но при этом клиентский код может работать со всеми типами создаваемых продуктов, поскольку их общий интерфейс был заранее определён.

Creational: Abstract Factory – Применение

2. Когда в программе уже используется Фабричный метод, но очередные изменения предполагают введение новых типов продуктов.

В хорошей программе каждый *класс отвечает только за одну вещь*. Если класс имеет слишком много фабричных методов, они способны затуманить его основную функцию. Поэтому имеет смысл вынести всю логику создания продуктов в отдельную иерархию классов, применив абстрактную фабрику.

Решим задачу

Вы – представители известного автоконцерна.

Ваша задача – производство автомобилей.

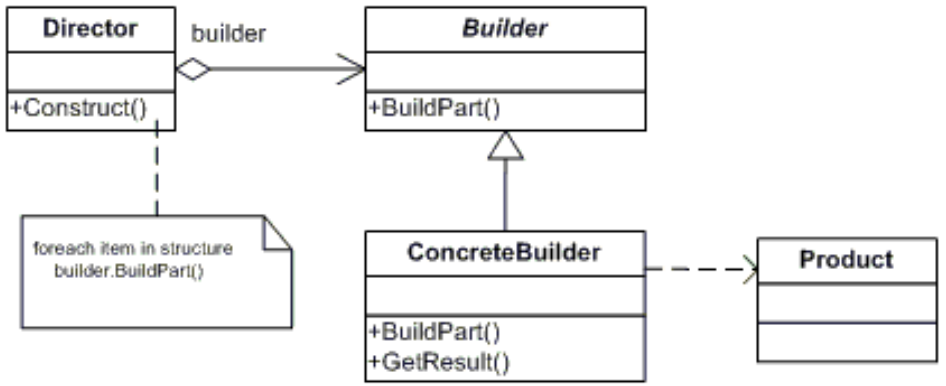
У концерна – целый ряд моделей автомобилей.

Каждый автомобиль нужно собрать.

Как вы будете решать эту задачу?!

Creational: Builder

Строитель — это порождающий паттерн проектирования, который позволяет создавать сложные объекты пошагово. Строитель даёт возможность использовать один и тот же код строительства для получения разных представлений объектов.



Creational: Builder

Преимущества:

1. Позволяет создавать продукты пошагово.
2. Позволяет использовать один и тот же код для создания различных продуктов.
3. Изолирует сложный код сборки продукта от его основной бизнес-логики.

Недостатки:

1. Усложняет код программы из-за введения дополнительных классов.
2. Клиент будет привязан к конкретным классам строителей, так как в интерфейсе строителя может не оказаться метода получения результата.

Creational: Builder – Применение

```
2  class Car {
3      public void buildBase() {
4          print("Собираем корпус");
5      }
6      public void buildWheels() {
7          print("Устанавливаем колеса");
8      }
9      public void buildEngine(Engine engine) {
10         print("Монтируем двигатель: " + engine.getEngineType());
11     }
12     private void print(String msg){
13         System.out.println(msg);
14     }
15 }
16
```

Creational: Builder – Применение

```
20 interface Engine {
21     String getEngineType();
22 }
23
24 class OneEngine implements Engine {
25     public String getEngineType() {
26         return "Бензиновый двигатель";
27     }
28 }
29
30 class TwoEngine implements Engine {
31     public String getEngineType() {
32         return "Дизельный двигатель";
33     }
34 }
35
```

Creational: Builder – Применение

```
36 abstract class Builder {
37     protected Car car;
38     public abstract Car buildCar();
39 }
40
```

Creational: Builder – Применение

```
41 class OneBuilderImpl extends Builder {
42     public OneBuilderImpl(){
43         car = new Car();
44     }
45     public Car buildCar() {
46         car.buildBase();
47         car.buildWheels();
48         Engine engine = new OneEngine();
49         car.buildEngine(engine);
50         return car;
51     }
52 }
53
```

Creational: Builder – Применение

```
54 class TwoBuilderImpl extends Builder {
55     public TwoBuilderImpl() {
56         car = new Car();
57     }
58     public Car buildCar() {
59         car.buildBase();
60         car.buildWheels();
61         Engine engine = new OneEngine();
62         car.buildEngine(engine);
63         car.buildWheels();
64         engine = new TwoEngine();
65         car.buildEngine(engine);
66         return car;
67     }
68 }
69
```

Creational: Builder – Применение

```
70  class Build {
71      private Builder builder;
72      public Build(int i){
73          if(i == 1) {
74              builder = new OneBuilderImpl();
75          } else if(i == 2) {
76              builder = new TwoBuilderImpl();
77          }
78      }
79      public Car buildCar(){
80          return builder.buildCar();
81      }
82  }
83
```

Creational: Builder – Применение

```
84 // тест Билдера
85 public class BuilderTest {
86     public static void main(String[] args) {
87         Build build = new Build(1);
88         build.buildCar();
89     }
90 }
```

Creational: Builder – Применение

1. Когда нужно избавиться от «телескопического конструктора».

Паттерн Строитель позволяет собирать объекты пошагово, вызывая только те шаги, которые вам нужны.

2. Когда код должен создавать разные представления какого-то объекта. Строитель можно применить, если создание нескольких представлений объекта состоит из одинаковых этапов, которые отличаются в деталях.

Creational: Builder – Применение

3. Когда нужно собирать сложные составные объекты, например, деревья Компоновщика.

Строитель конструирует объекты пошагово, а не за один проход. Более того, шаги строительства можно выполнять рекурсивно. А без этого не построить древовидную структуру, вроде Компоновщика.

Behavioral: Strategy

Стратегия — это поведенческий паттерн проектирования, который определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы можно взаимозаменять прямо во время исполнения программы.

Behavioral: Strategy

Преимущества:

- Горячая замена алгоритмов на лету.
- Изолирует код и данные алгоритмов от остальных классов.
- Уход от наследования к делегированию.
- Реализует *принцип открытости/закрытости*.

Недостатки:

- Усложняет программу за счёт дополнительных классов.
- Клиент должен знать, в чём состоит разница между стратегиями, чтобы выбрать подходящую.

Behavioral: Strategy – Применение

```
3  // Определяет ряд алгоритмов позволяя взаимодействовать между ними.
4  // Алгоритм стратегии может быть изменен во время выполнения программы.
5
6  interface Strategy {
7      void download(String file);
8  }
9
10 class DownloadWindownsStrategy implements Strategy {
11     public void download(String file) {
12         System.out.println("Windows download: " + file);
13     }
14 }
15
16 class DownloadLinuxStrategy implements Strategy {
17     public void download(String file) {
18         System.out.println("Linux download: " + file);
19     }
20 }
21
22
```

Behavioral: Strategy – Применение

```
23 class Context {
24     private Strategy strategy;
25     public Context(Strategy strategy){
26         this.strategy = strategy;
27     }
28 
29     public void download(String file){
30         strategy.download(file);
31     }
32 }
33 
34 
```

Behavioral: Strategy – Применение

```
35
36 // тест Стратегии
37 public class StrategyTest {
38     public static void main(String[] args) {
39         Context context = new Context(new DownloadWindownsStrategy());
40         context.download("file.txt");
41         context = new Context(new DownloadLinuxStrategy());
42         context.download("file.txt");
43     }
44 }
45
46
47
```

Behavioral: Strategy – Применение

1. Когда вам нужно использовать разные вариации какого-то алгоритма внутри одного объекта.

Стратегия позволяет варьировать поведение объекта во время выполнения программы, подставляя в него различные объекты-поведения (например, отличающиеся балансом скорости и потребления ресурсов).

2. Когда у вас есть множество похожих классов, отличающихся только некоторым поведением.

Стратегия позволяет вынести отличающееся поведение в отдельную иерархию классов, а затем свести первоначальные классы к одному, сделав поведение этого класса настраиваемым.

Behavioral: Strategy – Применение

3. Когда вы не хотите обнажать детали реализации алгоритмов для других классов.

Стратегия позволяет изолировать код, данные и зависимости алгоритмов от других объектов, скрыв эти детали внутри классов-стратегий.

4. Когда различные вариации алгоритмов реализованы в виде развесистого условного оператора. Каждая ветка такого оператора представляет собой вариацию алгоритма.

Стратегия помещает каждую лапу такого оператора в отдельный класс-стратегию. Затем контекст получает определённый объект-стратегию от клиента и делегирует ему работу. Если вдруг понадобится сменить алгоритм, в контекст можно подать другую стратегию.

Behavioral: Observer

Наблюдатель — это поведенческий паттерн проектирования, который создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах.

Behavioral: Observer

Преимущества:

- Издатели не зависят от конкретных классов подписчиков и наоборот.
- Вы можете подписывать и отписывать получателей на лету.
- Реализует *принцип открытости/закрытости*.

Недостатки:

- Подписчики оповещаются в случайном порядке.

Behavioral: Observer – Применение

```
3 // Позволяет одним объектам наблюдать за действиями,
4 // которые происходят в других объектах.
5 interface Observer {
6     void event(List<String> strings);
7 }
8
```

Behavioral: Observer – Применение

```
3  class University {
4      private List<Observer> observers = new ArrayList<Observer>();
5      private List<String> students = new ArrayList<String>();
6
7      public void addObserver(Observer observer){
8          observers.add(observer);
9      }
10
11     public void removeObserver(Observer observer) {
12         observers.remove(observer);
13     }
14
15     public void notifyObservers(){
16         for (Observer observer : observers) {
17             observer.event(students);
18         }
19     }
20 }
21
```

Behavioral: Observer – Применение

```
3  class University {
4      private List<Observer> observers = new ArrayList<Observer>();
5      private List<String> students = new ArrayList<String>();
6
7      // ...
8      public void addStudent(String name) {
9          students.add(name);
10         notifyObservers();
11     }
12
13     public void removeStudent(String name) {
14         students.remove(name);
15         notifyObservers();
16     }
17
18     // ...
19 }
20
```

Behavioral: Observer – Применение

```
28 class Director implements Observer {
29     public void event(List<String> strings) {
30         System.out.println("The list of students has changed: " + strings);
31     }
32 }
33
```

Behavioral: Observer – Применение

```
34 // тест Наблюдателя
35 public class ObserverTest {
36     public static void main(String[] args) {
37         University university = new University();
38         Director director = new Director();
39         university.addStudent("Igor Novoseltsev");
40         university.addObserver(director);
41         university.addStudent("Anna Karenina");
42         university.removeStudent("Igor Novoseltsev");
43     }
44 }
45
46
```

Behavioral: Observer – Применение

1. Когда после изменения состояния одного объекта требуется что-то сделать в других, но вы не знаете наперёд, какие именно объекты должны отреагировать.

Описанная проблема может возникнуть при разработке библиотек пользовательского интерфейса, когда вам надо дать возможность сторонним классам реагировать на клики по кнопкам.

Паттерн Наблюдатель позволяет любому объекту с интерфейсом подписчика зарегистрироваться на получение оповещений о событиях, происходящих в объектах-издателях.

Behavioral: Observer – Применение

2. Когда одни объекты должны наблюдать за другими, но только в определённых случаях.

Издатели ведут динамические списки. Все наблюдатели могут подписываться или отписываться от получения оповещений прямо во время выполнения программы.

Решим задачу

Мы разрабатываем некоторую финансовую систему.

Например, **интернет-банкинг**, которым будут пользоваться несколько банков-партнеров.

Задача: отображать баланс пользователя.

Как Вы будете решать эту задачу?

Structural: Adapter

Адаптер — это структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе. (Конвертер между двумя несовместимыми объектами.)

Structural: Adapter

Преимущества:

- Отделяет и скрывает от клиента подробности преобразования различных интерфейсов.

Недостатки:

- Усложняет код программы из-за введения дополнительных промежуточных классов.

Structural: Adapter – Применение

```
2  class PBank {
3      private int balance;
4      public PBank() { balance = 100; }
5      public void getBalance() {
6          System.out.println("PBank balance = " + balance);
7      }
8  }
9
10 class ABank {
11     private int balance;
12     public ABank() { balance = 200; }
13     public void getBalance() {
14         System.out.println("ABank balance = " + balance);
15     }
16 }
17
```

Structural: Adapter – Применение

```
20 class PBankAdapter extends PBank {
21     private ABank abank;
22     public PBankAdapter(ABank abank) {
23         this.abank = abank;
24     }
25     public void getBalance() {
26         abank.getBalance();
27     }
28 }
29
30
```

Structural: Adapter – Применение

```
32 // тест Адаптера
33 public class AdapterTest {
34     public static void main(String[] args) {
35         PBank pbank = new PBank();
36         pbank.getBalance();
37         PBankAdapter abank = new PBankAdapter(new ABank());
38         abank.getBalance();
39     }
40 }
41
42
43
```

Structural: Adapter – Применение

1. Когда вы хотите использовать сторонний класс, но его интерфейс не соответствует остальному коду приложения.

Адаптер позволяет создать объект-прокладку, который будет превращать вызовы приложения в формат, понятный стороннему классу.

2. Когда вам нужно использовать несколько существующих подклассов, но в них не хватает какой-то общей функциональности, причём расширить суперкласс вы не можете.

Вы могли бы создать ещё один уровень подклассов и добавить в них недостающую функциональность. Но при этом придётся дублировать один и тот же код в обеих ветках подклассов.

Анти-паттерны проектирования

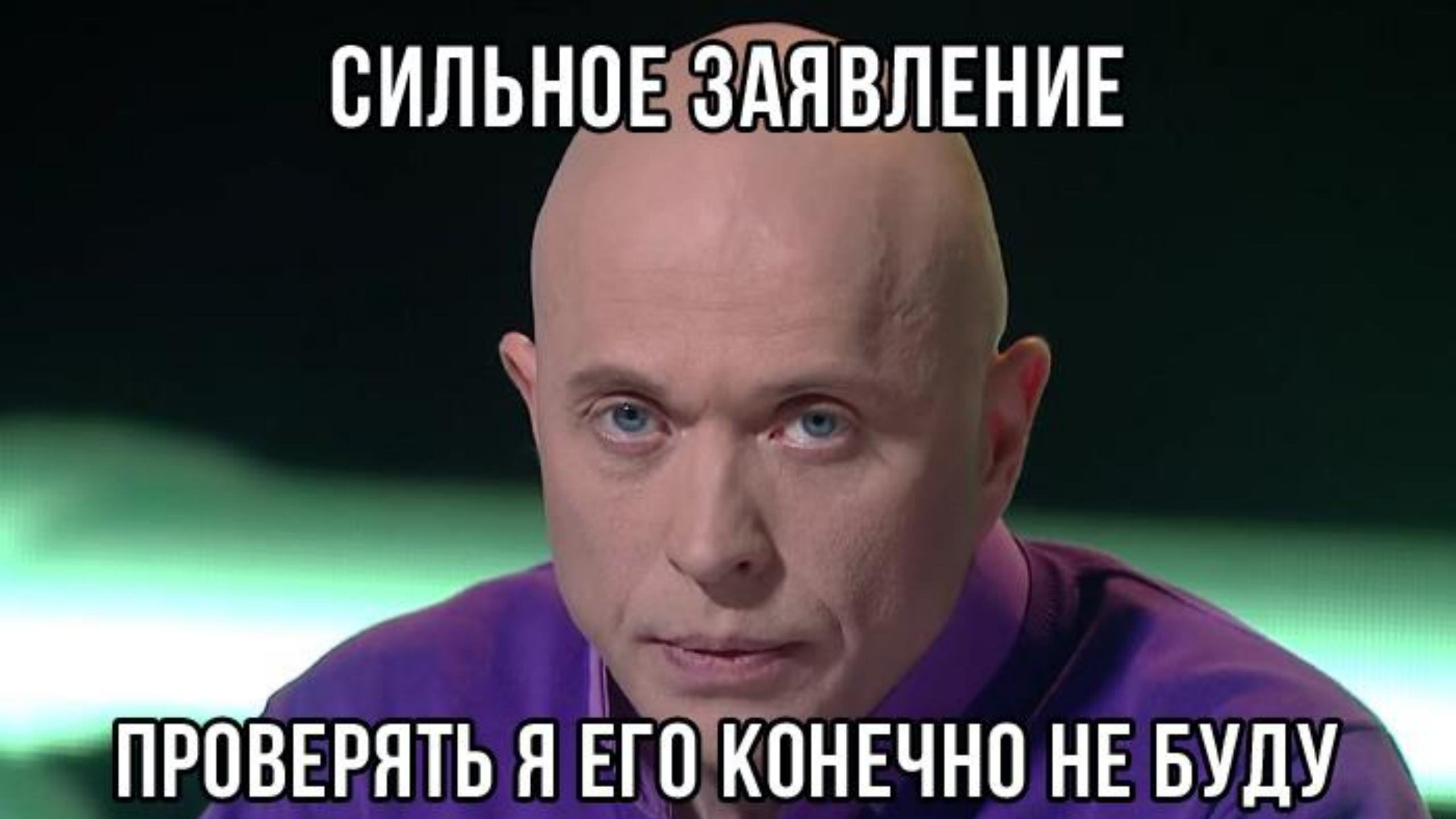
Считаю важным, что необходимо знать не только удачные способы решения задач, но и возможные ошибки при их решении.

- **Интерфейс для констант** - использует Java-интерфейс не по назначению;
- **Внесенная сложность** - добавляет ненужные сущности в иерархию классов.

Критика паттернов

Существует «авторитетное» мнение, что *«необходимость в некоторых паттернах вызвана недостатком конкретного языка программирования»*.

Может и так, но раз уж используемый нами язык не предоставляет нативный способ решения проблемы, то почему бы не воспользоваться идеей, которая уже кем-то проверена и является достаточно эффективной, если не оптимальной.


СИЛЬНОЕ ЗАЯВЛЕНИЕ
ПРОВЕРЯТЬ Я ЕГО КОНЕЧНО НЕ БУДУ

Критика паттернов

Неэффективные решения

- Паттерны пытаются стандартизировать подходы, которые и так уже широко используются.
- Эта стандартизация кажется некоторым людям догмой и они реализуют паттерны «как в книжке», не приспосабливая паттерны к реалиям проекта.

Неоправданное применение

- Похожая проблема возникает у новичков, которые только-только познакомились с паттернами.
- Вникнув в паттерны, человек пытается применить свои знания везде.

Достоинства шаблонов проектирования

Шаблоны проектирования это:

- готовые и надежные решения давно известных проблем;
- общая номенклатура в любом проекте (видите класс Фабрика и сразу понимаете как он работает).

Выводы из лекции

1. **Проектирование ПО до начала разработки – это важно!**
   - Проектирование помогает вам решить вашу задачу еще «на бумаге», т.е. на стадии обсуждения и принятия решения по развитию проекта и избежать огромных трудностей и затрат в будущем.

2. **Паттерны проектирования – решают конкретные задачи.**
   - Готовые решения известных проблем.
   - Эффективность и качество решений «шаблонных» задач.
   - Строгая логическая структура внутри программы.
   - Общая номенклатура (по названию класса легко можно определить его назначение).

«... I have no rival. No man can be my equal.
Take me to the future of your world…»

Queen. Princes of the Universe

Теперь для Вас нет ничего невозможного!

Важно понимать разницу!

- **Жизненный цикл разработки ПО**, в том числе и проектирование – это вопросы командной работы и всего, что с этим связано.

- **Паттерны проектирования** - это шаблонные решения, уже готовые инструменты, которые необходимы, чтобы помочь Вам решить конкретную задачу.

Осталась всего неделя лекций, а значит...
Скоро зачет :)

На сегодня все!
Увидимся на следующей неделе!