Лекция 9

Принципы программирования

- SOLID
- KISS
- YAGNI
- DRY

Принципы SOLID

- S – The Single Responsibility Principle
- O – The Open-Closed Principle
- L – The Liskov Substitution Principle
- I – Interface Segregation Principle
- D – The Dependency Inversion Principle

The Single Responsibility Principle

- Принцип единственной ответственности.
  - Любой сложный класс должен быть разбит на несколько простых составляющих, отвечающих за **определенный аспект поведения**, что упрощает как понимание, так и будущее развитие.

The Single Responsibility Principle

- Допустим, есть класс, который отвечает за вывод данных пользователю, обработку того, что ввел пользователь и сохранению данных в БД.
- Правильно иметь:
  - Класс для вывода данных
  - Класс для обработки данных
  - Класс для работы с БД

The Open-Closed Principle

- Принцип Открыт-Закрыт
  - программные сущности (классы, интерфейсы и т.п.) открыты для расширения, но закрыты для изменения. Это означает, что эти сущности могут менять свое поведение без изменения их исходного кода. Открытость сущностей означает возможность внесения изменений в поведении, путем изменения реализации или же путем переопределения поведения в наследниках.

The Open-Closed Principle

- Интерфейсы в модулях не меняем. Они один раз описаны, какие в них методы, что принимают, что возвращают.
- Требуется изменить поведение? Пишем новую реализацию для этих интерфейсов.

The Liskov Substitution Principle

- Принцип замещения Барбары Лисков
  - для корректной реализации отношения «ЯВЛЯЕТСЯ», наследник может ослаблять предусловие и усиливать постусловие (требовать меньше и гарантировать больше), при этом инварианты базового класса должны выполняться наследником.

The Liskov Substitution Principle

- Поведение наследуемых классов не должно противоречить поведению, заданному базовым классом.
- Логику работы наследника можно РАСШИРЯТЬ, но нельзя ПЕРЕОПРЕДЕЛЯТЬ логику, унаследованную от базового класса

Interface Segregation Principle

- Принцип разделения интерфейсов
  - что слишком «толстые» интерфейсы необходимо разделять на более маленькие и специфические, чтобы клиенты маленьких интерфейсов знали только о методах, которые необходимы им в работе.

Interface Segregation Principle

- Следование принципу ISP заключается в создании интерфейсов, которые достаточно специфичны и требуют только необходимый минимум реализаций методов. Избыточные интерфейсы, напротив, могут требовать от реализующего класса создание большого количества методов, причём даже таких, которые не имеют смысла в контексте класса

The Dependency Inversion Principle

- Принцип инверсии зависимостей
  - Модули верхнего уровня не должны зависеть от модулей нижнего уровня. И те и другие должны зависеть от абстракций.

The Dependency Inversion Principle

- При реализации высокоуровневых компонент вместо встраивания зависимостей от конкретных низкоуровневых классов формируется зависимость от абстракций (интерфейс, абстрактный класс, базовый класс). С последующей подстановкой (IoC) конкретных низкоуровневых реализаций.

KISS (Keep it short and simple)

- Если вы создаете простое приложение, которое будет использоваться локально на одной машине, без последующего усложнения архитектуры, то и не нужно создавать громоздкое решение с разбиением на несколько уровней и соблюдением жестко принципов программирования.

YAGNI (You aren't gonna need it)

- процесс и принцип проектирования ПО, при котором в качестве основной цели и/или ценности декларируется отказ от избыточной функциональности (отказ добавления функциональности, в которой нет непосредственной надобности).

DRY (Don’t repeat yourself)

- «Каждая часть знания должна иметь единственное, непротиворечивое и авторитетное представление в рамках системы»

Паттерны проектирования

- Паттернами проектирования – это решения часто встречающихся проблем в области разработки ПО.
- Паттерны **не являются готовыми решениями**, которые можно преобразовать в код, а представляют общее описание решения проблемы, которое можно использовать в различных ситуациях.

Паттерны проектирования

- Зачастую, паттерны представляют собой архитектурные решения для разрабатываемого приложения, либо его части.
- Паттерны могут реализовываться в проекте даже в тех местах, где их применение на данный момент является не очевидным, но позволит упростить дальнейшую разработку приложения.

Паттерны проектирования

- Типы паттернов:
  - **Структурные паттерны.**
  - **Порождающие паттерны.**
  - **Поведенческие паттерны.**

Структурные паттерны

- Структурные паттерны предназначены для выстраивания правильной иерархии классов (упрощение создания новых классов, расширение существующих) и правильного взаимодействия между классами (уменьшение взаимосвязанности и т.п.)

Структурные паттерны

- **Паттерн Adapter**
- Паттерн Bridge
- Паттерн Composite
- Паттерн Decorator
- Паттерн Facade
- Паттерн Flyweight
- Паттерн Proxy

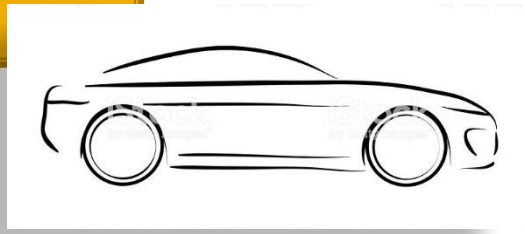
Паттерн Adapter

- Паттерн Adapter – это структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе.

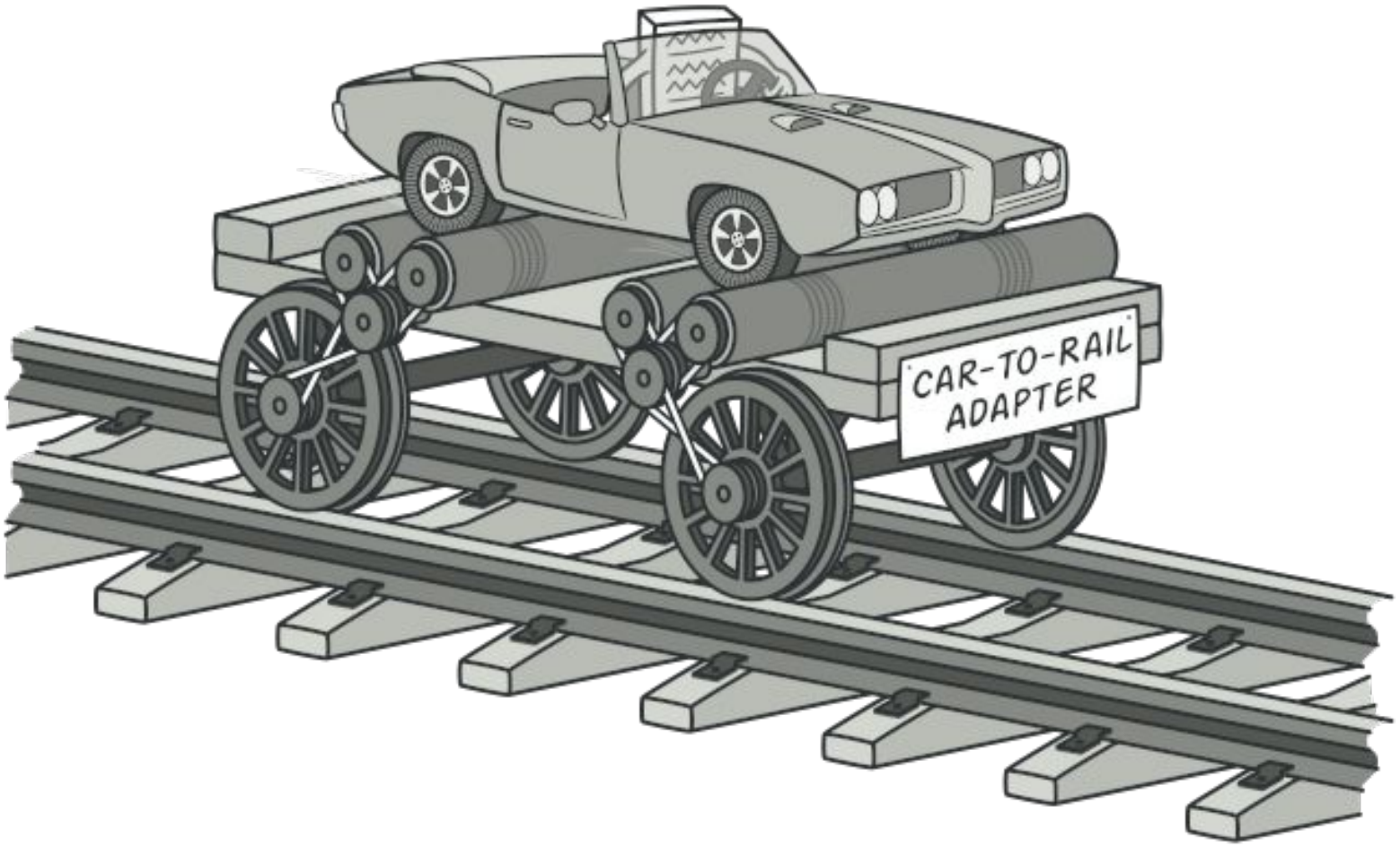
Паттерн Adapter

Двигаться по рельсам
Прикреплять вагоны
Гудеть в свисток

Двигаться по дорогам
Сигналить



Паттерн Adapter



Порождающие паттерны

- Порождающие паттерны проектирования предназначены для создания объектов, позволяя системе оставаться независимой как от самого процесса порождения, так и от типов порождаемых объектов.

Порождающие паттерны

- Паттерн Factory Method
- Паттерн Abstract Factory
- Паттерн Builder
- Паттерн Prototype
- **Паттерн Singleton**
- Паттерн Object Pool

Паттерн Singleton

- Паттерн Singleton контролирует создание единственного экземпляра некоторого класса и предоставляет доступ к нему.

Паттерн Singleton

- Singleton возлагает контроль над созданием единственного объекта на сам класс. Доступ к этому объекту осуществляется через **статическую** функцию-член класса, которая возвращает указатель или ссылку на него. Этот объект будет создан только при **первом** обращении к методу, а все последующие вызовы просто возвращают его адрес.

Паттерн Singleton

```
public class Singleton
{
    private static Singleton instance;

    private Singleton() { }

    public static Singleton getInstance()
    {
        if (instance == null)
        {
            instance = new Singleton();
        }
        return instance;
    }

}
```

```
static void Main(string[] args)
{
    #region Singleton
    Singleton s = new Singleton();

    s = Singleton.getInstance();
    #endregion
}
```

Паттерны поведения

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

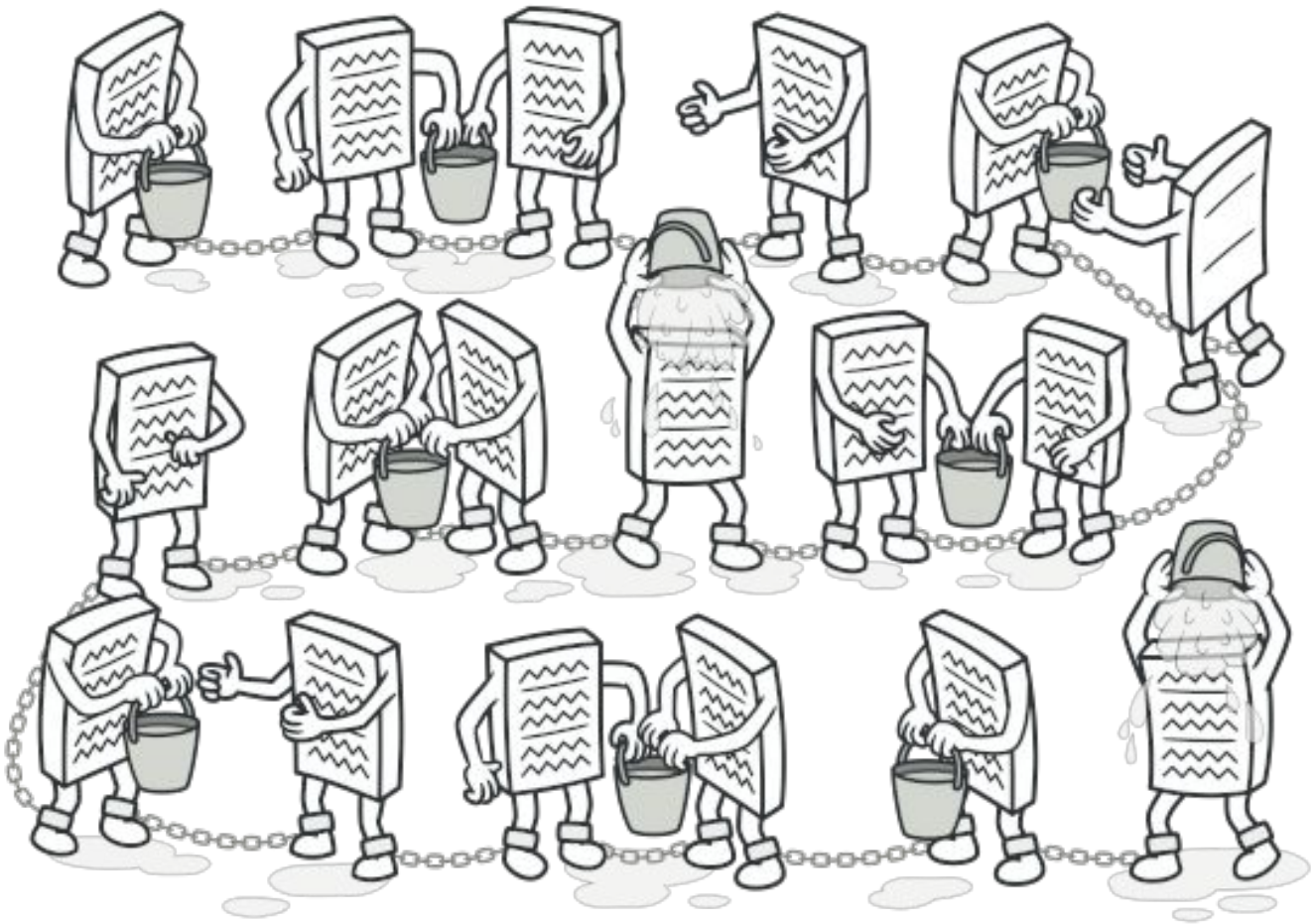
Паттерны поведения

- **Паттерн Chain of Responsibility**
- Паттерн Command
- Паттерн Iterator
- Паттерн Interpreter
- Паттерн Mediator
- Паттерн Memento
- Паттерн Observer
- Паттерн State
- паттерна Strategy
- Паттерн Template Method
- Паттерн Visitor

Паттерн Chain of Responsibility

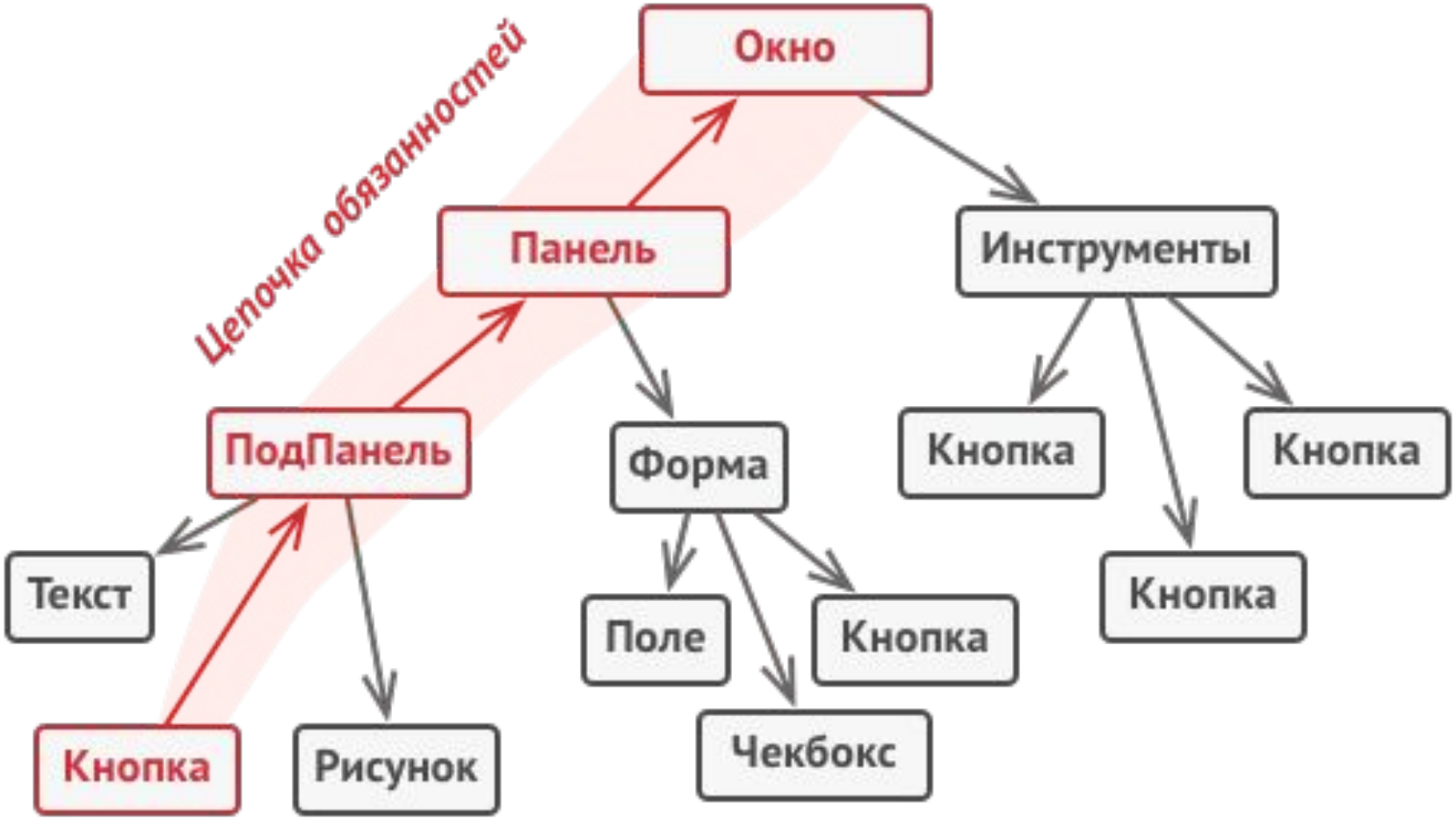
- Паттерн Chain of Responsibility – это поведенческий паттерн проектирования, который позволяет передавать запросы последовательно по цепочке обработчиков. Каждый последующий обработчик решает, может ли он обработать запрос сам и стоит ли передавать запрос дальше по цепи.

Паттерн Chain of Responsibility



Паттерн Chain of Responsibility

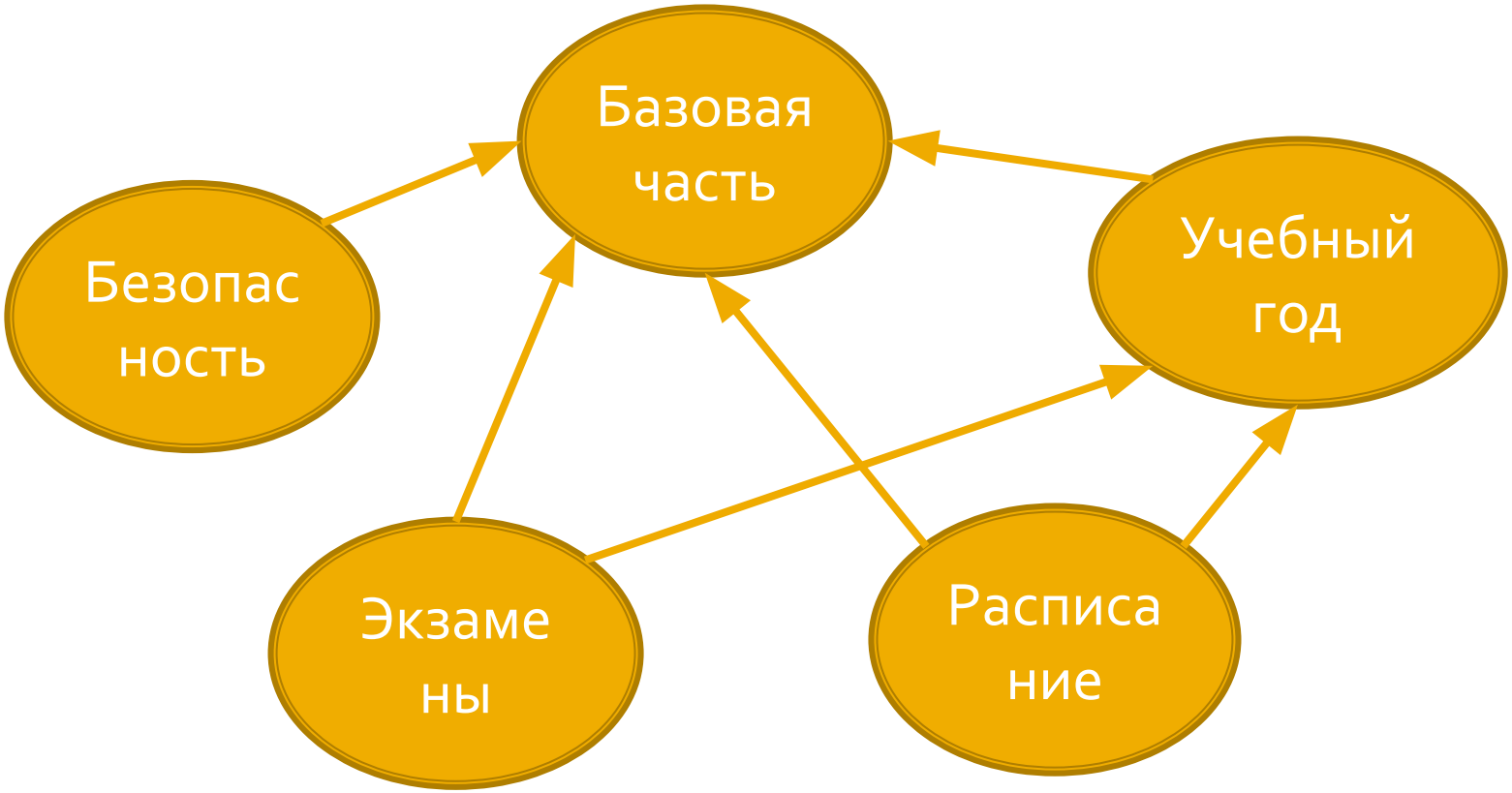
Паттерн Chain of Responsibility



Модули (подсистемы)

- В крупных проектах функционал разрабатываемого приложения часто разбивают на модули для упрощения ввода новых функций и поддержки существующих.
- В модулях выделяют внешние интерфейсы, через которые с ними можно взаимодействовать другим модулям.

Выделение модулей в системе кафедрального портала



Модули (подсистемы)

- Модули выделяют в отдельные проекты-библиотеки, чтобы их можно было просто подключать и использовать в других модулях и основной системе (исполняемых проектах).

Библиотеки

- Совокупность классов, интерфейсов и т.п., которые собираются в единый файл-библиотеку с расширением *.dll
- В дальнейшем эту библиотеку можно подключать в другие проекты и использовать классы и интерфейсы, описанные в ней*.

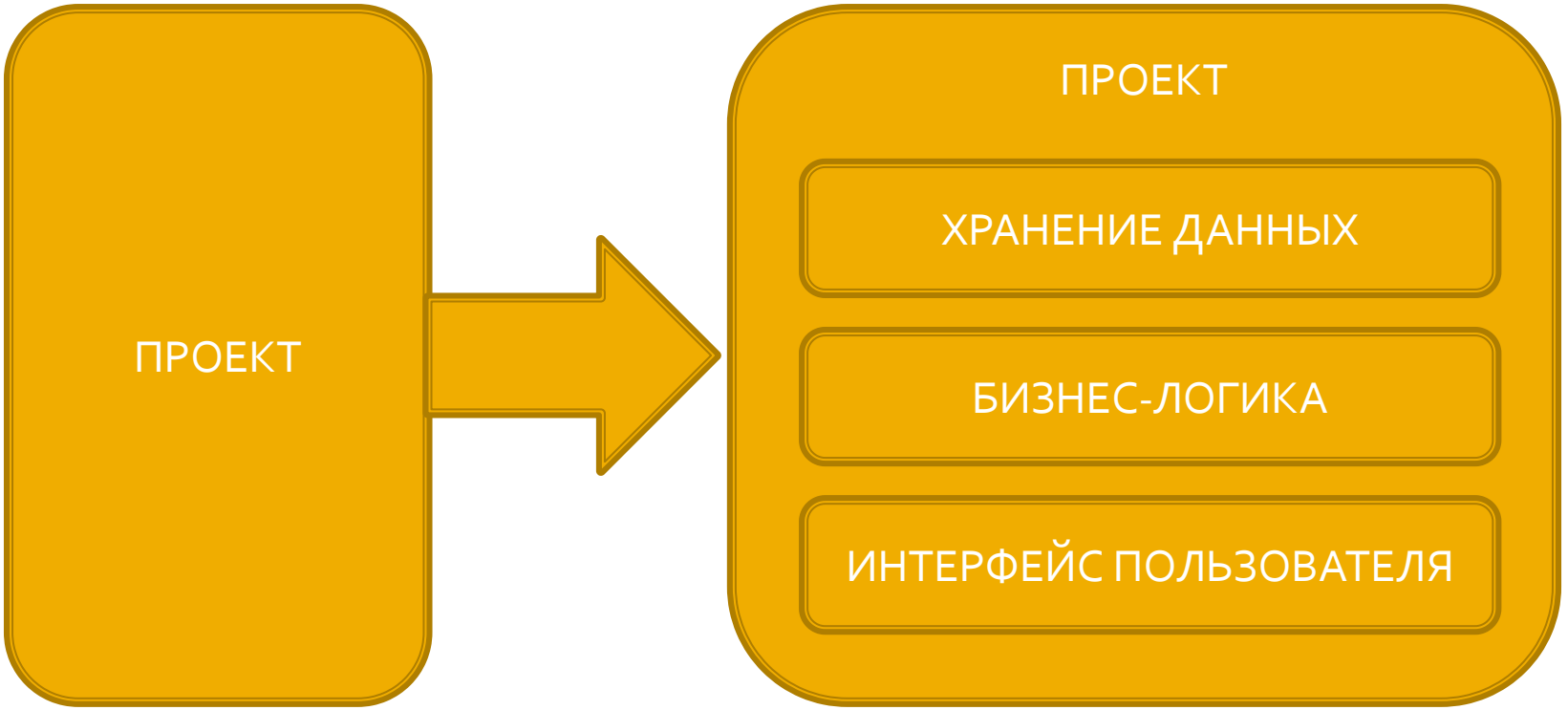
Модификатор internal

- класс и члены класса с подобным модификатором доступны из любого места кода в той же сборке, однако он недоступен для других программ и сборок.
- Класс будет доступен в библиотеке, но не будет доступен в других проектах, использующих эту библиотеку.

Библиотеки

- Файл с расширением *.dll не является исполняемым. Это значит, что его нельзя запустить на выполнение, как обычное приложение, типа десктопного.

Современные проекты



Все меняется

- IT среда очень быстро меняется, каждый день появляются новые инструменты обработки, хранения, передачи данных, новые технологии разработки ПО и т.д.
- Проект, который писался 2 года тому назад, сейчас будет считаться архаичным и жутко ресурсозатратным. Всегда будет вставать вопрос модернизации.

Концепция DAL

- Самым узким местом в данном случае является работа с хранилищем данных:
  - Появится иной способ хранения данных
  - Более дешевая и при этом качественная СУБД от другого производителя
- Для решения этой проблемы была придумана архитектура Data Access Layer

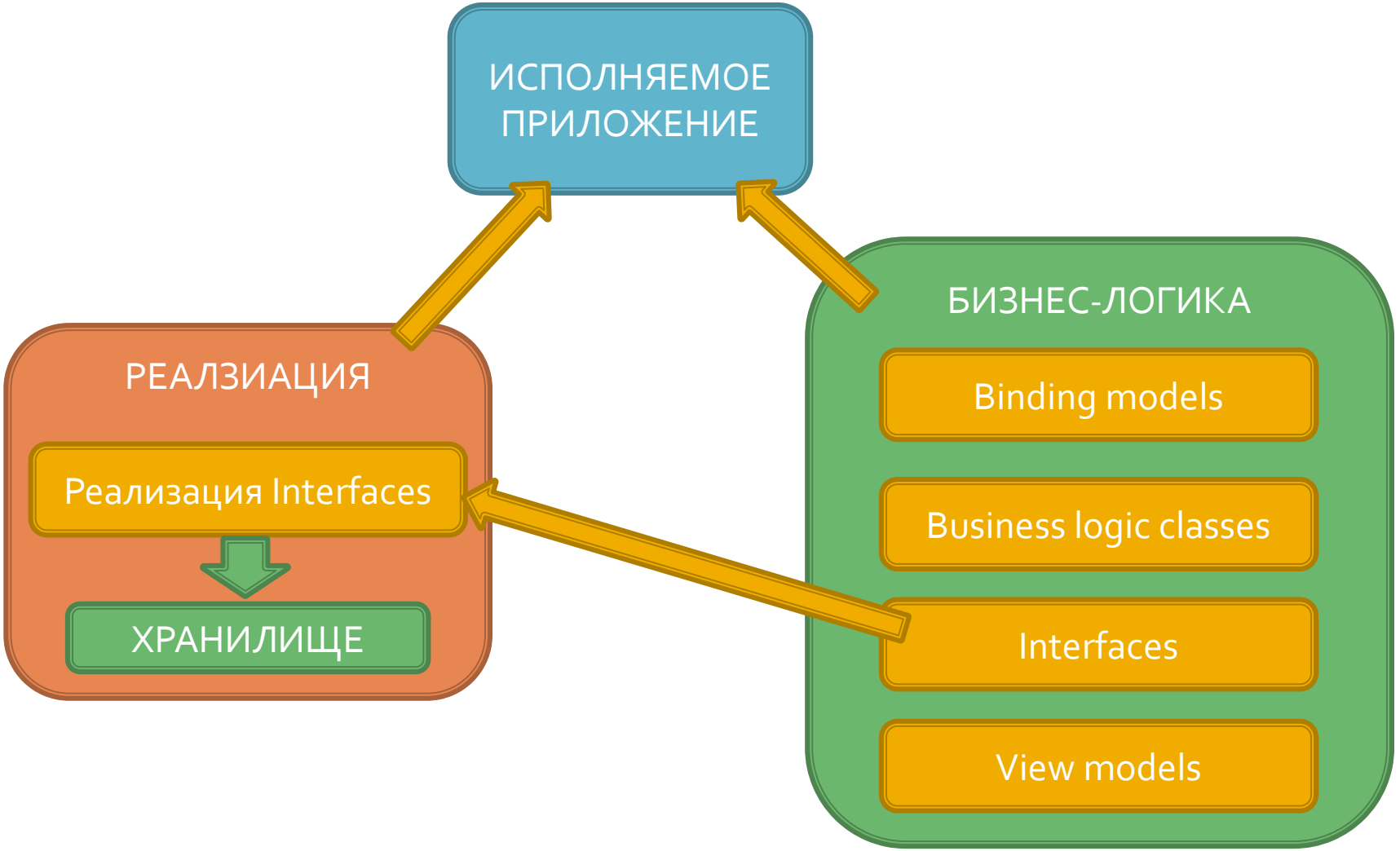
Концепция DAL

- Концепция обеспечивает возможность смены Хранилища данных без изменения бизнес-логики проекта.
- В основе концепции заложен принцип управляемой работы с данными, который определяет унифицированный и контролируемый способ доступа к различным данным для приложений.

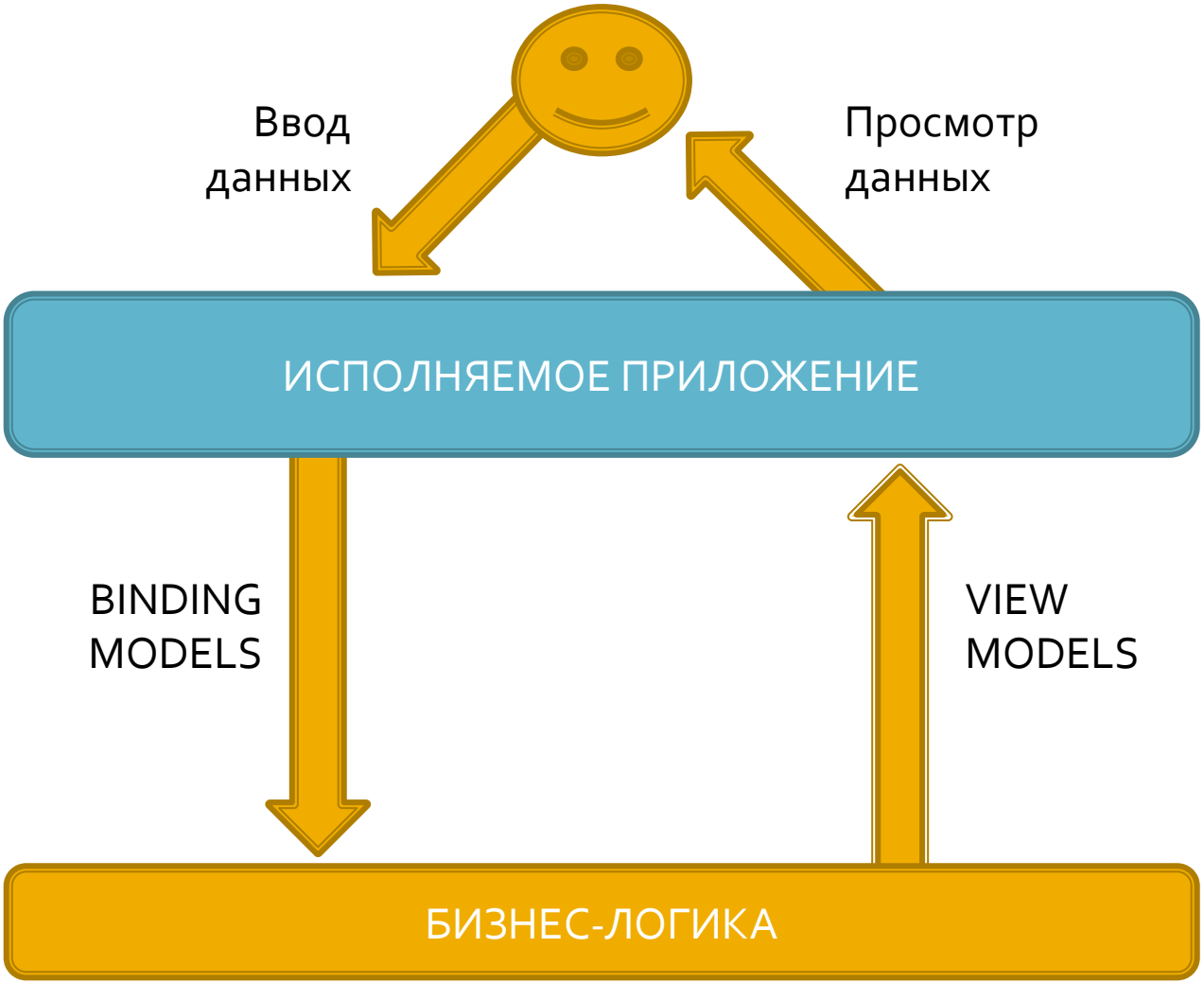
Концепция DAL

- Предлагается создавать интерфейсы, в которых описывать методы работы с данными в хранилище. При этом реализаций этих интерфейсов может быть множество, в зависимости от различных хранилищ данных, с которым предстоит работать.

Концепция DAL



Концепция DAL



Inversion of Control

- Инверсия управления – важный принцип ООП, используемый для уменьшения связанности классов. IoC – архитектурное решение интеграции, упрощающее расширение возможностей системы, при котором контроль над потоком управления программы остаётся за каркасом.

Dependency injection

- Внедрение зависимости – процесс предоставления внешней зависимости программному компоненту. Является специфичной формой «инверсии управления», когда она применяется к управлению зависимостями.

IoC-контейнер

- это библиотека, фреймворк, которая позволит упростить и автоматизировать написание кода с использованием данного подхода на столько, на сколько это возможно. В частности, позволяет реализовыватьDI.

IoC-контейнер

- Основная проблема – это new(). Для грамотного контроля зависимостей и тестирования, его нужно убрать.
- IoC (Inversion of Control) – это паттерн, в котором управление объектом (в частности – временем жизни объекта) поручено какой-то компоненте. Вместо создания объект самим (через new()) мы запрашиваем его у т.н. IoC-контейнера.

IoC-контейнер

- DI позволяет автоматически вытянуть из контейнера нужные зависимости при инициализации.

IoC-контейнер

- Ninject
- Unity
- Autofac
- Castle Windsor

IoC-контейнер

- 1. Настройка IoC-контейнера. Указание абстракций и какие реализации вместо них подставлять

```
private static IUnityContainer BuildUnityContainer()
{
    var currentContainer = new UnityContainer();
    currentContainer.RegisterType<IClientStorage, ClientStorage>(new HierarchicalLifetimeManager());
    currentContainer.RegisterType<IComponentStorage, ComponentStorage>(new HierarchicalLifetimeManager());
    currentContainer.RegisterType<IImplementerStorage, ImplementerStorage>(new HierarchicalLifetimeManager());
    currentContainer.RegisterType<IOrderStorage, OrderStorage>(new HierarchicalLifetimeManager());
    currentContainer.RegisterType<IProductStorage, ProductStorage>(new HierarchicalLifetimeManager());
    return currentContainer;
}
```

IoC-контейнер

- 2. Создание объектов в проекте через IoC-контейнер.

```
[STAThread]
Ссылок: 0
static void Main()
{
    var container = BuildUnityContainer();

    MailLogic.MailConfig(new MailConfig...);

    // создаем таймер
    var timer = new System.Threading.Timer(new TimerCallba

    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Application.Run(container.Resolve<FormMain>());
}
```