Лекция 20. Архитектура приложений

[**Цена ошибки**](#_bmvv84472vsa) **3**

[**Архитектура приложений**](#_g3clmu7if8bt) **4**

[OOP. Object Oriented Programming](#_cvr5bduw2xd8) 4

[Наследование](#_dxfhq285byci) 5

[Инкапсуляция](#_pcouknf8kpwd) 6

[Полиморфизм](#_8ef4byuc455y) 7

[Основные принципы, принятые в программировании](#_eb4ex3ijlu2z) 7

[KISS](#_2kns4j8zxtah) 7

[DRY](#_oq5jzbn6b91a) 8

[SOLID](#_a0ate57zqkjn) 11

[**Дополнительные инструменты**](#_dqpzgxgj44un) **13**

[Псевдокод](#_5g28xgkld0p2) 13

[MVC pattern](#_r1w9n9gz2du3) 13

[Модульность](#_tqus1ctx9vax) 14

[**Дополнительные материалы**](#_hqjpl9gij37k) **15**

[**Задание для закрепления материала**](#_nf51qpchsof4) **15**

# Цена ошибки

Про крупнейшие ошибки программистов и их стоимость можно почитать по ссылкам:

<https://habr.com/company/pvs-studio/blog/330762/>

<https://www.google.ru/search?newwindow=1&ei=vaRaW5GfIsmPmgXYxpSADA&q=%D1%86%D0%B5%D0%BD%D0%B0+%D0%BE%D1%88%D0%B8%D0%B1%D0%BA%D0%B8+%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%81%D1%82%D0%B0&oq=%D1%86%D0%B5%D0%BD%D0%B0+%D0%BE%D1%88%D0%B8%D0%B1%D0%BA%D0%B8+%D0%BF%D1%80%D0%BE%D0%B3%D1%80&gs_l=psy-ab.3.0.0j0i22i30k1.886607.892677.0.894507.17.16.0.1.1.0.130.1489.8j7.15.0....0...1c.1.64.psy-ab..1.16.1493...35i39k1j0i131k1j0i67k1.0.21DL2QaNn2Y>

Вообще, цену ошибки и ее исправления можно представить следующим образом:



# Архитектура приложений

Вы уже получили некоторый опыт модифицируя приложение калькулятора. Считаю это было достаточно показательно, насколько тяжело модифицировать код, который не спроектирован для расширения. Для того, чтобы код было легко понимать и модифицировать его архитектура должна быть хорошо продумана и спроектирована.

Для начала пройдемся по проблемам, которые я заметил:

1. Мало у кого была “входная точка” для приложения. Да, почти у всех был файл index или app, но никто не догадался, что приложение можно выделить в отдельный класс. Это нужно для того, чтобы управлять приложением из одной точки: создать калькулятор, очистить рабочую область, показать/скрыть журнал и тд. Это было бы типичным примером FrontController-а.
2. Все смешали “представление” (внешний вид) с бизнес-логикой. Почему так делать не стоит? При изменении внешнего вида, или отображения элементов не должна меняться логика самого приложения, т.е. как оно реагирует на действия пользователя, - это вносит эффект неожиданности даже при самых мелких изменениях. Приложение может сломаться из-за минимальных изменений.
3. Нарушен принцип единой ответственности, один огромный класс, который отвечает за все и сразу. Такой класс тяжело модифицировать, тяжело предсказать какие будут последствия изменений, тяжело читать код.

На самом деле ошибок при проектировании было много. Для того, чтобы не повторять эти ошибки, или избежать последствий в дальнейшем, к курсу была добавлена эта лекция.

## OOP. Object Oriented Programming

Как ясно из названия, этот принцип проектирования строит любое приложение на основе объектов, совсем не обязательно, реально существующих.

Например, для калькулятора реально существующими объектами будут:

* Цифровой блок (и каждая кнопка в нем)
* Поле с результатами расчетов
* Блок журнала (и каждая запись в нем)
* Поле с отображением строки формулы
* блок действий и кнопки в нем
* etc.

Несуществующими объектами могут быть:

* класс приложения (его нельзя соотнести с реально существующим объектом).
* класс, который передает набранный текст выражения для расчета

Классы, которые нельзя соотнести с реально существующим в реальности объектом называются Pure Fabrication (можно почитать про этот паттерн).

ООП базируется на 3х основных принципах: Наследование, Инкапсуляция, Полиморфизм.

### Наследование

Наследование — свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью. Класс, от которого производится наследование, называется базовым, родительским или суперклассом. Новый класс — потомком, наследником, дочерним или производным классом.

Что это значит? Это значит, что мы можем выделить общие черты для всех кнопок и поместить их в базовый класс:

**class BaseButton** {  
 constructor() {  
 // текст кнопки (он есть везде)

this.caption = ‘’;  
 // классы есть везде, или почти везде  
 this.classesList = [‘btn’];  
 }  
 // клик есть везде

clickBtn() {}  
  
 // создавать кнопки надо в любом случае  
 createDomElement(domElement) {  
 domElement.innerHtml = `<button class=”` + this.classesList.join(‘ ’) + `”>

` + this.caption + `

</button>`  
 }  
}

А для конкретного класса кнопок определить какие-то свои параметры:  
  
**class NumberButton extends BaseButton** {  
 constructor( number ) {  
 super(); // вызываем конструктор родителя

// на этот момент у нас уже созданы this.caption и this.classesList

this.classesList.push(‘number-btn’); // добавляем класс

this.caption = number; // добавляем надпись на кнопке

}  
  
 clickBtn() {  
 // сюда добавляем обработчик  
 }  
 // а вот createDomElement унаследуется от BaseButton, трогать его не будем, нас он пока устраивает  
}

таким же образом мы объявим ActionButton и MathActionButton, притом MathActionButton будет унаследован от ActionButton и может нести в себе черты любого из предков в цепочке.

**class ActionButton extends BaseButton** {

constructor(action) {

super(); // вызываем конструктор родителя

// на этот момент у нас уже созданы this.caption и this.classesList

this.classesList.push(‘btn-action’); // добавляем класс

this.caption = action; // добавляем надпись на кнопке

}

}

**class MathActionButton extends ActionButton** {

constructor(action) {

super( action ); // вызываем конструктор родителя

this.classesList.push(‘btn-math-action’); // добавляем класс

// на текущий момент в classesList следующие классы:

// btn-math-action, btn-action, btn

}

}

### Инкапсуляция

Считается, что человек способен удержать в голове не более 7 уровней абстракции, но этот расчет приведен для идеального программиста. На самом деле, этот показатель, для среднего программиста, не превышает 3-4.

Уровни абстракции можно представить как цепочку вызовов методов объектов, из одного объекта вызывается метод другого, в этом методе вызывается третий объект и т.д. Согласитесь, держать в уме что делает каждый объект и каждый метод, достаточно сложно. Инкапсуляция позволяет абстрагироваться от конкретных деталей реализации и мыслить в рамках текущего уровня абстракции. Мы знаем, что у кнопки есть: конструктор, метод рендера, и обработчик клика, этого нам достаточно. Объект не должен раскрывать детали реализации, - как это сделано, какие объекты подключаются в каждом вызове - на текущий момент это не важно.

**Инкапсуляция - это сокрытие деталей реализации.**

### Полиморфизм

Самое простое объяснение полиморфизма: Одна абстракция - множество реализаций.

В нашем примере с кнопками, BaseButton является абстракцией, которая описывает интерфейс (доступные извне методы):

* clickBtn
* createDomElement

В нашем случае, мы можем обращаться к потомкам этого класса одинаково (вне зависимости от внутренней реализации). Например:

class Calculator {

…

handleButtonClick( ButtonObject ) {

ButtonObject.clickBtn();

}

}

При условии, что ButtonObject является потомком BaseButton, мы уверены, что у него есть метод clickBtn. Как реализуется этот метод - нам не важно (инкапсуляция), важно, что мы уверены, что этот метод есть.

## Основные принципы, принятые в программировании

Как и в любой дисциплине, в программировании, со временем, сформировались свои правила и принципы, несоблюдение которых считается плохим тоном.

### KISS

Keep It Simple Stupid

Keep It Short and Simple

Не нужно ничего усложнять, сложная система сложна в поддержке, сложна для изменения и расширения. Чем проще вы спроектируете систему, тем проще будет вам и любому другому, кто возьмется за ее поддержку.

В программировании есть несколько неписаных правил:

1. Количество символов в строке не должно превышать 80 символов (это правило не строгое, но лучше этот лимит не превышать без необходимости ). В крайних случаях, допустим размер строки в 120 символов (это правило строгое)
2. Количество строк в методе не должно превышать 20, это правило не строгое, но это, своего рода, маркер, что метод спроектирован неправильно и его можно упростить.
3. Количество строк в классе не должно превышать 200, это правило не строгое, но это маркер неправильной архитектуры.

### DRY

Don’t Repeat Yourself

Не повторяйтесь!

Если какой-то кусок кода выполняется постоянно - вынесите его в отдельный метод/класс.

Самый банальный пример:

Экземпляр класса создается по всей системе, в разных местах, в разных классах.

При изменении самого класса или его конструктора вам необходимо найти и изменить все места, где экземпляры этого класса создаются. Никто не даст гарантий, что вы измените абсолютно все места, где экземпляры создаются, особенно, если вы не знаете систему целиком.

Решением этой проблемы может стать фабричный метод (паттерн проектирования), т.е. вы выносите код для создания объекта в отдельный метод и уже вызываете метод, а не пишете код каждый раз, когда необходимо создать экземпляр.

class Calculator {

…

createActionButton() {

return new ActionButton();

}

createNumberButton() {

return new NumberButton();

}

createMathActionButton() {

return new MathActionButton();

}

}

Хорошо, мы вынесли методы создания кнопок в отдельные фабричные методы, это упростило нам работу. Но, сейчас мы видим 3 однотипных метода которые похожи по функционалу, их можно объединить в отдельный класс:

class Calculator {

…

createActionButton() {

return ButtonFabric.createActionButton();

}

createNumberButton() {

return ButtonFabric.createNumberButton();

}

createMathActionButton() {

return ButtonFabric.createMathActionButton();

}

}

class ButtonFabric {

createActionButton() {

return new ActionButton();

}

createNumberButton() {

return new NumberButton();

}

createMathActionButton() {

return new MathActionButton();

}

}

В этом примере, мы вынеси весь функционал по созданию кнопок в отдельный класс. И слегка разгрузили класс Calculator. Класс фабрики сгруппировал все фабричные методы из калькулятора в себе. Таким образом, можно вынести вообще все методы по созданию объектов в один класс, получится глобальная фабрика. Но делать этого не рекомендуется по нескольким причинам:

* Класс фабрики получится просто огромным
* Сформируется огромный список зависимостей у такой фабрики
* Такую фабрику переиспользовать будет невозможно.

Рекомендуется использовать несколько фабрик, которые будут обслуживать отдельную группу объектов. Например, сейчас у нас получилась фабрика, отвечающая за создание только кнопок, а т.к. все кнопки у нас относятся к одном абстрактному типу (BaseButton) и имеют один интерфейс, наша фабрика называется абстрактной.

Но вернемся к нашему примеру:

в классе Calculator у нас очень много однотипных действий и работают они все по одному типу - это нарушение DRY принципа. Попробуем переоформить код таким образом, чтобы минимизировать повторения.

class Calculator {

…

// buttonType one of [‘action’, ‘number’, ‘math-action’]

createButton( buttonType ) {

return ButtonFabric.createButtonByType( buttonType );

}

}

class ButtonFabric {

createButtonByType( buttonType ) {

switch( buttonType ) {

case ‘number’:

return this.createNumberButton();

break;

case ‘action’:

return this.createNumberButton();

break;

case ‘math-action’:

return this.createMathActionButton();

}

}

createActionButton() {

return new ActionButton();

}

createNumberButton() {

return new NumberButton();

}

createMathActionButton() {

return new MathActionButton();

}

}

Отлично, мы сильно упростили класс Calculator и сделали метод создания кнопок параметризованным, но мы слегка усложнили ButtonFabric и у нас все равно остались повторяющиеся действия. попробуем упростить создание объектов и убрать повторяющийся код:

class ButtonFabric {

constructor() {

// тут мы сопоставили название типа и класс кнопки

this.buttonTypes = {

‘number’ : NumberButton,

‘action’ : ActionButton,

‘math-action’: MathActionButton

};

}

createButtonByType( buttonType ) {

if( this.buttonTypes.hasOwnProperty( buttonType ) ) {

return **new this.buttonTypes[ buttonType ]**;

}

// тут можно добавить обработчик ошибки

}

}

Вот теперь мы очень сильно упростили дальнейшую работу - для добавления нового типа кнопки нам нужно лишь добавить название типа и класса в ButtonFabric.buttonTypes и мы уже можем использовать новый тип в Calculator.createButton

Такие **последовательные** улучшения кода называются **Рефакторингом** и хорошо описаны в книге Мартина Фаулера “Рефакторинг”, книга настоятельно рекомендована к прочтению.

### SOLID

SOLID является аббревиатурой:

|  |  |  |
| --- | --- | --- |
| **Буква** | **Расшифровка** | **Объяснение** |
| S | Single Responsibility | Принцип единой ответственности. Класс должен реализовывать только одну задачу |
| O | Open-Closed Principle | Класс должен быть открыт для расширения. но закрыт для редактирования |
| L | The Liskov Substitution Principle | Принцип подстановки Барбары Лисков.  Наследующий класс должен дополнять, а не изменять базовый. |
| I | Interface Segregation | Принцип разделения интерфейсов. Лучше использовать много мелких интерфейсов, чем один универсальный |
| D | Dependency Inversion | Принцип инверсии зависимостей. Приложение должно строиться на основе Абстракций (Интерфейсов), а не конкретной реализации |

SOLID является одним из главнейших сводов правил в ООП. Этих принципов необходимо придерживаться при проектировании и разработке систем.

# Дополнительные инструменты

## Псевдокод

Для описания сложных механизмов или алгоритмов иногда полезно использовать псевдокод, перед написанием непосредственно кода. Псевдокод это неформализованное описание алгоритма, выполненная комментариями:

// получить список кнопок в калькуляторе

// для каждой кнопки в списке сделать следующее

// начало цикла

// добавить data-value атрибут

// добавить обработчик события onclick, который будет ссылаться на btnClick калькулятора, в качестве контекста передать текущий объект калькулятора

// конец цикла

Потом под каждой строкой комментария вставлять код на текущем языке программирования:

// получить список кнопок в калькуляторе

let buttonsList = document.querySelectorAll(‘.calc-button’);

// для каждой кнопки в списке сделать следующее

for(let i = 0; i < buttonsList.length; i++)

// начало цикла

{

// добавить data-value атрибут

buttonsList[i].setAttribute(‘data-value’, ...);

// добавить обработчик события onclick, который будет ссылаться на btnClick калькулятора, в качестве контекста передать текущий объект калькулятора

buttonsList[i].addEventListener(‘click’, this.btnClick.bind(this));

// конец цикла

}

## MVC pattern

Всегда старайтесь отделять бизнес-логику от отображения данных, это позволит избежать проблем поломки приложения при простом изменении внешнего вида.

Пример: заказчик просит изменить формат отображения данных на главной странице сайта с dd.mm.yyyy (который используется на всем сайте) на dd MMMM yyyy (этот формат будет использован только на главной странице). В этом случае необходимо будет изменять и конвертировать все даты только на одной странице, притом держать в уме, что на остальных страницах формат дат будет другой. Если еще происходит передача данных с главной страницы на остальные (например форма контактов), то необходимо конвертировать даты из одного формата в другой.

В любом случае, внутри системы следует использовать один формат дат (вообще, внутри системы лучше использовать объект Date), а выводить лучше в любом необходимом формате.

пример:

**let template = `<input type=”text” value=”` + formatDate(item.date, “dd MMMM YYYY”) + `” />`;**

В рамках фрэймворков это будет выглядеть в виде модификаторов вывода:  
**{{item.date | dateFormat : “dd MMMM YYYY”}}**

**MVC (Model View Controller)** пошел еще дальше, он предлагает разделить:  
- отображение данных (View)

- общение с пользователем (Controller)  
- обработку данных (Model)

это позволяет более гибко контролировать код.

На текущий момент, MVC патерн является одним из основных, используемых при разработке ПО.

## Модульность

Любые(или почти любые) задачи, которые вам когда-либо доведется решать и реализовывать, уже когда-то и кем-то были реализованы, не обязательно писать решение задачи, которое уже есть, посмотрите в списке пакетов NPM, возможно вы найдете подходящий модуль. Более того, писать свои собственные стоит таким образом, чтобы его потом можно было использовать повторно. Возвращаясь к задаче калькулятора:

Модульность приложения можно было бы реализовать следующим образом:

* Модуль обычной кнопки
* Модуль кнопки-действия
* Модуль кнопки-цифры
* Модуль блока-клавиатуры
* Модуль блока с кнопками-действиями

каждый из представленных модулей, можно использовать повторно, как отдельно, так и совместно. Притом **Модуль блока-клавиатуры** может использовать **Модуль кнопки-цифры**, который, в свою очередь, будет использовать **модуль обычной кнопки.**

Проблема со сторонними модулями лишь одна - вы не можете доверять издателю модуля. Вы не можете гарантировать работоспособность модуля при любых обстоятельствах, вы не можете гарантировать чистоту кода, вы не можете гарантировать полное покрытие тестами. Перед использованием модуля просмотрите код.

# Дополнительные материалы

Мартин Фаулер - Рефакторинг

Э. Хант‎ - Программист-прагматик. Путь от подмастерья к мастеру

# Задание для закрепления материала

В ближайшие несколько занятий, мы будем разрабатывать игру “Морской бой”. На текущий момент необходимо продумать архитектуру приложения. Сам код реализовывать не нужно. Использование псевдокода будет плюсом.