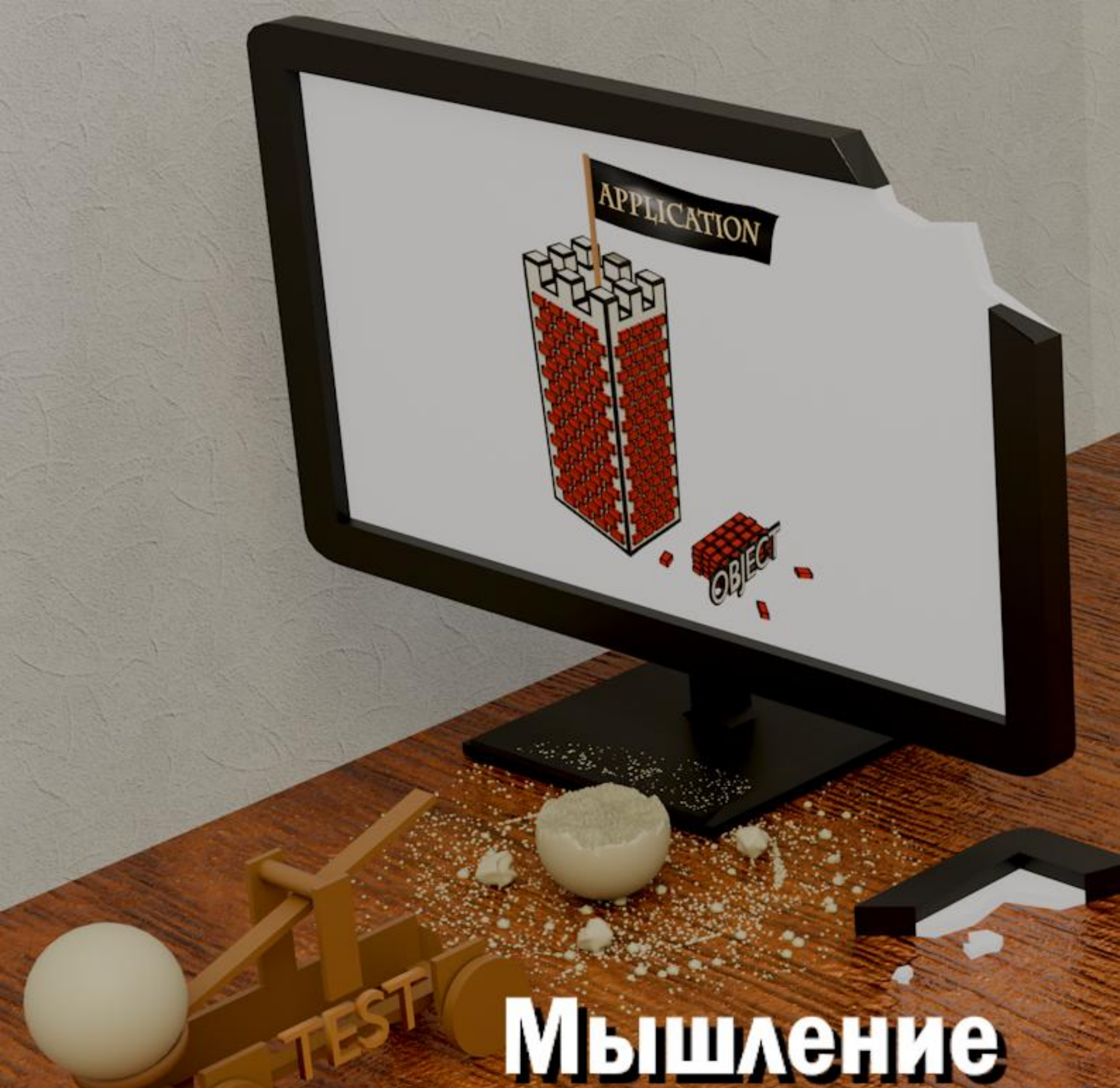


Мышление за пределами кода

Андрей Корниенко

МЫШЛЕНИЕ
ЗА ПРЕДЕЛАМИ КОДА

v2025-1.0

Содержание

Предисловие

О чём эта книга?

Давай я немного расскажу тебе, о чём эта книга и что ты сможешь почерпнуть из неё.

Идея написать её была сложной, потому что я **не хотел создавать что-то похожее на документацию**. Лично я не люблю утомительные строчки текста, в которых легко потеряться и забыть прочитанное. Поэтому я решил подойти **более креативно**, сделать текст **более свободным и лёгким для восприятия**, снизив порог вхождения.

Формат книги

Возможно, кто-то посчитает минусом **упрощение формулировок**, но я сделал это **осознанно**, чтобы тема была понятной **любому человеку, независимо от опыта**. Я мог бы наполнять книгу точными терминами, сложными схемами и формальными примерами, но был бы ты готов читать такую книгу? Лично я — нет.

После работы, вечером, с чашкой чая мне не хочется **максимально напрягаться**, чтобы осилить сухой материал. И тебе не нужно будет программировать, разбирать сложные примеры или набирать код на клавиатуре. **Мы будем размышлять**. **Ты и я.**

Формат диалога

Мы построим **диалог**:

- Как поступить в той или иной ситуации?
- Какие есть проблемы?
- Какие есть решения?
- Где могут быть противоречия?

Важно, что цель этого диалога — не просто объяснить концепции, а **помочь тебе осознать, зачем всё это нужно** и как ты можешь

применить эти идеи в своей работе или проекте.

Как читать эту книгу?

Если ты **не знаком** с какими-то понятиями, **не переживай**. После прочтения книги ты сможешь углубиться и изучить их более подробно. Ведь **идеальных архитектур не существует**, и нет универсального подхода, который подойдёт для любого проекта. Всегда нужно адаптировать, дополнять и переосмысливать.

Но суть и основу — **мы с тобой попробуем разобрать**.

Если ты дочитал до этого момента, поздравляю! В таком свободном формате ты сможешь **дойти до конца книги**.

Глава 1. Введение в ООП

ООП — что это и зачем?

ООП — это... Хотя нет, давай оставим определение на потом. Определение — это слишком банально. Нам нужно разобраться не в том, **что такое ООП**, а **зачем оно тебе и мне**.

Другими словами, **какая от него польза?**
Что мы получаем, применяя его? Какие проблемы оно решает? И главное — **"что бы что?"**

*Я сам **не люблю делать** что-то, на что **не могу ответить** на простой вопрос: **"А что бы что?"***

Давай разбираться вместе.

Давай вспомним какие подходы есть?

Ты, скорее всего, слышал (или, возможно, уже используешь) функциональный стиль программирования. В его основе лежат функции, а сам подход строится на цепочке вызовов, которые, в идеальном случае, не должны быть мутабельными или асинхронными.

Но в реальности на таком подходе далеко не уедешь. Поэтому мы вынуждены писать код, который вроде бы следует функциональному стилю, но при этом использует нефункциональные концепции. Это приводит к тому, что в проекте накапливается множество файлов с огромным количеством функций.

Знакома ли тебе ситуация, когда ты бегаешь по файлам в поисках нужной функции или пытаешься проследить цепочку вызовов, а она неожиданно приводит тебя к модулям, о которых ты даже не думал?

Где же проблема?

Что является корнем этой сложности: сам подход, задача, программист, нехватка времени? Я бы сказал, что однозначного ответа нет. Такой

результат неизбежен в сложных проектах, где логика разрастается и изменяется. В процессе разработки появляются дублирующиеся функции, файлы, перегруженные логикой, и методы с огромным количеством параметров.

Как же с этим справиться? Может, стоит задуматься об архитектуре?

Нужна ли нам архитектура?

"Архитектура" звучит пугающе, но давай разберемся. Это не что-то страшное — это всего лишь результат твоего подхода и тех мыслей, которые ты фиксируешь перед тем, как открыть пустой проект и начать писать код.

Допустим, у тебя есть UI с кнопками, выполняющими определенные действия. Например, кнопка добавления кубика в сцену и кнопка его удаления. На этом этапе мы не задумываемся о том, как именно кубик добавляется или удаляется — нас интересует, что у нас есть две кнопки и, скорее всего, две функции, которые их обслуживают.

Через некоторое время заказчик просит добавить возможность изменения количества кубиков. Позже приходит запрос на изменение их цвета. В идеальном мире у тебя есть четыре функции для четырех кнопок, но мы понимаем, что 3D-логика требует вспомогательных методов и функций общего назначения, чтобы избежать дублирования кода.

Проект готов — все работает. Но вот заказчик говорит: "А теперь добавьте историю изменений, чтобы можно было отменить последние действия по Ctrl + Z."

Тут встает вопрос: где вообще хранятся кубики? Если централизованного объекта нет, придется его создавать и рефакторить код. Затем нужно пройтись по всем функциям (добавление, удаление и т. д.), встроить в них механизм сохранения копий состояния и передавать эти данные в систему истории.

Фух, вроде бы всё работает. Но вот приходит новая задача, и теперь в каждой новой функции тебе снова нужно помнить про историю, добавлять в неё данные и следить, чтобы ничего не сломалось.

Звучит не очень приятно, правда?

А что, если я скажу тебе, что можно было бы добавлять новые функции без необходимости вручную обновлять историю? Что если бы твоя архитектура сама справлялась с этим без дополнительных изменений в коде?

Что дальше?

Мы с тобой разобрали реальные проблемы, которые возникают в проектах. Они есть у всех, в том числе и у меня, и это нормально.

Но нам нужно стремиться к улучшению. Я предлагаю не просто изменить подход, но и на его основе построить архитектуру приложения...

Глава 2. Базовые концепции ООП

Прежде всего, если ты никогда не работал с ООП, то, как и в любом другом случае, стоит немного почитать и посмотреть видео по теме. Благо, на дворе XXI век, и информации предостаточно.

Но тебе не нужно срочно бежать и смотреть видосы и читать книги. В этой главе я немного затрону основные концепции ООП, но мы будем говорить не столько о коде, сколько о мышлении.

Что такое ООП?

ООП — это всего лишь один из подходов к организации кода. Его ключевая особенность заключается в том, что у нас есть некая сущность (или модель), которая объединяет состояние и **поведение**.

И самое важное — состояние и поведение привязаны к конкретной модели.

Я объясняю это своими словами, без заумных определений. Моя цель — чтобы ты прочитал и понял идею, а не перегружал себя сложной технической терминологией.

ООП в контексте

Объектно-ориентированный подход особенно хорошо работает, когда мы имеем дело с **предметной областью**. Чтобы лучше его понять, давай возьмем пример — компьютерную игру.

Почти в любой игре есть **персонаж** (игрок), которого ты можешь контролировать. Есть **другие персонажи**(например, враги), которые могут атаковать тебя. В какой-то момент твой персонаж может проиграть, например, умереть.

Как это можно описать в стиле ООП?

```
class Character {
```

```
  constructor(name, health, speed, attackPower) {
    this.name = name;
    this.health = health;
    this.speed = speed;
    this.attackPower = attackPower;
  }

  move(direction) {
    console.log(`${this.name} moves ${direction} at speed
${this.speed}`);
  }

  attack(target) {
    console.log(`${this.name} attacks ${target.name} with power
${this.attackPower}`);
    target.takeDamage(this.attackPower);
  }

  takeDamage(amount) {
    this.health -= amount;
    console.log(`${this.name} takes ${amount} damage. Health
left: ${this.health}`);
    if (this.health <= 0) {
      this.die();
    }
  }

  die() {
    console.log(`${this.name} has died.`);
  }
}

// Пример использования
const player = new Character("Hero", 100, 10, 20);
const enemy = new Character("Goblin", 50, 5, 10);

player.move("forward");
player.attack(enemy);
enemy.attack(player);
```

Что мы здесь видим?

Мы смогли **описать состояние** нашего персонажа:

```
    this.name = name;
    this.health = health;
    this.speed = speed;
    this.attackPower = attackPower;
```

А также **определить его поведение** с помощью методов:

```
move
attack
takeDamage
die
```

Теперь давай попробуем реализовать ту же логику, но используя функциональный стиль.

```
// Функциональный подход

function createCharacter(name, health, speed, attackPower) {
  return {
    name,
    health,
    speed,
    attackPower
  };
}

function move(character, direction) {
  console.log(`${character.name} moves ${direction} at speed
${character.speed}`);
}

function attack(attacker, target) {
  console.log(`${attacker.name} attacks ${target.name} with
power ${attacker.attackPower}`);
  takeDamage(target, attacker.attackPower);
}

function takeDamage(character, amount) {
  character.health -= amount;
  console.log(`${character.name} takes ${amount} damage. Health
```

```
left: ${character.health}`);
  if (character.health <= 0) {
    die(character);
  }
}

function die(character) {
  console.log(`${character.name} has died.`);
}

// Пример использования
const player = createCharacter("Hero", 100, 10, 20);
const enemy = createCharacter("Goblin", 50, 5, 10);

move(player, "forward");
attack(player, enemy);
attack(enemy, player);
```

ООП vs. Функциональный подход

Обрати внимание, что во втором варианте у нас просто **набор функций**, которые не связаны между собой контекстом.

Каждой функции нужно явно передавать все необходимые данные через параметры. Пока код небольшой — это нормально, но при масштабировании легко можно получить те же самые проблемы, о которых мы говорили в предыдущей главе:

- **Дублирование кода**
- **Сложность отслеживания связей между функциями**
- **Отсутствие единого контекста**

Теперь возвращаемся к ООП.

В чем разница? В объектно-ориентированном варианте у нас есть **сущность (класс)**, которая объединяет в себе **состояние и поведение**. Это по умолчанию делает его **модулем**, который сам по себе логично организует код.

Еще немного базовых концепций *(После прочтения этой главы ты можешь изучить их подробнее, но я буду делать акцент на мышлении и проблемах, которые мы можем решить.)*

Я постараюсь затронуть основные моменты, чтобы, если ты что-то не знал или забыл, можно было вспомнить. Эти термины и концепции станут основой для дальнейшего понимания.

ООП включает в себя множество аспектов, но сейчас нам не нужно погружаться в них слишком глубоко. Наша цель — вместе поразмышлять над архитектурой с использованием ООП. Поэтому набираемся сил и читаем следующий отрезок с пониманием того, что это необходимо, чтобы мы говорили на одном языке в следующих главах.

Модификаторы доступа

В объектно-ориентированном программировании (ООП) **модификаторы доступа** определяют, какие части кода могут взаимодействовать с определёнными свойствами и методами классов.
Они помогают **ограничивать доступ** к данным и предотвращать случайные изменения или нежелательное использование.

В большинстве языков программирования (например Java, C#) существуют три основных модификатора доступа:

1. public (публичный)

Открытый доступ: свойство или метод доступны из любой части программы.

```
class Character {

public name: string; // Доступен везде
constructor(name: string) {
    this.name = name;
}
publicsayHello() { console.log(`Hello, my name is
${this.name}`); } }

const player = new Character("Hero");
```

```
console.log(player.name); // Можно свободно обращаться
player.sayHello(); // Можно вызвать метод
```

2. private (приватный)

Доступен **только внутри класса**. Ни дочерние классы, ни внешний код не могут его использовать.

```
class Character {
  private health: number; // Доступно только внутри этого
класса

  constructor(health: number) {
    this.health = health;
  }

  private takeDamage(amount: number) {
    this.health -= amount;
    console.log(`Health left: ${this.health}`);
  }

  public simulateAttack() {
    this.takeDamage(10); // Можно вызывать внутри класса
  }
}

const player = new Character(100);
// player.health = 50; // Ошибка: свойство private
// player.takeDamage(10); // Ошибка: метод private
player.simulateAttack(); // Работает
```

3. protected (защищённый)

Доступен **внутри класса и в его потомках**, но **не снаружи**.

```
class Character {
  protected health: number; // Доступен только в этом классе и
его наследниках
```

```
  constructor(health: number) {
    this.health = health;
  }
}

class Player extends Character {
  public heal() {
    this.health += 10; // Можно изменять, так как класс
наследуется
    console.log(`Health after healing: ${this.health}`);
  }
}

const player = new Player(100);
player.heal(); // Работает
// player.health = 50; // Ошибка: health protected
```

Три главные парадигмы ООП

1. Инкапсуляция (Encapsulation)

Суть: **объединение данных и логики в одном объекте, а также ограничение доступа к внутренним данным**. Это помогает защитить состояние объекта от неконтролируемых изменений.

```
class BankAccount {
  private balance: number; // Приватное свойство

  constructor(initialBalance: number) {
    this.balance = initialBalance;
  }

  public deposit(amount: number) {
    if (amount > 0) {
      this.balance += amount;
      console.log(`Deposited ${amount}, new balance:
${this.balance}`);
    }
  }
```

```
  public getBalance() {
    return this.balance;
  }
}

const account = new BankAccount(1000);
account.deposit(500);
console.log(account.getBalance()); // 1500
// account.balance = 0; // Ошибка: balance – private
```

2. Наследование (Inheritance)

Суть: **создание новых классов на основе уже существующих**. Дочерний класс получает все свойства и методы родительского, но может их переопределять или дополнять.

```
class Animal {
  public makeSound() {
    console.log("Some generic animal sound");
  }
}

class Dog extends Animal {
  public makeSound() {
    console.log("Woof! Woof!");
  }
}

const genericAnimal = new Animal();
genericAnimal.makeSound(); // "Some generic animal sound"

const myDog = new Dog();
myDog.makeSound(); // "Woof! Woof!"
```

3. Полиморфизм (Polymorphism)

Суть: **объекты разных классов могут обрабатывать одинаковые вызовы по-разному**. Это позволяет переопределять методы в

наследниках и абстрагировать логику.

```
class Shape {
  public draw() {
    console.log("Drawing a shape");
  }
}

class Circle extends Shape {
  public draw() {
    console.log("Drawing a circle");
  }
}

class Square extends Shape {
  public draw() {
    console.log("Drawing a square");
  }
}

const shapes: Shape[] = [new Circle(), new Square(), new
Shape()];
shapes.forEach(shape => shape.draw());

// Output:
// Drawing a circle
// Drawing a square
// Drawing a shape
```

Зачем это все?

В реальном проекте очень важно, чтобы код **был структурирован**, иначе он превратится в хаос. Использование классов помогает группировать логику в понятные и удобные **модули**, что делает код более управляемым.

Но это не значит, что ООП — идеальный инструмент для всего. У него есть свои плюсы и минусы, о которых мы поговорим в следующих главах.

Теперь, когда у нас есть базовое понимание, давай попробуем разобраться, **как архитектура помогает нам избежать проблем в сложных проектах...**

Глава 3. Предметная область

Что-то знакомое, но не совсем?

Не совсем понимаешь, о чем идет речь? А что, если я скажу тебе, что ты уже выполнял часть этой работы — возможно, неосознанно или не полностью, но все же выполнял?
Давай для начала попробуем сформулировать простыми словами, что это такое.

Предметная область — это что-то (или даже кто-то), по отношению к чему мы исследуем **поведение, состояние, условия и зависимости**.

Я постарался объяснить это максимально просто, но давай разберем небольшой пример.

Пример предметной области: 3D-конфигуратор диванов

Представь, что тебе нужно разработать **3D-конфигуратор диванов**. Пользователь должен иметь возможность выбирать **тип секций, цвет, материал обивки** и видеть изменения в реальном времени.

Какие объекты и процессы здесь могут быть?

- **Сам диван** — объект с определёнными параметрами (размер, цвет, материал).
- **Пользовательский выбор** — механика, позволяющая изменять характеристики.
- **Отображение в 3D** — визуальная модель, которая обновляется при изменении параметров.
- **Логика конфигурации** — например, некоторые материалы нельзя комбинировать с определёнными секциями.

Таким образом, **предметная область** здесь — это **конфигурируемая мебель**, а наша задача — продумать, как код будет учитывать и обрабатывать эти изменения.

Анализ предметной области

Степень анализа зависит от потребностей. Например, если мы разрабатываем продукт **с нуля**, не имея никакой основы или базы, то в идеальном варианте придется проводить **постепенный анализ**: начиная с изучения **аналогов**подобных продуктов и заканчивая составлением **Use Cases**, описывающих различные сценарии использования.

Но на практике все немного иначе. Чаще всего мы **разрабатываем дополнение или сервис** к уже существующему продукту для заказчика.

Например, у заказчика уже есть **интернет-магазин по продаже диванов**, где представлено множество моделей с разными параметрами: **формой, комплектацией, цветом, материалами** и т. д. В этом случае у нас уже есть:

- **Список правил и ограничений**,
- **Набор существующих данных**,
- **Готовые требования по дизайну и логике**.

Поэтому **глубокий анализ аналогов уже не требуется**, так как основа системы уже определена.

В рамках этого обсуждения мы будем рассматривать именно **такой вариант** — работу с уже существующим продуктом.

Если же тебе интересен **полный анализ с нуля**, ты можешь изучить эту тему дополнительно. Но сразу предупреждаю: некоторые части могут показаться бессмысленными или сложными для понимания. Это нормально.

На данный момент **нам достаточно того, что мы разберем здесь**, чтобы заложить базу для понимания анализа предметной области.

Анализ предметной области "3D-конфигуратор диванов"

В данном случае **анализ предметной области** будет более **локальным**, так как мы **не создаем продукт с нуля**, а разрабатываем **дополнение или сервис** к уже существующей системе (например, интернет-магазину

по продаже диванов). Это значит, что **анализ должен учитывать существующий контекст**, а не разрабатываться с чистого листа.

1. Что уже есть у заказчика?

Перед началом разработки **важно понять**, какие элементы системы уже существуют. В случае с **магазином диванов** у заказчика могут быть:

- **Готовые данные** (база товаров, их характеристики, цены).
- **Правила и ограничения** (например, нельзя выбрать цвет, который отсутствует на складе).
- **Требования по дизайну** (сайт уже оформлен в определенном стиле).
- **Логика взаимодействия** (например, добавление товаров в корзину, оформление заказа).

2. Какие новые функции добавляются?

Так как мы разрабатываем **дополнение** (например, 3D-конфигуратор диванов), важно **понять, какие новые сущности** появятся в системе и как они будут взаимодействовать с существующими данными.

Новые сущности

- **3D-модель дивана** — визуальное представление товара, изменяемое в реальном времени.
- **Панель конфигурации** — UI-элемент для изменения параметров.
- **Логика изменения параметров** — механизм обновления модели в зависимости от выбора пользователя.
- **Пересчет цены** — динамическое обновление стоимости в зависимости от характеристик.

Связь с существующей системой:

- **Данные о диванах** уже есть в базе → используем их для настройки конфигуратора.
- **Выбор пользователя** влияет на отображение **3D-модели** и обновляет стоимость.
- **Оформление заказа** должно учитывать параметры, настроенные в конфигураторе.

3. Какие ограничения и зависимости нужно учесть?

Поскольку мы работаем с **существующей системой**, необходимо учитывать **уже установленные бизнес-правила и технические ограничения**.

Ограничения конфигуратора

- Выбор материала **должен соответствовать** тем, что есть в базе магазина.
- Цветовая палитра **зависит от поставщика** (нельзя выбрать несуществующий цвет).

Зависимости между параметрами

- Цена динамически пересчитывается в зависимости от **размера, материала и дополнительных опций**.
- Конфигуратор **должен корректно передавать данные в корзину** для оформления заказа.

4. Архитектурные решения на основе анализа

Проведенный анализ помогает понять, **как правильно интегрировать конфигуратор**.

1. **Используем существующую базу товаров** — конфигуратор получает данные из магазина.
2. **Разделяем логику конфигуратора и рендер 3D-модели**, чтобы можно было изменять UI без переписывания всей системы.
3. **Обеспечиваем валидацию выбора** (например, предотвращаем выбор недоступных опций).
4. **Настроенная конфигурация должна корректно передаваться в систему заказов**.

Заключение

В этом варианте **анализ предметной области направлен не на создание новой системы, а на адаптацию новых функций к**

существующему контексту.

Главное отличие от полного анализа — мы **не изучаем аналоги, не создаем глобальные схемы**, а **работаем в рамках уже готового продукта**, добавляя новые сущности и связывая их с существующими данными.

Анализ Предметной области **задаёт контекст для разработки**, и без этого этапа можно легко построить неудобную или неэффективную систему.

Глава 4. Проектирование структуры

После того как мы с тобой поговорили о **предметной области** и определили **ключевые сущности**, нам нужно перейти к **организации этих сущностей**, то есть вывести **структуру**, которая будет **расширяемой**.

Когда слышишь **"структура проекта"** или **"структура модуля"**, сразу приходит на ум: «Ну вот, существуют такие папки, как **components, models, pages...**» и так далее. И вроде бы все понятно: **из проекта в проект мы переносим структуру**. Да, она может немного отличаться — папки могут иметь **другие названия**, когда ты смотришь на проект другого человека. **Но суть остается схожей**: папки хранят определенные данные и функциональность.

Так что, **это и есть структура? И да, и нет!**

Проектирование структуры — это нечто большее, чем просто **названия папок**. Это важнейший процесс, который определяет, как будет устроена архитектура, насколько **гибкой и масштабируемой** она окажется.

Теперь разберемся, что такое действительно продуманная структура и как ее проектировать правильно.

Проектирование структуры — это **не просто разбиение файлов на папки**, а продуманный процесс, который влияет на удобство работы с кодом, масштабируемость и поддерживаемость проекта.

Почему важна правильная структура кода?

Хорошим ответом на этот вопрос, зачем и почему, это пример который описывает проблему которая приводит к последствия в будущем.

Продолжаем тему с конфигуратором, у нас есть **3D-конфигуратор диванов**, но логика изменения модели, управления UI и хранения данных **перемешана в одном файле**. Если заказчик попросит добавить новый параметр (например, возможность выбора текстуры), нам придется **переделывать весь код**, а это риск появления багов.

Плохая структура может привести к:

- Перепутанным зависимостям
- Дублированию кода
- Сложности внесения изменений
- Проблемам при расширении функционала

Хорошая структура делает код:

- Читаемым и логичным
- Гибким и масштабируемым
- Простым в тестировании
- Удобным для командной работы

Основные принципы организации кода

Разделение ответственности (Separation of Concerns)

Каждый модуль должен выполнять **только свою задачу**.
Пример:

- **UI-компоненты** отвечают только за рендеринг интерфейса
- **Бизнес-логика** управляет правилами конфигурации
- **Хранилище данных** сохраняет текущие настройки

Так код становится **гибким**: если нужно изменить UI, не придется переписывать бизнес-логику.

Чистая архитектура (Clean Architecture)

Один из распространенных подходов — **разделение кода на слои**, где каждый уровень имеет свою четкую ответственность.

Пример структуры для 3D-конфигуратора:

```
/src
  ├── components/          # UI-компоненты
  ├── models/              # Бизнес-логика
  ├── services/            # API и взаимодействие с сервером
  ├── store/               # Управление состоянием
```

```
├── assets/              # Изображения, 3D-модели, стили
├── hooks/               # Пользовательские хуки
├── utils/               # Вспомогательные функции
├── types/               # Определения типов (если TypeScript)
```

Плюсы такого подхода:

- UI остается **независимым** от логики
- Бизнес-логика **отделена** от хранения данных
- Код **масштабируется** без необходимости менять всю систему

Давайте создадим небольшую **диаграмму процесса**, которая поможет описать **поток взаимодействия между состоянием, логикой и UI**. Это позволит наглядно увидеть, **какой структуры мы должны придерживаться** при проектировании системы.

Основной принцип:

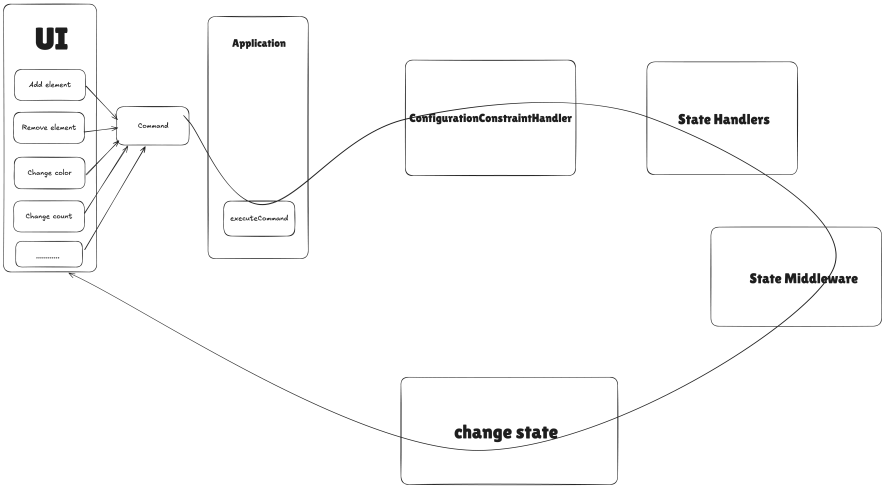
- **UI (Компоненты)** → отображает данные и вызывает действия
- **Логика (Бизнес-слой)** → управляет изменениями данных
- **Состояние (Хранилище данных)** → сохраняет и передает актуальные данные

Теперь давай представим это в виде **диаграммы**, но не будем рисовать банальные примеры. Вместо этого я возьму **диаграмму из моего проекта**.

Важно: Я **рисую диаграммы разных типов** как часть документации к своим проектам и **советую вам делать то же самое**. Это помогает лучше понимать архитектуру и упрощает поддержку кода.
Мы еще **вернемся к этой теме ближе к концу книги**, чтобы подробнее разобрать, **как правильно документировать проекты с помощью диаграмм**.

А сейчас давай **посмотрим на диаграмму**, обращая внимание на **поток данных и направление взаимодействий**.

Data modification process



Анализ диаграммы: что мы видим?

Что можно заметить, глядя на диаграмму? **Как направлен поток данных? Что от чего зависит? Насколько предсказуема структура проекта?**
Думаю, **ответы очевидны**.
Мы видим, что у нас есть **три главные сущности**, которые составляют этот процесс:

1. **UI (пользовательский интерфейс)**
2. **BLL (бизнес-логика)**
3. **State (хранилище состояния)**

Как устроен процесс?

1. **UI** вызывает **функцию**, которая в свою очередь может включать в себя **цепочку логик**.
2. **BLL (бизнес-логика)** состоит из **модулей**, в которых происходит обработка данных, применение правил и управление состоянием.
3. **State (состояние)** обновляется — **это наша истина** (я иногда называю состояние "истиной").

4. После изменения **состояния** происходит **перерендер UI** или его части, **замыкая цикл**.

Как тебе такая модель? Я предлагаю **попробовать самостоятельно**. **Возьми листочек и ручку** и **нарисуй такую диаграмму**, описывая, например, **добавление секции дивана в 3D-конфигураторе**.

Это упражнение поможет лучше понять, **как данные текут внутри системы** и какие **модули взаимодействуют друг с другом**.

Глава 5. Декомпозиция предметной области

После того как мы разобрались с предметной областью и определили ключевые сущности, следующим шагом будет **их декомпозиция**.

Если вы работали с проектированием **баз данных** (далее — **БД**), то должны быть знакомы с этим понятием.
Я сам познакомился с ним **в полном объеме** во время учебы в колледже, когда проектировал **БД для курсовой работы**.
Если вы думаете, что **декомпозиция** связана **только с БД** и не имеет отношения к проектированию системы в целом, то это **не так**.
Декомпозиция — это универсальный процесс, который помогает разбить предметную область на небольшие структуры и сущности, с которыми вам предстоит работать.

Это **ключевой этап проектирования**, поскольку он позволяет:

- **Оптимизировать систему**
- **Упростить понимание логики**
- **Сделать код более масштабируемым**

Давайте разберем этот процесс **более подробно**.

Почему важна декомпозиция?

Когда система слишком сложная, работать с ней становится неудобно. Теперь представь **3D-конфигуратор диванов**, в котором есть десятки параметров: **размер, форма, материал, цвет, цена, дополнительные элементы и т. д.**

Если вся логика **сосредоточена в одном месте**, это приведет к хаосу:

- Изменение одного параметра может сломать другие.
- Невозможно переиспользовать части системы в других модулях.
- Увеличивается сложность тестирования и отладки.

Чтобы избежать этого, **разделяем систему на независимые компоненты**.

Как правильно декомпозировать предметную область?

1. Разбиение на домены (крупные блоки системы)

На этом этапе мы определяем **основные части предметной области**, каждая из которых отвечает за свою функцию.

Пример:
Для **3D-конфигуратора диванов** можно выделить такие домены:

- **Каталог товаров** — список доступных диванов.
- **Конфигуратор** — изменение параметров модели.
- **Заказы** — оформление покупки.
- **Профиль пользователя** — сохранение настроек и истории заказов.

Каждый **доменный модуль** работает **изолированно** и взаимодействует с другими **через API или события**.

2. Разделение на модули (функциональные части)

Внутри каждого домена можно выделить **модули**, которые отвечают за конкретные задачи.

Пример модулей в **Конфигураторе**:

- **Модуль выбора материала и цвета**
- **Модуль отображения 3D-модели**
- **Модуль валидации**
- **Модуль управления состоянием**

Каждый модуль **можно переиспользовать**, что делает систему гибче.

Декомпозиция 3D-конфигуратора диванов

Давай **более детально** проведем декомпозицию. Пока что сделаем это **условно**, поскольку **названия** в дальнейшем могут отличаться.

Как ты думаешь, **что является истиной**? Для меня — это **данные**, то есть **состояние приложения**, которое описывает **истину в виде данных в определенный момент времени**.

Почему в **определенный момент**? Потому что **главное свойство состояния** заключается в том, что оно **может и должно изменяться**. Однако, обратившись к нему **в конкретный момент времени**, ты можешь быть уверен, что **получишь актуальные данные**.

1. Истина и ее представление

Такую сущность я привык называть **Configuration — некую конфигурацию**, которая является **истиной** для нашего приложения.
Она — **ключевая точка, изменяя которую, мы должны видеть результат в 3D-сцене**.

Пример:
Я изменяю **цвет**, то есть **меняю какое-то свойство (поле) объекта Configuration**, и это должно **запустить механизм** внутри приложения. Этот механизм может включать:

- **Расчеты**
- **Обновление модели**
- **Отображение изменений**

Конечный результат? **Цвет дивана в 3D-сцене изменился.**

2. Как изменять истину?

Теперь, когда мы разобрались с **истиной**, нам нужно **научиться ее изменять**.
В целом, мы **могли бы** написать **множество функций**:

- **Добавление**
- **Удаление**
- **Изменение цвета**

Эти функции **будут изменять нашу истину**, и если **подпереть их везде нужными вызовами**, то система **будет работать**.
Но будет ли такая система **расширяемой и гибкой**?

Мне кажется, **в лучшем случае — слабой**.

3. Делаем систему гибкой: команда (Command)

Что же делать?

Давай **создадим сущность**, которая станет **образцом состояния и поведения** для **всех типов действий**, которые мы хотим выполнять над истиной.

Назовем ее **Command**.

На данный момент **важно только одно**:

- У нас **есть сущность Command**, которая позволяет **реализовывать разные типы действий** над истиной.

Почему это удобно?
Теперь, при **добавлении новой функции**, мы **не создаем хаос** в коде, а работаем **с сущностью**, которая **уже встроена в систему**.
Все, что нам остается — **описать поведение новой команды**, которая будет **изменять истину**.

Не правда ли **круто**? **Как думаешь, делает ли это систему более гибкой?**

4. Работа с 3D-логикой

Что у нас есть?

- Истина (Configuration)
- Что-то, что изменяет истину (Command)

Но что делать дальше?

Ааа, **точно!** У нас есть часть, отвечающая за 3D, и у нее своя логика.
По идее, она у нас самостоятельная, и мы можем прикрутить любую реализацию 3D-логики в нашу систему.

Как это сделать?

Создать **сущность**, которая **определяет API методов и состояний**, используемых в коде.
При этом **что находится внутри этих методов — нам не важно**.
Мы **вызываем их** и **просто ожидаем определенный результат**.

Пример:
Допустим, такая сущность называется **Player3D**.

- Если мы вызываем `Player3D.changeColor()`, мы ожидаем, что **цвет в 3D-сцене изменится**.
- Если мы вызываем `Player3D.addSection()`, мы ожидаем, что **будет добавлена новая секция дивана**.

То есть, **логика 3D-отрисовки скрыта внутри**, а интерфейс взаимодействия с ней **четко определен**.

5. Связываем все воедино: Application

На данный момент мы **описали три ключевые сущности** нашего приложения:

- **Configuration** — хранит истину (состояние).
- **Command** — изменяет истину.
- **Player3D** — отвечает за 3D-отрисовку.

Но **они пока не связаны**.

Как это исправить?
Добавим **связующее звено** — **сущность Application**.
Она **будет управлять всем процессом** и обеспечивать **взаимодействие между частями системы**.

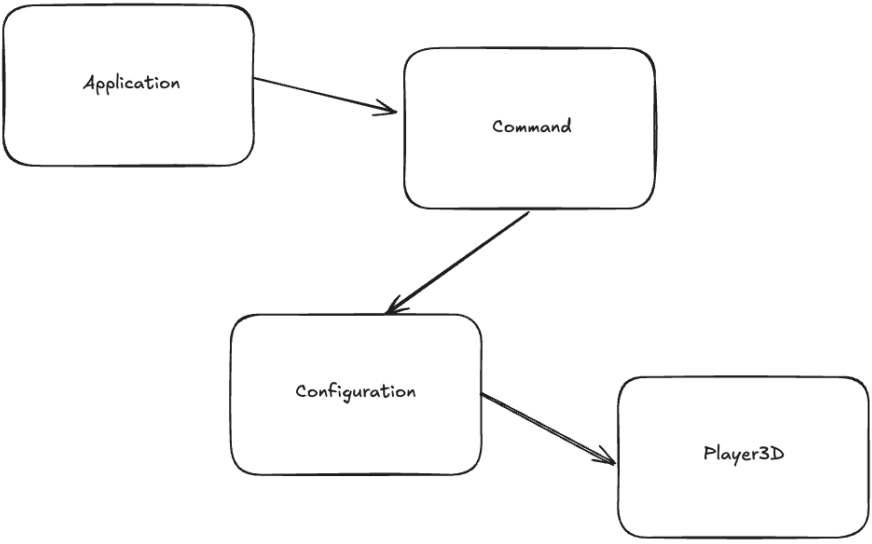
6. Визуализация структуры

Следующим шагом давай схематично нарисуем диаграмму, которая покажет направление работы и изменения сущностей.

Важный момент:

- На диаграмме будет показано **направление работы/изменения сущностей**.

- **Это не означает, что одна сущность напрямую вызывает другую** — между ними могут быть **второстепенные модули**, обслуживающие ту или иную логику.



Глава 6. Начинаем проектировать

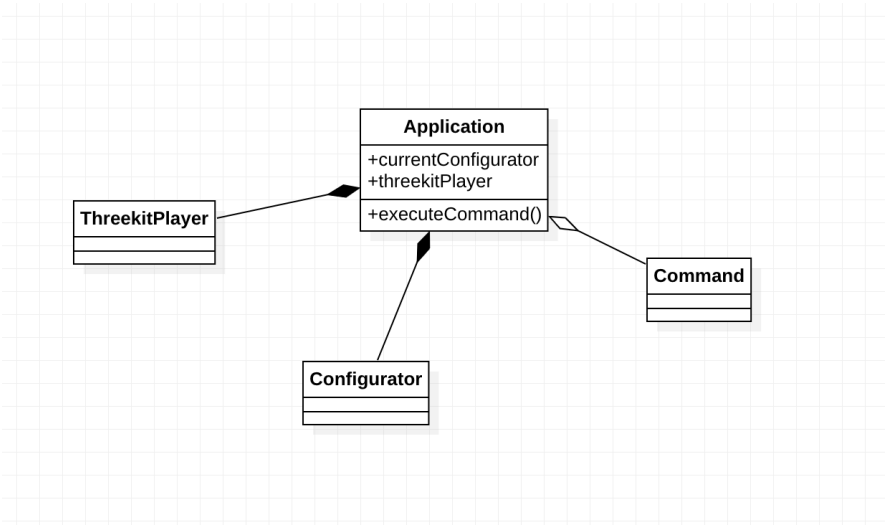
После того как мы выполнили декомпозицию, пора переходить к структурному мышлению и построению основных узлов системы.

Если ты помнишь из прошлой главы, главной идеей было определение истины и выделение ключевых узлов, таких как изменение истины и модуля, отвечающего за отображение данных в 3D. На данном этапе нам не важно, как именно данные будут рендериться в 3D, какой модуль или движок за это отвечает. Главное — понимать, что мы можем передать данные (нашу истину) модулю **Player3D** и получить условный результат.

Итог того, что у нас есть на данный момент:

- **Configurator** (ранее Configuration) — сущность, отвечающая за истину. Мы переименовали ее в **Configurator**, чтобы дать ей возможность не только хранить состояние, но и описывать поведение, которое может быть нам полезно.
- **Command** — ключевой элемент системы. Мы определили, что эта сущность будет описывать некий шаблон, с помощью которого можно внедрять новые функции в систему. Создавая новую команду в **Command**, мы добавляем новое поведение, не прибегая к костылям или изменениям существующей логики.
- **ThreekitPlayer** (ранее Player3D) — сущность, отвечающая за работу с 3D. Хотя раньше мы использовали название **Player3D**, теперь осознанно корректируем его на **ThreekitPlayer**. Это не изменяет смысл сущности, но более точно описывает инструмент, работающий с 3D-сценой. Если ты не знаком с **Threekit**, можешь ознакомиться с ним отдельно. Однако важно понимать, что этот модуль по-прежнему отвечает за отрисовку в 3D — это может быть **Three.js**, кастомная логика или, как в моем случае, **Threekit**.
- **Application** — сущность, объединяющая все вышеупомянутые элементы и позволяющая ими управлять.

Теперь попробуем отобразить взаимодействие этих четырех сущностей с помощью диаграммы. Для этого и последующих диаграмм я буду использовать **StarUML**.



Если вы посмотрите на диаграмму, то увидите, что в этот раз я использовал не просто квадратики и стрелочки, а инструменты, которые явно показывают, что перед нами сущности со своими свойствами и методами. Главная идея заключалась в том, чтобы продемонстрировать существование верхнего уровня, который управляет процессом. При этом остальные сущности остаются независимыми — они не знают о существовании других модулей и не должны знать, если это не является необходимым для конкретной реализации.

Стоит отметить, что это лишь схематическое отображение текущего состояния системы. В данном случае мы видим два новых свойства и один метод:

Свойства:
• **currentConfigurator**
• **threekitPlayer**

Метод:
• **executeCommand**

Со свойствами все очевидно: это объекты, созданные на основе наших классов, обладающие состоянием и методами для взаимодействия. Однако пока неясно, как именно все эти элементы будут взаимодействовать между собой.

Эту задачу выполняет метод **executeCommand**. Давай рассмотрим его работу с помощью схематического кода.

```
class Application {
    public currentConfigurator: Condigurator;
    public threekitPlayer: ThreekitPlayer;


    public executeCommand(command: Command) {
        const resultConf = command.execute();

        this.currentConfigurator = resultConf;
        this.threekitPlayer.renderConfigurator(resultConf);
    }
}
```

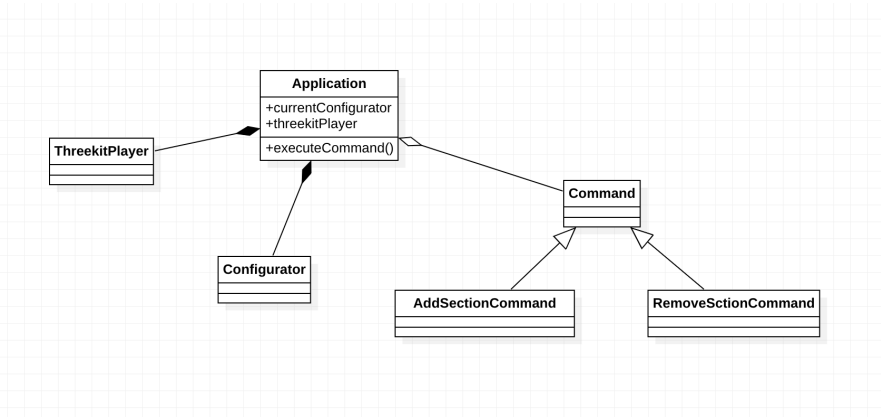
Я схематически изобразил, как это могло бы выглядеть, если бы мы ограничились только имеющимися сущностями. Главная идея заключается в том, что метод **executeCommand** умеет:

- выполнять переданную ему команду (**Command**),
- получать значения,
- вызывать рендер сцены для отображения новых изменений.

Однако пока не совсем понятно, какие именно команды он может выполнять. Давайте расширим нашу диаграмму и добавим новые функции:

- **Добавление новой секции**,
- **Удаление секции**.

Это позволит более детально показать, как команды интегрируются в нашу систему и какие изменения они могут вносить.



Мы добавили две новые сущности, которые наследуют основные методы от **Command** и реализуют собственную логику в соответствии со своими задачами.

Теперь давайте дополним наш схематический код:

```
class Application {
    public currentConfigurator: Condigurator;
    public threekitPlayer: ThreekitPlayer;

    public addSection(section: any) {
        this.executeCommand(new
AddSectionCommand(this.currentConfigurator, section))
    }

    public removeSection(section: any) {
        this.executeCommand(new
RemoveSectionCommand(this.currentConfigurator, section))
    }


    public executeCommand(command: Command) {
        const resultConf = command.execute();

        this.currentConfigurator = resultConf;
        this.threekitPlayer.renderConfigurator(resultConf);
```

```
    }
}
```

Я добавил всего два метода, которые можно вызывать в UI там, где это необходимо. Если внимательно посмотреть на их реализацию, можно заметить, что оба метода вызывают один и тот же **executeCommand**. Это позволяет нам стандартизировать процесс, обеспечивая единую точку изменения истины.

Благодаря такому подходу, в 95% случаев, чтобы добавить новую логику, достаточно просто описать новую **Command** и выполнить ее через **executeCommand**. Круто, не правда ли?

Теперь давайте перепишем метод **executeCommand** так, чтобы он соответствовал более логичной структуре и последовательности, где выполнение команды происходит перед вызовом изменений в 3D.

```
public executeCommand(command: Command): Promise<boolean> {
        return new Promise(resolve => {
            command.execute().then((res) => {
                if (!res) {
                    resolve(false);
                    return;
                }


                this.threekitPlayer
                    .renderConfigurator(command.configurator);

            })
        })

    }
```

Мы добавили обработку через **Promise**, что позволяет гарантировать выполнение команды в первую очередь, а затем — всех последующих изменений.

Что особенно важно, этот подход открывает дополнительные возможности:

- Мы можем логировать информацию о выполнении команд.
- Мы можем организовать историю команд, позволяя пользователю перемещаться по истории действий вперед и назад.

Теперь давай модифицируем **Application**, чтобы новые методы выглядели логично с точки зрения асинхронного выполнения команд.

```
class Application {
    public currentConfigurator: Condigurator;
    public threekitPlayer: ThreekitPlayer;

    public addSection(section: any): Promise<boolean> {
        return this.executeCommand(new
AddSectionCommand(this.currentConfigurator, section))
    }

    public removeSection(section: any): Promise<boolean> {
        return this.executeCommand(new
RemoveSectionCommand(this.currentConfigurator, section))
    }


    public executeCommand(command: Command): Promise<boolean> {
        return new Promise(resolve => {
            command.execute().then((res) => {
                if (!res) {
                    resolve(false);
                    return;
                }
                this.threekitPlayer
                    .renderConfigurator(command.configurator);

            })
        })

    }
}
```

Глава 7. Проектирование конфигуратора

Неплохой результат мы с тобой получили в предыдущей главе, не так ли? В целом, уже можно взять это за основу — система выглядит вполне достойно.

Но мы будем двигаться дальше. Система должна представлять собой нечто большее, чем просто работу с данными. Нам нужно добиться **расширяемости** и **гибкости**.

Что делать с логикой самого 3D?

У нас есть **ThreekitPlayer**, который явно отвечает за API, позволяющее изменять что-то в **Player3D**. Но что, если нам потребуется модифицировать данные, полученные из **Player**, или задать какие-то параметры?

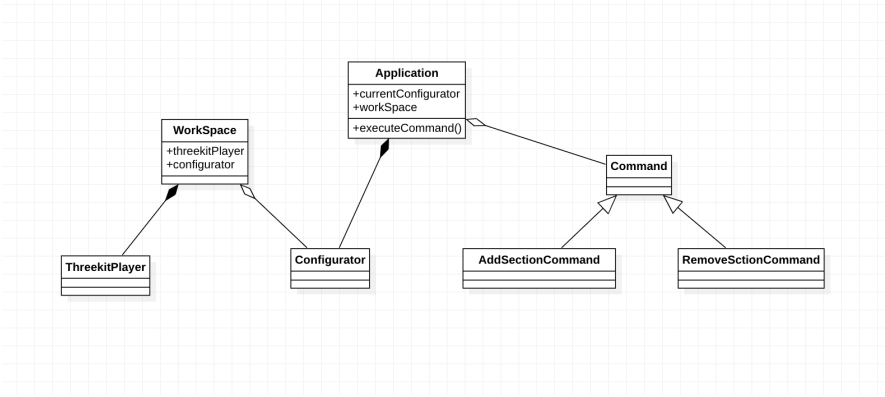
- Где должны храниться методы, выполняющие роль “прослойки”?
- Где должны храниться, например, данные о выбранной секции дивана?

Допустим, нам нужно выделить определенную секцию дивана. Хранить это в **истине**, в нашем **Configurator**, кажется не лучшим решением. Нам необходимо нечто, что обеспечит:

1. **Временное состояние** — для работы с текущими изменениями.
2. **Методы для обработки логики** — связь между **Configurator** и **ThreekitPlayer**.

Сейчас у нас отсутствует соответствующая сущность. Ее необходимо добавить. Я думаю, хорошим названием для нее будет **WorkSpace**.

Давай рассмотрим следующую диаграмму, где эта новая сущность уже включена.



WorkSpace выступает связующим звеном, объединяющим наше состояние и логику, работающую с **ThreekitPlayer**. Это дает нам больше возможностей для написания специфической логики, которая:

- **Не относится напрямую к состоянию** (то есть к **Configurator**),
- **Не относится непосредственно к ThreekitPlayer**.

В дальнейшем будут отдельные главы, которые помогут тебе глубже разобраться в ключевых сущностях этой системы.

Важность предметной области

Как я уже говорил, сложность системы напрямую зависит от предметной области и функционала. В данном случае у нас **конфигуратор диванов**, который предполагает работу с **секторами** (секции дивана). Эти секции представляют собой определенные сущности, которые:

1. Можно **добавить** в нашу истину (**Configurator**).
2. Можно **увидеть** на сцене (**Player**).

Введение новой сущности — Section

Назовем эту сущность **Section**. Она включает в себя свойства, которые ее характеризуют, например:

- **Название секции**
- **Материал**

Однако, когда мы добавляем **Section**, она должна иметь **определенную позицию** в сцене (**Player**). Это значит, что мы должны уметь:

- Узнать, **где** она находится.
- Пересчитать или изменить ее **позицию**, если это необходимо.

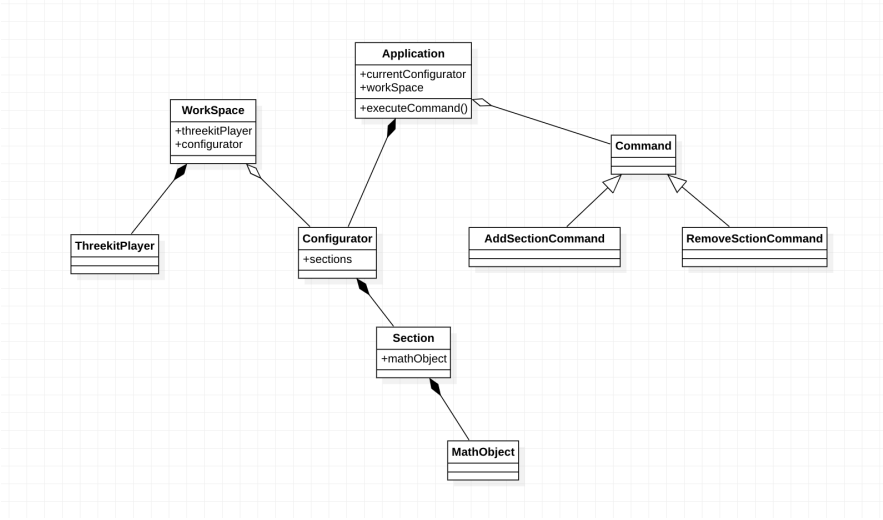
Здесь явно прослеживается логика, связанная с математическими вычислениями. Но материал секции, цвет, количество подушек и другие характеристики **не пересекаются** с ее позицией.

Введение новой сущности — MathObject

Таким образом, у нас появляется новая сущность — **MathObject**. Это абстрактный математический объект, который:

- Хранит **координаты**.
- Включает в себя **математические методы** для работы с положением и вычислениями.

Теперь давай рассмотрим новую диаграмму с этими сущностями.



Теперь наша система начинает выглядеть как настоящая, в которой есть сущности, описывающие **состояние** и **поведение**. Это система, которая может существовать и развиваться.

Но важно понимать: **нет волшебной таблетки**. Всегда нужно адаптировать и настраивать архитектуру в зависимости от потребностей конкретного проекта. Я не раз повторю эту мысль.

Моя цель — не предложить шаблон архитектуры, а показать **мыслительный процесс** на примере проекта **“Конфигуратор диванов”**. Анализируя проблемы, функционал и предметную область, мы строим систему, которая:

- **Минимально удовлетворяет** базовые будущие изменения и расширения,
- **Может масштабироваться** под более сложные задачи.

Гибкость системы и необязательные сущности

В следующей главе я добавлю новые сущности, которые сделают систему более **гибкой**. Однако они **не всегда обязательны**.

Например, в нашем случае, где система работает с **секторами (секции дивана)**, такие сущности необходимы. Но если система **более простая**, например:

- Она **просто изменяет** какие-то значения,
- Не хранит сложные структуры, такие как секции,
- Не требует управления позициями,

то некоторые узлы системы могут **отсутствовать**.

Понимая **цепочку мыслей**, которая привела к данной архитектуре, вы всегда можете двигаться **в обратном направлении**, убирая ненужные узлы и упрощая систему.

Глава 8. Проектирование, дополнительные узлы

Дополнительные узлы для гибкости системы

Давайте рассмотрим дополнительные узлы, которые позволят сделать систему более **гибкой**.

Вернемся к **Section**. Как ты думаешь, будет ли добавление секции включать в себя множество расчетов и логики? Где лучше сосредоточить обработку **позиции** и ее перерасчет?

Может быть, в **Command**? В целом, это хорошая идея, ведь мы уже обсуждали, что Command отвечает за **описание логики**. Однако **не лучшим решением** будет помещать именно логику расчета **позиции** в **Command**. Почему?

1. **Логика расчета позиций может быть сложной** — со временем она усложнится и потребует переиспользования.
2. **Эта логика может понадобиться в разных местах**, а не только в одной команде.

Введение новой сущности

Какой же должна быть сущность, которая занимается расчетом позиций для **Section**?

- Она должна **принимать текущий Configurator**.
- Она должна **обрабатывать добавление новой Section**.
- Внутри себя она выполняет **необходимые вычисления** и **устанавливает позицию** новой секции.

Таким образом, эта сущность **инкапсулирует** процесс, который может понадобиться в **разных частях системы**, делая архитектуру **более понятной и гибкой**.

В коде это будет выглядеть так:

1. **Command** вызывает эту новую сущность.
2. Передает ей **текущую конфигурацию** и **новую секцию**.
3. Получает в ответ **секцию с рассчитанными координатами**.
4. **Command** модифицирует **Configurator**, обновляя данные.

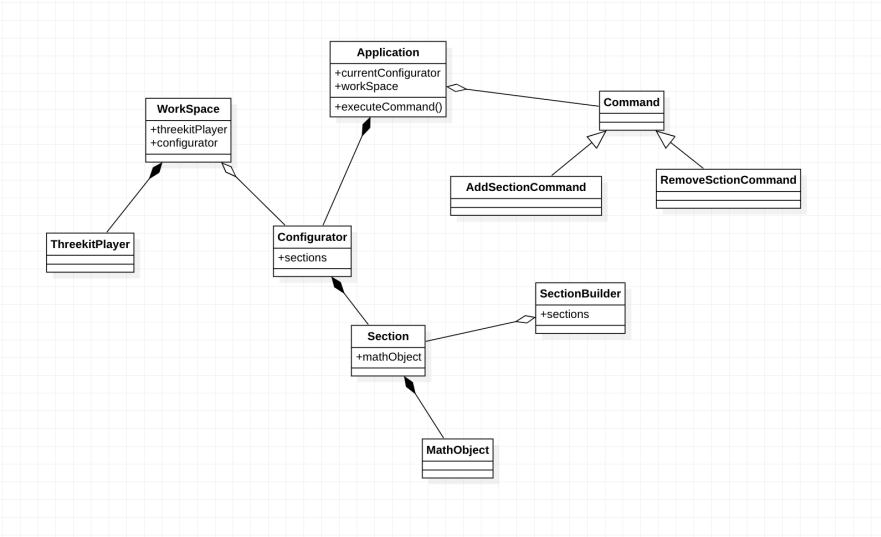
В результате у нас остается четкое **разделение логики** на модули.

Как назвать эту сущность?

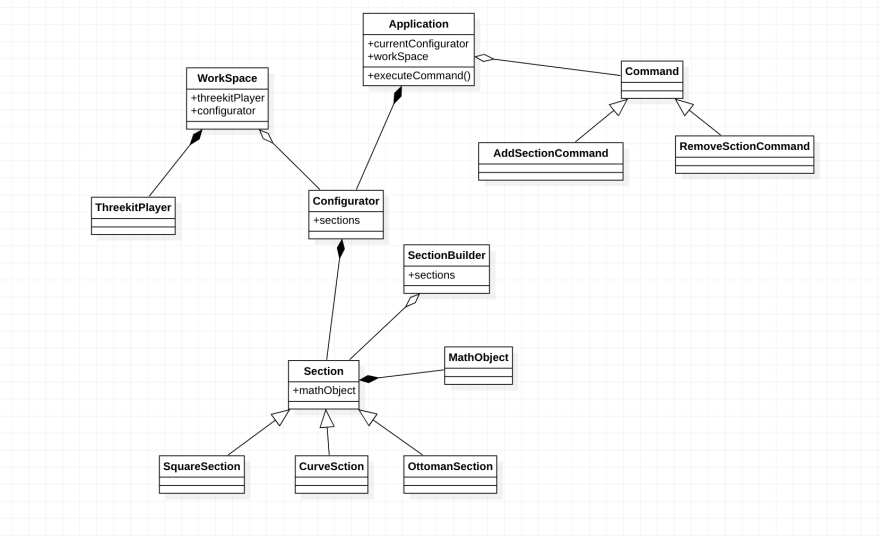
Она выполняет **роль строителя** — умеет **анализировать текущую структуру** и **добавлять новую секцию** с правильной позицией.

Идеальное название для нее — **SectionBuilder**.

Теперь давай рассмотрим схему с новой сущностью.



После того как мы добавили **SectionBuilder**, отвечающий за изменение позиций секций, давай также **расширим типы секций**, чтобы сделать процесс **добавления новой секции** более наглядным.



Теперь у нас появились **типы секций**, из которых состоит наш диван. Это позволяет нам **конфигурировать диван**, добавляя и изменяя его состав. Этим процессом будет управлять **SectionBuilder**, который позволяет добавлять новые секции к уже существующей конфигурации.

Дополнительные возможности для Command

Следующим необязательным узлом системы станут **дополнительные возможности для Command**.

На данный момент **Command** представляет собой **четко структурированную и предсказуемую сущность**. Она:

- Получает **конфигурацию** и необходимые данные для выполнения команды (например, новую секцию).
- Вносит изменения в конфигурацию **согласно заданной инструкции**.

Однако бывают ситуации, когда **командам требуется дополнительное поведение**. Это поведение:

- Может **быть актуальным** для одной команды в определенный момент.
- Может **утратить актуальность** для одной команды и стать нужным для другой (например, по запросу заказчика).

Как сделать систему расширяемой?

Нам нужно организовать структуру кода так, чтобы:

- **Добавлять новое поведение** в систему в любой момент.
- **Не изменять и не ломать существующую логику команд**.

Как мы помним, **хорошая архитектура** не требует изменения старого кода при добавлении новой функциональности.

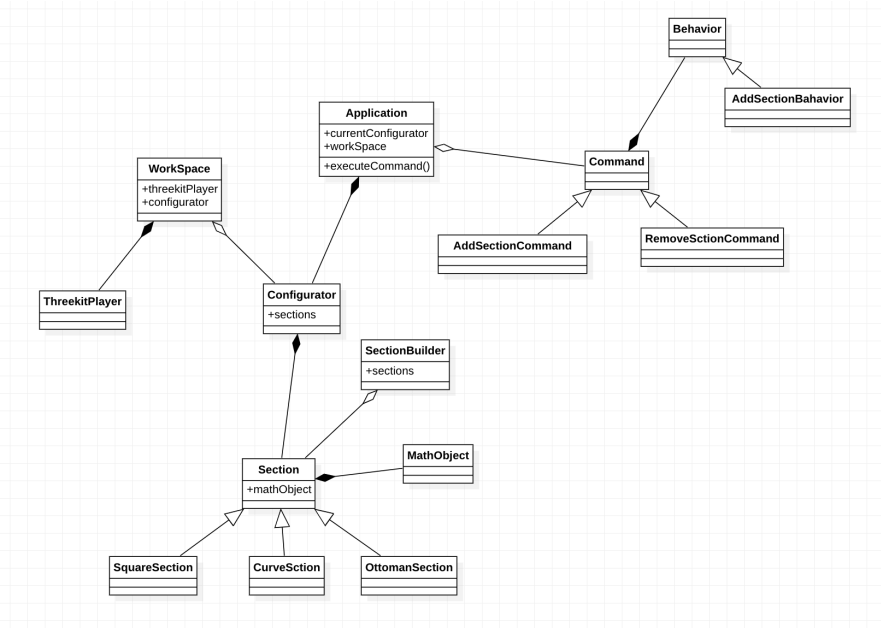
Введение новой сущности — Behavior

Дополнительное поведение команд — это **отдельная сущность**, которая:

- **Может быть применена к команде**, но не является обязательной.
- Позволяет **добавлять и изменять поведение команд** без модификации их основной логики.

Я предлагаю назвать эту сущность **Behavior**.

Теперь давай посмотрим на диаграмму с новой сущностью.



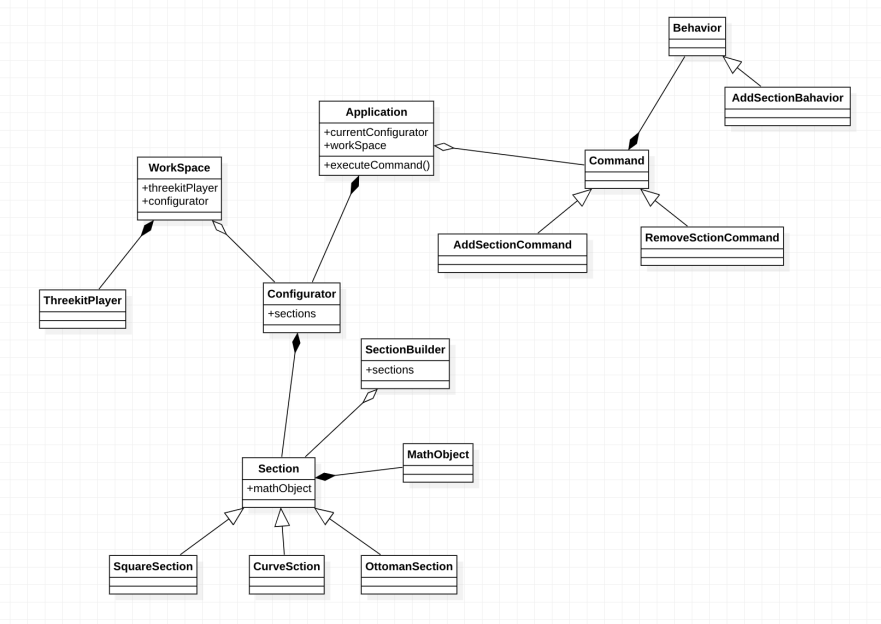
У нас появилась новая сущность — **Behavior**, которая позволяет **наделять команды дополнительной логикой (поведением)**. В рамках примера я создал **AddSectionBehavior** — сущность, описывающую **специфическое поведение** для команды **AddSectionCommand**.

На данном этапе мы рассмотрели **все сущности**, которые могут **расширять логику системы**, если это необходимо.

В следующих главах мы подробно разберем **ключевые сущности**, чтобы у тебя сложилось **четкое понимание** их предназначения и того, **какую логику стоит в них описывать**.

Глава 9. Уровни ответственности

Если ты посмотришь на диаграмму, которую мы нарисовали недавно, то что ты увидишь?



Я сейчас рассматриваю эту структуру с точки зрения того, что у нас есть некоторые слои, то есть уровни ответственности определенного модуля. Этот уровень ответственности влияет на то, насколько будут критичны изменения в системе, если мы этот модуль изменим или допишем какую-то логику. То есть, смотря на диаграмму, мы уже можем понять риски нового функционала и исправления существующего.

У нас есть две крайности — это высокий уровень и низкий уровень. Всё, что между, соответственно, зависит от того, к какой из крайностей он ближе. Те параметры ответственности он больше отражает.

Какой у нас самый высокий уровень?

Application — он объединяет все главные модули в себе и служит как бы внешним API для взаимодействия, например, в UI. То есть, если мы добавим что-то в **Application** или удалим, мы потенциально с малой долей вероятности сломаем приложение или какую-то логику.

Какой у нас самый низкий, или один из самых низких уровней?

Section — это единица, модуль, класс (по-разному можно называть), которая сама по себе включает состояние и поведение. Этот модуль может быть использован во многих местах. Как просто некоторая единица в списке, где будут храниться данные, так и как логика, которая может производить математические и алгоритмические вычисления.

Представим, что наш класс **Section** используется 30 раз в разных участках проекта, например, в Configurator, SectionBuilder, Command и т.д. Каждый из этих модулей включает в себя свою логику, которая использует логику **Section**.

Теперь представим, что мы в **Section** переписываем логику или изменяем какую-то часть. Насколько велики риски того, что наше приложение сломается или станет работать неправильно? Риск сломать что-то при внесении изменений в логику **Section** очень велик.

Какой средний уровень?

Configurator — это звено между двумя сущностями Application и Section. То есть, при изменении **Configurator** риск сломать приложение 50/50. Этот модуль включает в себя логику из низкоуровневых модулей, при этом сам их не реализовывает, а также используется в модулях более высокого уровня.

Вы будете правы, если скажете, что всё зависит от типа изменения. И риск сломать что-то зависит от того, что и как вы будете изменять. Если вы добавляете новые методы, например, в **Section**, чтобы их использовать потом где-то, то риск сломать что-то будет минимальным.

Это нужно понимать, но также не забывать о том, что риск сломать что-то всегда есть.

Глава 10. Детальный разбор сущностей

Configurator

Мы уже обсуждали, что такое **Configurator** в предыдущих главах, но я заметил, что люди зачастую, не зная, куда положить какую-то логику или состояние, запихивают их в классы или модули, где этого быть не должно. Поэтому давайте более подробно разберём, что такое **Configurator**.

Что такое Configurator?

Configurator — это сущность (класс), с помощью которой мы описываем истину нашего приложения. То есть данные, которые там хранятся, являются истиной для всех действий нашего приложения.

Поэтому логично будет хранить, например, состояние о всех секциях, которые выбраны для дивана. То есть будет поле:

```
section: Section[];
```

которое будет содержать список всех выбранных секций для нашего дивана.

Основываясь на этом, мы можем добавить методы, которые будут добавлять или удалять секции из поля **section**.

Что ещё можно хранить в Configurator?

Вроде понятно, но не совсем ясно, что ещё можно хранить и добавлять. Давай подумаем в декларативном стиле, ведь основная идея всех этих сущностей и уровней заключается в их взаимосвязи. Если сущность использует внутри себя другие сущности, как в случае с **Section**, то логику мы должны реализовывать, основываясь на этом.

Например, каждая секция имеет цену. Мы можем реализовать метод, который будет возвращать общую стоимость дивана, то есть пробежится по всем секциям и просуммирует их цены:

```
getTotalPrice(): number {
    return this.section.reduce((sum, sec) => sum + sec.price,
0);
}
```

Таким образом, мы можем добавлять логику, которая помогает обслуживать наши нужды.

Когда не стоит хранить логику в Configurator?

Но что, если нам нужно выделить секцию при клике? В таком случае **не стоит** хранить эту логику в **Configurator**, так как она временная и не является частью истины приложения. Хранение состояния и методов для такой логики в **Configurator** будет неверным решением.

WorkSpace

Этот модуль вызывает достаточно смутные мысли при написании логики, так как в него легко можно запихнуть логику, которая там не должна быть. Однако, если придерживаться декларативного стиля, то можно чётко определить, какие сущности должны взаимодействовать между собой. В нашем случае мы стремимся к этому, используя сущности более высокого уровня для объединения сущностей более низкого уровня.

WorkSpace как связующее звено

В данном случае **WorkSpace** имеет доступ как к **Configurator**, так и к **ThreekitPlayer**. То есть он является тем самым **средним звеном**, которое объединяет в себе разные сущности.

В предыдущей главе я приводил пример, что выделенную секцию курсором мышки не нужно хранить в **Configurator**, так как это временное состояние. Однако **WorkSpace** в этом случае подходит идеально. Таким образом, у нас будет поле:

```
selectedSections: Section[];
```

которое может хранить выделенные секции, а также методы, которые будут добавлять или удалять секции из этого поля.

Интеграция WorkSpace с ThreekitPlayer

Что примечательно, так как **WorkSpace** является связующим звеном, при добавлении или удалении секции мы можем вызывать логику из **ThreekitPlayer**, которая будет изменять отображение в 3D-плеере. Тем самым мы объединяем **3D-логику** с **истиной приложения**, сохраняя разделение между этими двумя сущностями.

Разделение логики

WorkSpace — это сущность, которая обслуживает нашу работу с истиной (**Configurator**) и с логикой **ThreekitPlayer**.

Важно понимать, что логика, касающаяся только **3D-отображения**, должна реализовываться в **ThreekitPlayer** или соответствующем модуле, который обслуживает **3D-логику**.

А, например, логику, которая работает **только с Section**, например, расчёт цены или количества подушек, доступных в диване, логично располагать в **Configurator**.

SectionBuilder

Эта сущность является специфической, так как включает в себя изолированную логику, которая будет использоваться во многих местах.

Основная концепция

Главная идея заключается в том, что, передавая на вход **одни и те же секции**, мы всегда должны получать **один и тот же диван**.

То есть, при добавлении секции происходит перерасчёт, и она подключается к определённой стороне дивана согласно заранее установленным правилам. Эти правила **всегда** работают одинаково и зависят **только от входных данных**.

Функциональность

Данная сущность может включать в себя следующую логику:

- **Добавление новой секции**

- **Удаление секции**
- **Пересчёт всех секций** в зависимости от произведённых действий

Таким образом, мы обеспечиваем строгую детерминированность системы, в которой изменения в секциях приводят к предсказуемому результату.

Command

Главная черта этой сущности заключается в том, что она должна **описывать логику** и то, как она **изменяет истину**в **Configurator**.

Например, такие функции, как:

- **Добавление секции**
- **Удаление секции**
- **Изменение материала**

Разделение логики

Важно отметить, что **Command** — это не просто сборище всей логики. Она не должна содержать всю бизнес-логику внутри себя.

Мы можем **разбивать логику на отдельные сущности**, такие как **SectionBuilder**, и использовать их внутри **Command**. Это делает код более модульным, читаемым и удобным для тестирования.

Behavior

Если вы задумываетесь, зачем нам нужен этот класс, то правильно делаете. Он входит в **дополнительный список сущностей**, которые **не являются обязательными**, но могут быть полезны в зависимости от **сложности** и **потребностей проекта**.

Пример использования

Предположим, у вас есть **валидация материалов**, которые можно применять к определённой секции. Эта валидация выполняется **на сервере заказчика**. Однако проверка должна происходить **именно в момент**, когда пользователь пытается применить материал к секции.

Важно сделать это **так, чтобы не нарушить команды**, которые уже были написаны.

Идеальным вариантом является **описание Behavior**, который:

1. **Отправляет запрос** к бэкенду
2. **Получает результат валидации**
3. **Прерывает выполнение команды**, если валидация не пройдена
4. **Позволяет продолжить выполнение**, если валидация успешна

Такой подход делает систему более гибкой и позволяет легко встраивать дополнительные проверки без изменения существующей логики команд

Заключение

Поздравляю! Ты добрался до конца!

Надеюсь, эта книга стала для тебя **путеводителем**, который не только показал пример мышления, но и помог взглянуть на проектирование программных систем с новой стороны.

Я сознательно допустил некоторые **упрощения** и **избегал слишком жёстких формулировок**, чтобы чтение и размышления над содержанием были комфортными.

Итог

Этот подход **не является идеальным**, как и любой другой. У него есть **преимущества** и **слабые стороны**, но я убеждён в одном: если ты **вдумчиво подходишь к задачам**, тщательно **продумываешь архитектуру** и выстраиваешь систему в логически структурированном виде, ты получаешь значительное преимущество.

Да, возможно, такая работа покажется излишней **в моменте**, но спустя год, возвращаясь к проекту, чтобы добавить новую функцию, ты осознаешь:

- Тебе **приятно** работать с этим кодом
- Ты **понимаешь** свою систему
- Ты можешь вносить изменения **с минимальными рисками**

Связь

Если ты хочешь оставить **отзыв** или просто связаться со мной, пиши на pproger@pproger.me. Буду рад твоим мыслям и обратной связи!