SOLID и шаблоны проектирования

Evgeny Markov

План лекции

1. SOLID
2. Порождающие шаблоны
3. Структурные шаблоны
4. Поведенческие шаблоны

Robert Martin

SOLID

S Принцип единственной ответственности

O Принцип открытости-закрытости

L Принцип подстановки Барбары Лисков

I Принцип разделения интерфейса

D Принцип инверсии зависимостей

Принцип единственной ответственности (SRP)

Существует лишь одна причина, приводящая к изменению класса

Неправильно

```
interface IUser {
  id: number;

  name: string;
}
```

Неправильно

```
class UserController {
  getUserInfo(id: number): string {
    const user = this.getUserFromDb(id);

    return this.formatToHtml(user);
  }

  protected getUserFromDb(id: number): IUser {
    // SELECT id, name FROM users WHERE users.id = ${id};

    return { id: 1, name: 'Martin' };
  }

  protected formatToHtml(user: IUser): string {
    return `<h1>Name: ${user.name} </h1>`;
  }
}
```

Правильно

```
class UserRepository {
  getUserById(id: number): IUser {
    // SELECT id, name FROM users WHERE users.id = ${id};

    return { id: 1, name: 'Martin' };
  }
}

class UserFormatter {
  formatToHtml(user: IUser): string {
    return `<h1>Name: ${user.name} </h1>`;
  }
}
```

Правильно

```
class UserController {
  protected repository = new UserRepository();
  protected formatter = new UserFormatter();

  getUserInfo(id: number): string {
    const user = this.repository.getUserById(id);

    return this.formatter.formatToHtml(user);
  }
}
```

Принцип открытости-закрытости (OCP)

Программные сущности должны быть открыты для расширения, но закрыты для модификации

Неправильно

```
interface IUser {
  id: number;
  name: string;
}

interface IRegisteredUser extends IUser {
  email: string;
}
```

Неправильно

```
class UserFormatter {
  formatToHtml(user: IUser | IRegisteredUser): string {
    if ('email' in user) {
      return `
        <p>Name: ${user.name} </p>
        <br>
        <p>Email: ${user.email} </p>
      `;
    }

    return `<p>Name: ${user.name} </p>`;
  }
}
```

Правильно

```
interface IUser {
  id: number;
  name: string;
  getDisplayFields(): string[];
}

interface IRegisteredUser extends IUser {
  email: string;
}
```

Правильно

```
class UserFormatter {
  formatToHtml(user: IUser): string {
    return user
      .getDisplayFields()
      .map(field => `<p>${field}: ${user[field]}</p>`)
      .join('<br>');
  }
}
```

Принцип подстановки Барбары Лисков (LSP)

Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом

LSP

```
abstract class BaseUserRepository {
  abstract getById(id: number): IUser;
}

class MockUserRepository extends BaseUserRepository {
  getById(id: number): IUser {
    return { id, name: 'Robert' };
  }
}

class FileUserRepository extends BaseUserRepository {
  getById(id: number): IUser {
    const fileContent = fs.readFileSync('users.json', 'utf-8');
    const user = JSON.parse(fileContent)[id];

    return user;
  }
}

// ...
```

LSP

```
class UserController {
  protected repository: BaseUserRepository;

  constructor() {
    switch (process.env.NODE_ENV) {
      case 'development':
        this.repository = new FileUserRepository();
        break;
      case 'test':
        this.repository = new MockUserRepository();
        break;
      default:
        this.repository = new DbUserRepository();
    }
  }

  getUserInfo(id: number): IUser {
    return this.repository.getById(id);
  }
}
```

Принцип разделения интерфейса (ISP)

Нельзя заставлять клиента реализовать интерфейс, которым он не пользуется

Неправильно

```
interface IRepository<T> {
  findAll(): Promise<T[]>;
  findOne(id: string): Promise<T>;
  create(item: T): Promise<boolean>;

  delete(id: string): Promise<boolean>;
  update(id: string, item: T): Promise<boolean>;
}

class MockUserRepository implements IRepository<IUser> {
  findAll(): Promise<IUser[]> {
    return Promise.resolve([{ id: 1, name: 'Sergey' }, { id: 2, name: 'Maxim' }]);
  }

  // ...

  update(id: string, item: IUser): Promise<boolean> {
    throw new Error('Method not implemented.');
  }
}
```

Правильно

```
interface IReadableRepository<T> {
  findAll(): Promise<T[]>;
  findOne(id: string): Promise<T>;
}


interface IWritableRepository<T> {
  create(item: T): Promise<boolean>;
  delete(id: string): Promise<boolean>;
  update(id: string, item: T): Promise<boolean>;
}

class MockUserRepository implements IReadableRepository<IUser> {
  findAll(): Promise<IUser[]> {
    return Promise.resolve([{ id: 1, name: 'Sergey' }, { id: 2, name: 'Maxim' }]);
  }

  findOne(id: string): Promise<IUser> {
    return Promise.resolve({ id: 3, name: 'Igor' });
  }
}
```

Принцип инверсии зависимостей (DIP)

Высокоуровневые модули не должны зависеть от низкоуровневых. Оба вида модулей должны зависеть от абстракций.

Абстракции не должны зависеть от подробностей. Подробности должны зависеть от абстракций.

Неправильно

```
class MySQLConnection {
  query(sql: string): object[] {
    // Execute SQL query


    return [];
  }
}

class DbUserRepository {
  protected dbConnection: MySQLConnection;

  constructor(dbConnection: MySQLConnection) {
    this.dbConnection = dbConnection;
  }

  findAll() {
    return this.dbConnection.query('SELECT * FROM users;');
  }
}
```

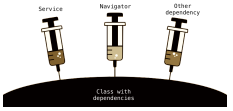
Правильно

```
interface IConnection {
  query(sql: string): object[];
}


class MySQLConnection implements IConnection {
  query(sql: string): object[] {
    // Execute SQL query written in MySQL dialect

    return [];
  }
}

class PostgreSQLConnection implements IConnection {
  query(sql: string): object[] {
    // Execute SQL query written in PostgreSQL dialect

    return [];
  }
}
```

Правильно

```
class DbUserRepository {
  protected dbConnection: IConnection;

  constructor(dbConnection: IConnection) {

    this.dbConnection = dbConnection;
  }

  findAll() {
    return this.dbConnection.query('SELECT * FROM users;');
  }
}

const userRepositoryMySQL = new DbUserRepository(
  new MySQLConnection()
);
const userRepositoryPostgreSQL = new DbUserRepository(
  new PostgreSQLConnection()
);
```

Dependency Injection



- Основы внедрения зависимостей
- inversify/InversifyJS
- Microsoft/tsyringe
- thiagobustamante/typescript-ioc
- bem/bem-react

Шаблоны проектирования

Это типичные способы решения часто встречающихся проблем при проектировании программ

Шаблоны проектирования

- Порождающие
- Структурные
- Поведенческие

Порождающие шаблоны

Это шаблоны, которые абстрагируют процесс создания объектов классов

- Фабрика (Factory)
- Строитель (Builder)
- Одиночка (Singleton)
- ...

Фабрика (Factory)

Это объект или функция для создания других объектов

Фабрика (Factory)

```
interface IStorage {
  get(key: string): string;
  set(key: string, value: string): void;
}

enum StorageType {
  Redis,
  InMemory
}

class RedisStorage implements IStorage {
  get(key: string): string { /* ... */ }
  set(key: string, value: string): void { /* ... */ }
}

class InMemoryStorage implements IStorage {
  get(key: string): string { /* ... */ }
  set(key: string, value: string): void { /* ... */ }
}
```

Фабрика (Factory)

```
function createStorage(type: StorageType): IStorage {
  switch (type) {
    case StorageType.Redis:
      return new RedisStorage();
    case StorageType.InMemory:
      return new InMemoryStorage();
    default:
      throw new Error('Invalid storage type');
  }
}
```

Структурные шаблоны

Это шаблоны, которые отвечают за построение удобных в поддержке иерархий классов

Структурные шаблоны

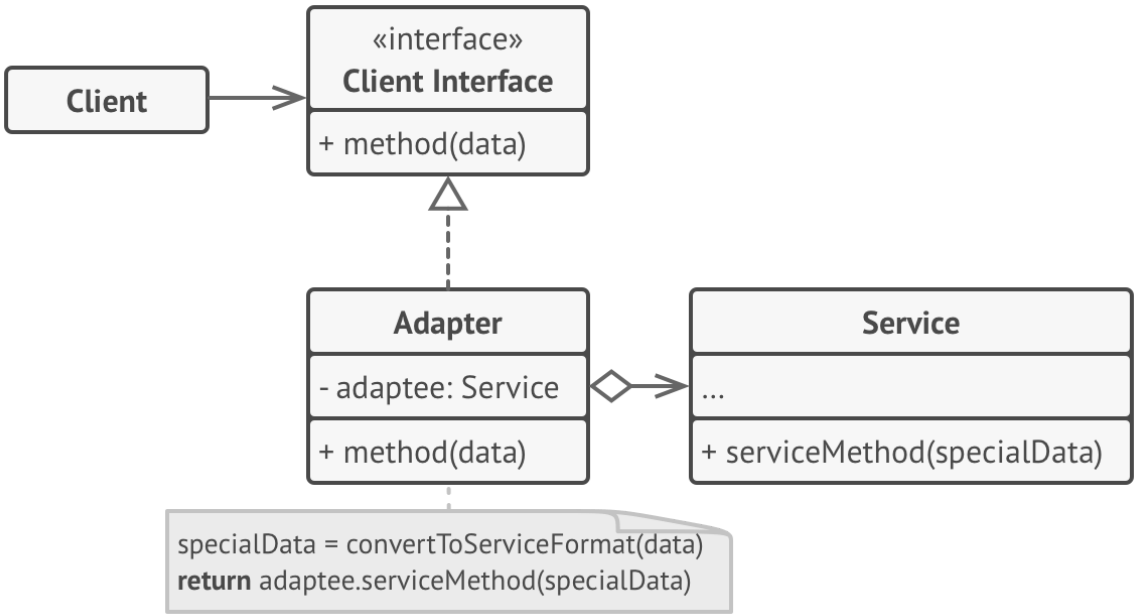
- Адаптер (Adapter)

- Заместитель (Proxy)
- Декоратор (Decorator)
- Компоновщик (Composite)
- Фасад (Facade)
- Мост (Bridge)

Адаптер (Adapter)

Это шаблон, который позволяет объектам с несовместимыми интерфейсами работать вместе

Адаптер (Adapter)



Адаптер (Adapter)

```
interface ILogRecord {
  level: string;
  timestamp: number;
  message: string;
}

class LoggerService {
  sendRecord(record: ILogRecord) {
    console.log(record);
  }
}
```

Адаптер (Adapter)

```
interface ILogger {
  log(message: string): void;
}


class Client {
  protected logger: ILogger;

  constructor(logger: ILogger) {
    this.logger = logger;
  }

  doSomething() {
    this.logger.log('DEBUG 1556523689224 Client works fine');
  }
}
```

Адаптер (Adapter)

```
class ServiceAdapter implements ILogger {
  protected service = new LoggerService();


  log(message: string): void {
    const match = message.match(/(\w+) (\w+) (.*)/);

    if (match) {
      this.service.sendRecord({
        level: match[1],
        timestamp: Number(match[2]),
        message: match[3]
      });
    }
  }
}
```

Поведенческие шаблоны

Это шаблоны, определяющие алгоритмы и способы реализации взаимодействия различных объектов и классов

Поведенческие шаблоны

- Наблюдатель (Observer)
- Стратегия (Strategy)
- Цепочка обязанностей (Chain of responsibility)
- Посетитель (Visitor)
- Итератор (Iterator)
- Интерпретатор (Interpreter)
- ...

Стратегия (Strategy)

Это шаблон, который определяет семейство схожих алгоритмов, инкапсулирует каждый из них и обеспечивает их взаимозаменяемость

Стратегия (Strategy)

```
interface IAuthStrategy {
  authenticate(): void;
}


class TokenAuthStrategy implements IAuthStrategy {
  authenticate() {
    console.log('Authenticating using TokenAuthStrategy');
  }
}

class CookiesAuthStrategy implements IAuthStrategy {
  authenticate() {
    console.log('Authenticating using CookiesAuthStrategy');
  }
}
```

Стратегия (Strategy)

```
class Passport {
  protected strategy: IAuthStrategy | null = null;


  use(strategy: IAuthStrategy) {
    this.strategy = strategy;

    return this;
  }

  authenticate() {
    if (this.strategy === null) {
      throw new Error('No authentication strategy set');
    }

    this.strategy.authenticate();
  }
}
```

Стратегия (Strategy)

```
// Setup
const passport = new Passport();
const cookiesStrategy = new CookiesAuthStrategy();
passport.use(cookiesStrategy);

// Usage
passport.authenticate();
```

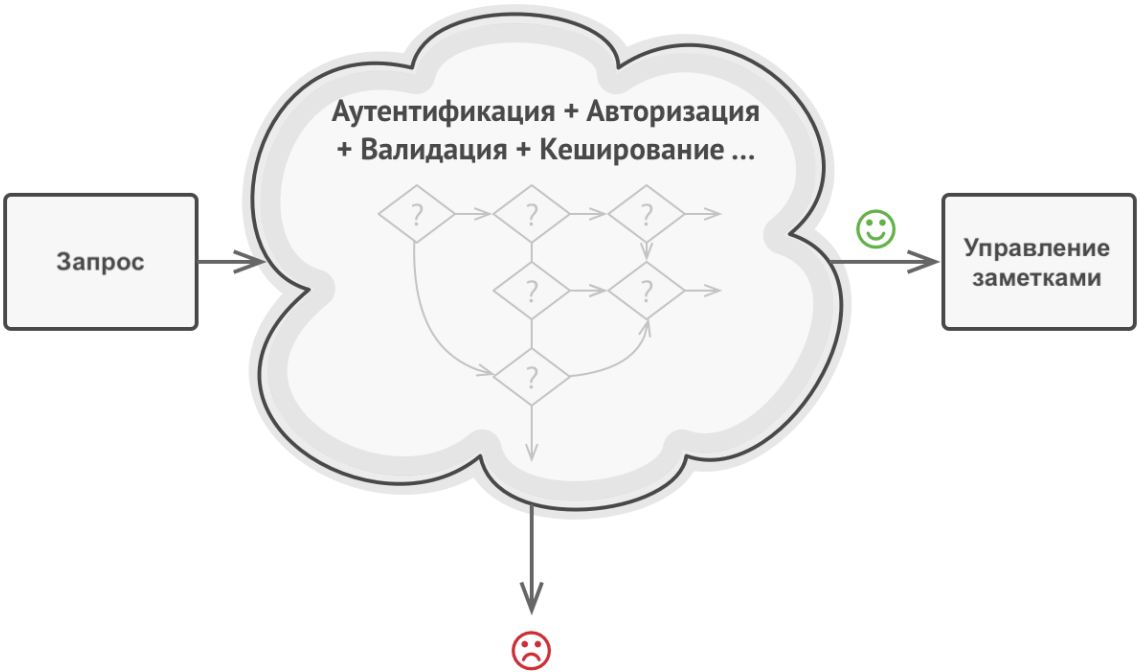
Цепочка обязанностей (Chain of responsibility)

Шаблон проектирования, который позволяет избежать жёсткой привязки отправителя запроса к получателю, позволяя нескольким объектам обработать запрос

Проблема


Запрос
Аутентификация +
Авторизация
?
?
Управление
заметками

Проблема усугубляется



Решение

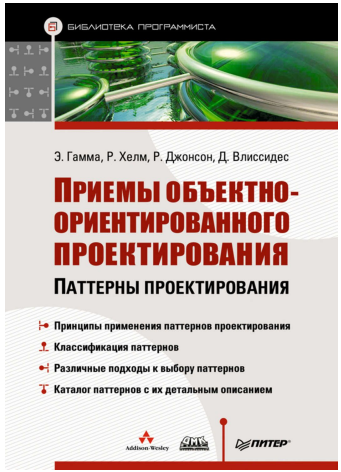

Цепочка обязанностей
Запрос
Обработчик
Обработчик
Обработчик
Управление заметками

Анти-паттерны

Почитать


ЧИСТЫЙ КОД
СОЗДАНИЕ, АНАЛИЗ И РЕФАКТОРИНГ
РОБЕРТ МАРТИН


Эрик Фримен, Элизабет Фримен
при участии Кэтти Сьерра и Берта Бейтса
ПАТТЕРНЫ проектирования
Избегайте нелепых ошибок наследования
Узнайте, почему все, что вы знали о паттернах, возможно, ошибочно
Узнайте секреты гуру проектирования паттернов
Усвойте теорию паттернов проектирования максимально эффективно
Изучите сотни примеров и практических упражнений
Посмотрите, как улучшается жизнь программиста благодаря паттернам
O'REILLY
ПИТЕР


БИБЛИОТЕКА ПРОГРАММИСТА
Э. Гамма, Р. Хелм, Р. Джонсон, Д. Влиссидес
ПРИЕМЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОЕКТИРОВАНИЯ
Паттерны проектирования
Принципы применения паттернов проектирования
Классификация паттернов
Различные подходы к выбору паттернов
Каталог паттернов с их детальным описанием
Addison-Wesley
ПИТЕР

Демо

Приложение с аутентификацией

Спасибо!

Вопросы?