

4. Отношения между классами и объектами. Диаграммы UML

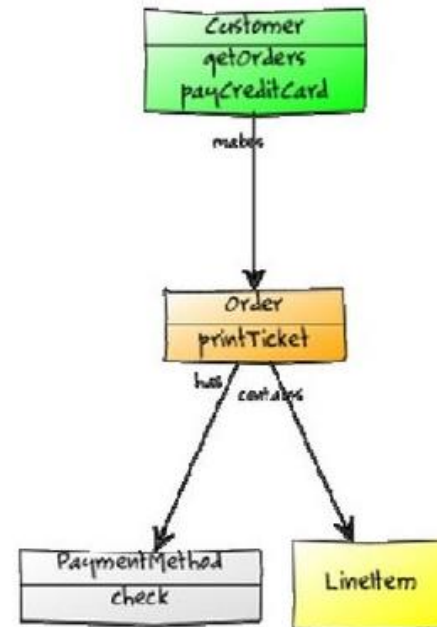
Процедурный подход



Программа – алгоритм последовательного вызова процедур изменения данных в памяти.

Какие функции создавать и как они будут взаимодействовать друг с другом?

Объектно-ориентированный



Программа – взаимодействие объектов, компонентов, отсылка и обработка событий.

Какие классы создавать и как их связывать друг с другом?

Причины создания классов

- Моделирование объектов реального мира
- Моделирование абстрактных объектов
- Хранение конфигурации для выполнения действий
- Соккрытие глобальных данных
- Упаковка родственных операций

Моделирование объектов реального мира

Классы помогают описать сущности (машина, пользователь) с их свойствами и действиями.

php

```
1 class Car {
2     public $brand;
3     public $model;
4     public $year;
5
6     public function startEngine() {
7         return "Двигатель $this->brand запущен!";
8     }
9 }
10
11 $car = new Car();
12 $car->brand = "Toyota";
13 echo $car->startEngine(); // "Двигатель Toyota запущен!"
```

Моделирование абстрактных объектов

Классы могут представлять абстракции (профиль пользователя, банковский счет).

php

```
1 class UserProfile {
2     private $username;
3     private $email;
4
5     public function __construct($username, $email) {
6         $this->username = $username;
7         $this->email = $email;
8     }
9
10    public function updateEmail($newEmail) {
11        $this->email = $newEmail;
12    }
13 }
14
15 $user = new UserProfile("alex", "alex@example.com");
16 $user->updateEmail("new@example.com");
```

Хранение конфигурации

Классы централизуют настройки (базы данных, API-ключи).

```
php
1 class DatabaseConfig {
2     private $config = [
3         'host' => 'localhost',
4         'user' => 'root',
5         'password' => '',
6         'dbname' => 'mydb'
7     ];
8
9     public function getConfig() {
10         return $this->config;
11     }
12 }
13
14 $dbConfig = new DatabaseConfig();
15 print_r($dbConfig->getConfig());
```

Соккрытие глобальных данных

Классы инкапсулируют данные, защищая их от внешнего вмешательства.

```
php
1 class Logger {
2     private static $instance;
3     private $logs = [];
4
5     private function __construct() {}
6
7     public static function getInstance() {
8         if (!$self::$instance) {
9             self::$instance = new Logger();
10        }
11        return self::$instance;
12    }
13
14    public function log($message) {
15        $this->logs[] = $message;
16    }
17 }
18
19 // Использование:
20 Logger::getInstance()->log("Ошибка!");
```

Упаковка родственных операций

Классы группируют связанные функции (математические операции, обработка строк).

```
php
1 class MathOperations {
2     public static function add($a, $b) {
3         return $a + $b;
4     }
5
6     public static function multiply($a, $b) {
7         return $a * $b;
8     }
9 }
10
11 echo MathOperations::add(2, 3); // 5
12 echo MathOperations::multiply(2, 3); // 6
```

Виды объектов

- Объекты-сущности
- Объекты-значения

Эти понятия помогают структурировать код и моделировать предметную область.

Объекты-сущности

Сущности — это объекты, которые имеют уникальную идентичность (ID), сохраняющуюся даже при изменении их свойств.

Есть **идентификатор** объекта, который отличает его от других объектов, даже если остальные данные совпадают.

- Пользователь в системе (с уникальным id)
- Товар в интернет-магазине (с уникальным артикулом)
- Банковский счет (с уникальным номером)

Объекты-сущности обычно изменяемы (смена фамилии)

Объекты-сущности

```
class User {
    private $id;
    private $name;
    private $email;

    public function __construct(string $id, string $name, string $email) {
        $this->id = $id;
        $this->name = $name;
        $this->email = $email;
    }

    // Геттер для ID
    public function getId(): string {
        return $this->id;
    }

    // Два пользователя считаются одинаковыми, если совпадают их ID
    public function equals(User $other): bool {
        return $this->id === $other->getId();
    }
}

// Использование
$user1 = new User("1", "Alice", "alice@example.com");
$user2 = new User("1", "Alice Smith", "alice_new@example.com");

var_dump($user1->equals($user2)); // true (ID одинаковые)
```

Объекты-значения

Объекты-значения определяются своими атрибутами , а не идентификатором. Если два объекта имеют одинаковые данные — они считаются равными .

- Деньги (сумма и валюта: 100 USD)
- Дата (2024-01-01)
- Адрес (город, улица, дом)

Объекты-значения обычно неизменяемы (иммутабельны)

```
class Money {
    private $amount;
    private $currency;

    public function __construct(float $amount, string $currency) {
        $this->amount = $amount;
        $this->currency = $currency;
    }

    // Геттеры
    public function getAmount(): float {
        return $this->amount;
    }

    public function getCurrency(): string {
        return $this->currency;
    }

    // Сравнение по значениям
    public function equals(Money $other): bool {
        return $this->amount === $other->getAmount()
            && $this->currency === $other->getCurrency();
    }
}

// Использование
$money1 = new Money(100, "USD");
$money2 = new Money(100, "USD");
$money3 = new Money(100, "EUR");

var_dump($money1->equals($money2)); // true (одинаковые значения)
var_dump($money1->equals($money3)); // false (разные валюты)
```

Сущности и значения

- **Сущности** помогают работать с объектами, которые должны быть уникальными в системе (например, пользователь с id=123).
- **Значения** упрощают работу с данными, где важны только их характеристики (например, сумма денег).
- Разделение на сущности и значения делает код более предсказуемым и соответствует принципам ООП.
- Разделение помогает избежать ошибок, связанных с некорректным сравнением или изменением объектов.

Монолитный подход vs разделение ответственностей

Что лучше?

- Вся бизнес-логика приложения реализуется в одном большом классе.
- В приложении описаны несколько классов, которые могут быть вложены друг в друга.

Принцип единственной ответственности

Каждый класс должен иметь одну чёткую задачу .
Если бизнес-логика "размазана" по огромному классу, это приводит к:

- Сложности в понимании кода.
- Риску поломать что-то при изменении одной функции.
- Повторяющемуся коду.

Принцип единственной ответственности

Пример без композиции (плохо):

php

```
1 class Order {
2     // Отвечает за заказ, оплату, уведомления и доставку
3     public function processPayment() { /* ... */ }
4     public function sendEmailNotification() { /* ... */ }
5     public function updateInventory() { /* ... */ }
6     public function generateShippingLabel() { /* ... */ }
7 }
```

Принцип единственной ответственности

- С композицией (хорошо):
- Каждый класс (`PaymentService` , `EmailService`) отвечает за свою зону.
 - `Order` только координирует работу, а не реализует всё сам.

php

```
1 class Order {
2     private $paymentService;
3     private $emailService;
4     private $shippingService;
5
6     public function __construct(
7         PaymentService $paymentService,
8         EmailService $emailService,
9         ShippingService $shippingService
10    ) {
11        $this->paymentService = $paymentService;
12        $this->emailService = $emailService;
13        $this->shippingService = $shippingService;
14    }
15
16    public function process() {
17        $this->paymentService->charge();
18        $this->emailService->notifyCustomer();
19        $this->shippingService->createLabel();
20    }
21 }
```

Повторное использование кода

Если логика вынесена в отдельные объекты, их можно использовать в разных частях приложения

```
// Сервис отправки email может использоваться в заказах, регистрациях и т.д.  
$emailService = new EmailService();  
  
$order = new Order($paymentService, $emailService, $shippingService);  
$userRegistration = new UserRegistration($emailService);
```

Удобство тестирования

Композиция позволяет мокать зависимости (заменять их на тестовые реализации).

Пример теста без композиции (плохо):

php

```
1 // Тестирование метода process() в монолитном Order
2 // Придётся мокать внутренние вызовы оплаты, email и доставки внутри класса
3 // Это сложно и хрупко
```

С композицией (хорошо):

php

```
1 // Внедряем моки через конструктор
2 $mockPayment = $this->createMock(PaymentService::class);
3 $mockEmail = $this->createMock(EmailService::class);
4
5 $order = new Order($mockPayment, $mockEmail, $shippingService);
6 $order->process();
7
8 // Проверяем, что методы были вызваны
9 $mockPayment->expects($this->once())->method('charge');
10 $mockEmail->expects($this->once())->method('notifyCustomer');
```

Гибкость и расширяемость кода

Без композиции:

```
php
1 class Order {
2     public function processPayment(string $method) {
3         if ($method === 'paypal') { /* ... */ }
4         else { /* ... */ }
5     }
6 }
7 // При добавлении нового метода приходится править существующий код
```

С композицией:

```
php
1 interface PaymentService {
2     public function charge();
3 }
4
5 class PayPalService implements PaymentService { /* ... */ }
6 class StripeService implements PaymentService { /* ... */ }
7
8 // Внедряем нужный сервис в Order
9 $order = new Order(new PayPalService(), $emailService, $shippingService);
```

Слабая связанность

Композиция снижает зависимость между компонентами. Например, если изменится логика отправки `email`, это не затронет класс `Order`.

```
// Если EmailService изменится (например, добавится API-ключ),  
// исправления нужны только в нём самом, а не в Order  
class EmailService {  
    private $apiKey;  
  
    public function __construct(string $apiKey) {  
        $this->apiKey = $apiKey;  
    }  
}
```

Читаемость и поддержка

Код с композицией проще понять:

- Каждый класс решает конкретную задачу.
- Зависимости явно указаны (например, в конструкторе).
- Легко отслеживать, где используется та или иная функция.

Внедрение объектов и композиция

- Упрощают поддержку кода.
- Повышают переиспользование компонентов.
- Облегчают тестирование.
- Снижают риски при внесении изменений.

Отношения между классами

Отношения между классами

- **Наследование**

Кошка является животным

- **Ассоциация**

В свойстве класса содержится ссылка на экземпляр(ы) другого класса

- **Композиция**

Квартира имеет комнату.

- **Агрегация**

Автомобиль имеет колесо.

Агрегация и композиция



Отношение
агрегации



Отношение
КОМПОЗИЦИИ

Композиция

Объект А управляет временем жизни объекта В

```
class Engine
{
    private $power;

    public function __construct($power)
    {
        $this->power = $power;
    }
}

class Car
{
    private $model;
    private $engine;

    public function __construct($model, $power)
    {
        $this->model = $model;
        $this->engine = new Engine($power);
    }
}

$myCar = new Car( model: "Saturn VUE", power: 160);
```

Плюсы

- Скрывает отношения использования объекта от глаз клиента.
- Делает API использования класса более простым.

Минусы

- Отношение жесткое (один объект должен знать конкретный тип и иметь доступ к функции создания другого объекта).

Агрегация

Объект А получает ссылку на объект В

```
class Engine
{
    private $power;

    public function __construct($power)
    {
        $this->power = $power;
    }
}

class Car
{
    private $model;
    private $engine;

    public function __construct($model, $engine)
    {
        $this->model = $model;
        $this->engine = $engine;
    }
}

$myCar = new Car( model: "Saturn VUE", new Engine( power: 160));
```

Плюсы

- Более слабая связанность между объектом и его клиентом. Объекту не нужно знать, как именно создавать другой объект.
- Большая гибкость.

Минусы

- Выставление наружу деталей реализации, увеличение сложности в работе клиентов.
- «Целое» не может самостоятельно заменить «составную часть».

Различие между композицией и агрегацией

	Композиция	Агрегация
Жизненный цикл	Дочерний объект живет столько же времени, сколько и родительский	Объекты живут независимо друг от друга
Контроль над зависимостями	Владелец контролирует создание и удаление своих частей	Владелец просто использует другие объекты без контроля над их созданием и удалением
Зависимость	Объект жестко привязан к своему владельцу	Связь менее жесткая, так как объект может принадлежать нескольким владельцам одновременно
Уровень абстракции	Часто используется для реализации внутренней структуры объекта	Применяют для объединения нескольких независимых сущностей

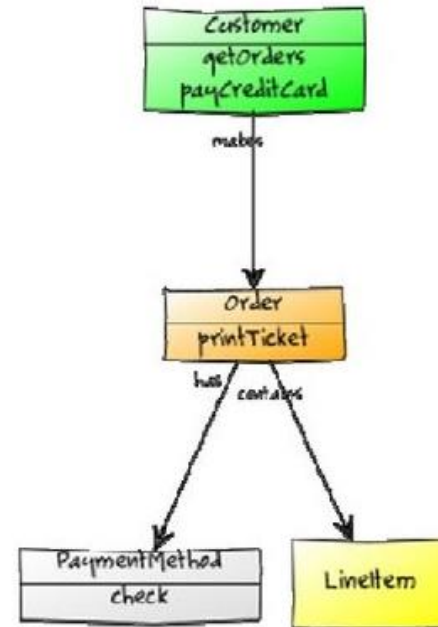
Взаимодействие объектов

Процедурный подход



Программа – алгоритм последовательного вызова процедур изменения данных в памяти.

Объектно-ориентированный



Программа – взаимодействие объектов, компонентов, отсылка и обработка событий.

Объекты – «кирпичи» для построения приложения.

Взаимодействие объектов



Запросить текущее состояние
(вызов метода-запроса)

- **Ответить на запрос**
(`return ...`)
- **Не ответить на запрос**
(`exception ...`)
- **Делегировать запрос**
(вызов метода-запроса)

Взаимодействие объектов



Указать выполнить действие
(вызов метода-команды)

- **Выполнить действие**
- **Отказаться, назвав причину**
(exception ...)
- **Прервать выполнение**
(exception ...)
- **Записать задачу в ежедневник**
(поставить в очередь)
- **Сообщить о выполнении**
(событие)
- **Делегировать приказ**
(вызов метода-команды)

UML-диаграммы

UML (Unified Modeling Language)

UML — это стандартный язык визуального моделирования, используемый для проектирования, документирования и анализа программных систем.

Он предоставляет набор графических элементов (диаграмм), которые помогают описать структуру, поведение и взаимодействие компонентов системы.

Создание UML

UML был разработан в середине 1990-х годов тремя ведущими экспертами в области ООП:

- Гради Буч (Grady Booch) — автор метода Booch.
- Айвар Якобсон (Ivar Jacobson) — создатель Use Case диаграмм.
- Джеймс Рамбо (James Rumbaugh) — разработчик метода ОМТ.

В 1997 году UML был принят в качестве стандарта организацией OMG (Object Management Group).

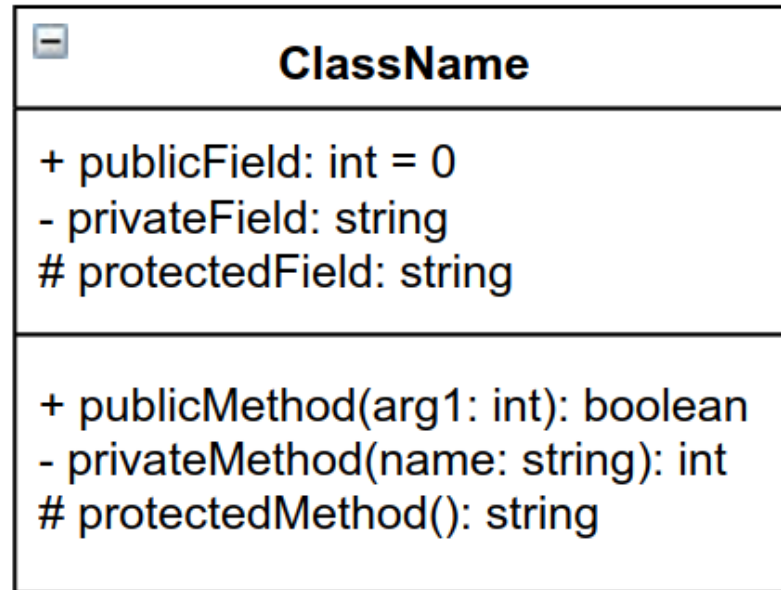
Основные цели UML

- **Стандартизация.** UML объединил в единый язык несколько конкурирующих методов моделирования.
- **Визуализация сложных систем.** Помогает представить систему в виде понятных диаграмм, что упрощает обсуждение требований с заказчиками, проектирование архитектуры, обучение новых сотрудников.
- **Поддержка ООП.** Позволяет описать классы, объекты, наследование, ассоциации и другие концепции.
- **Документирование.** Диаграммы UML служат "чертежами" системы, что упрощает её поддержку и развитие.

UML-диаграммы

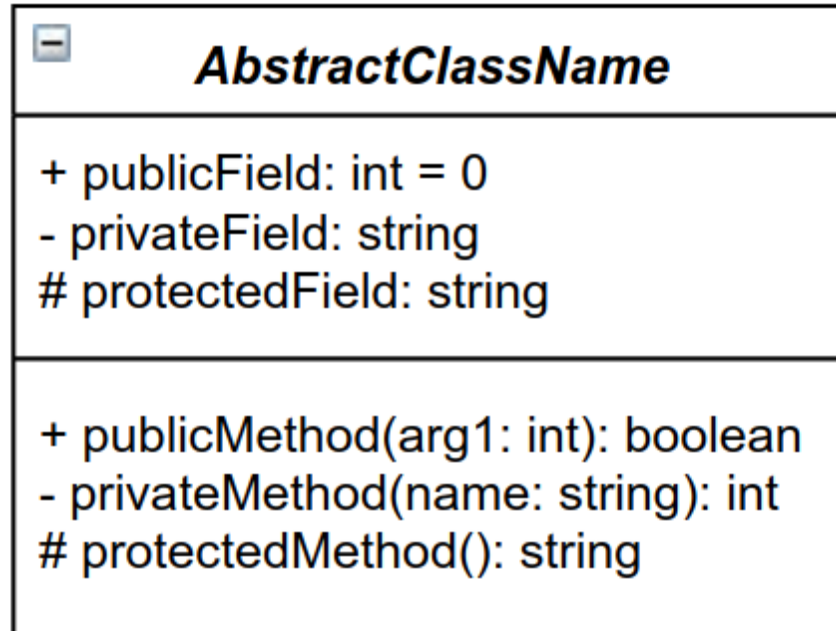
- Диаграмма классов (class diagram)
- Диаграмма использования (use case diagram)
- Диаграмма автомата (stat machine diagram)
- Диаграмма деятельности (activity diagram)
- Диаграмма последовательности (sequence diagram)
- Диаграмма коммуникации (communicatio diagram)

Диаграммы классов



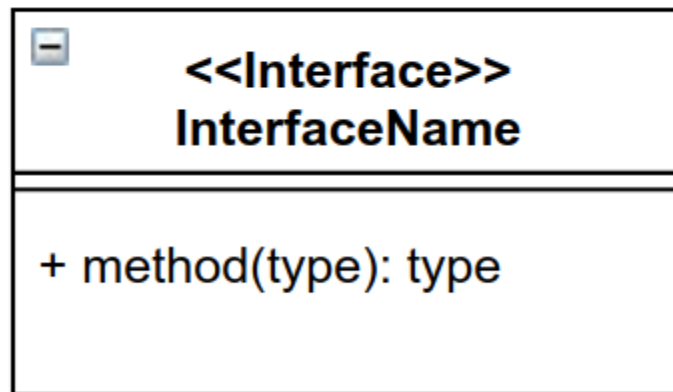
Класс с атрибутами и операциями

Диаграммы классов



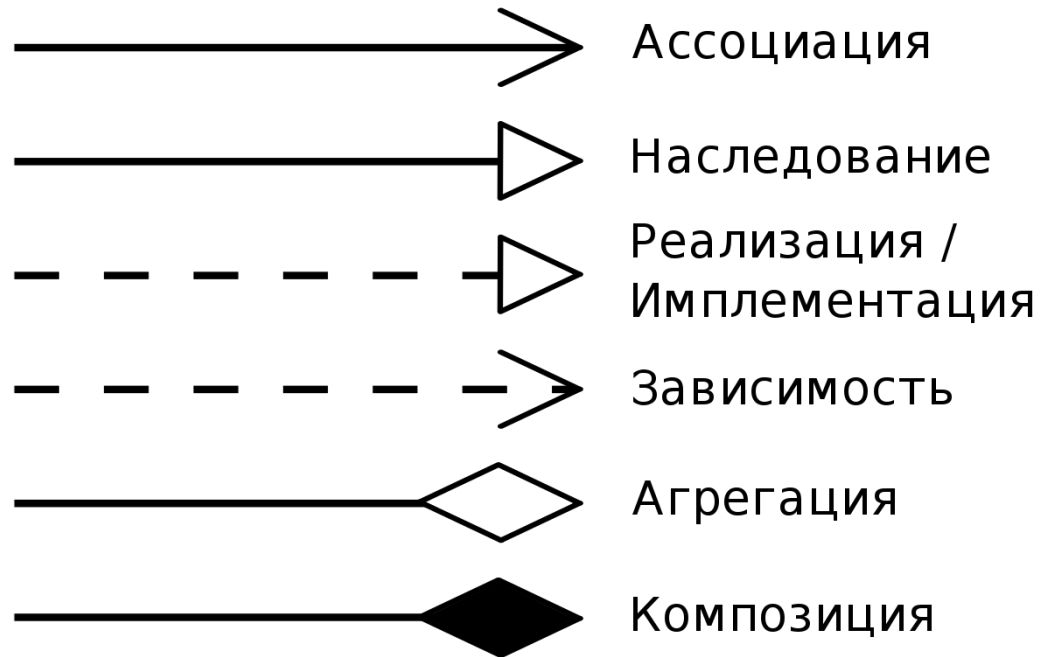
Абстрактный класс

Диаграммы классов



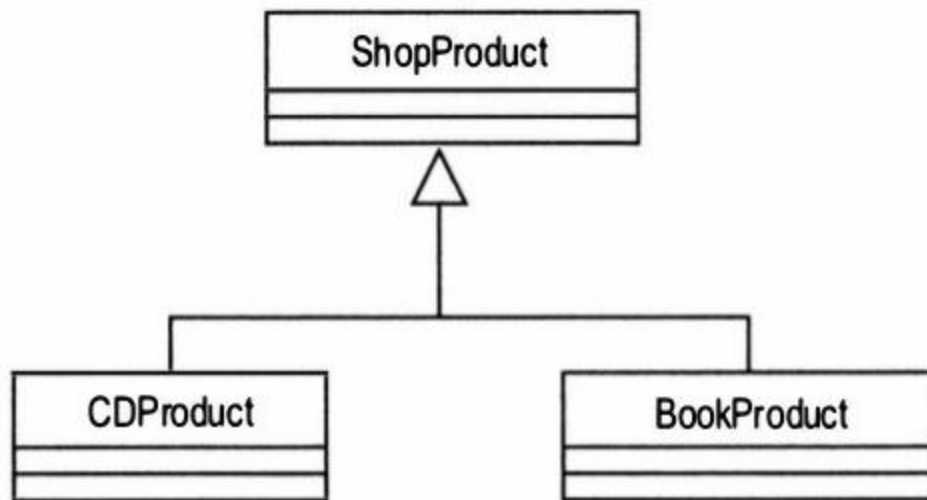
Интерфейс

Диаграммы классов



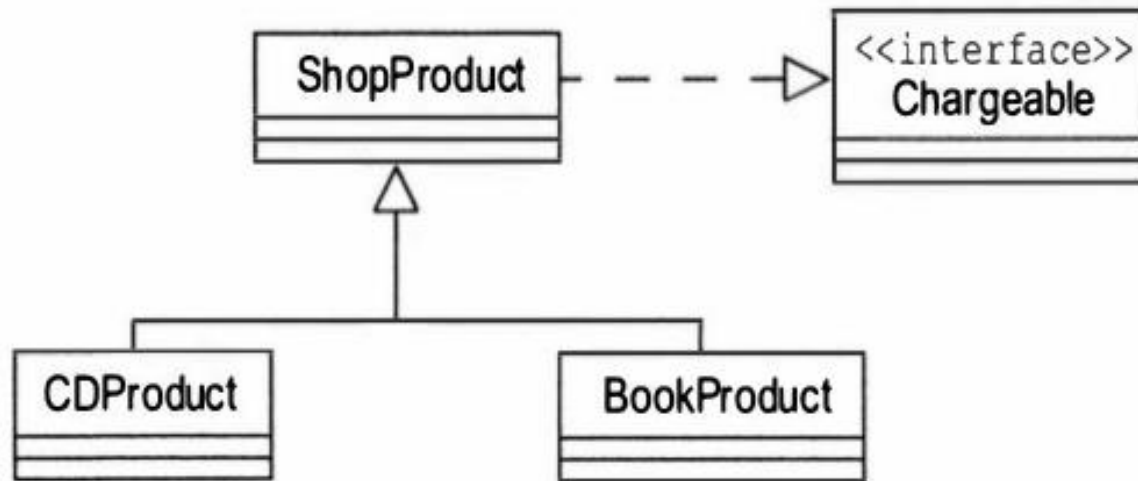
Отношения между классами

Диаграммы классов



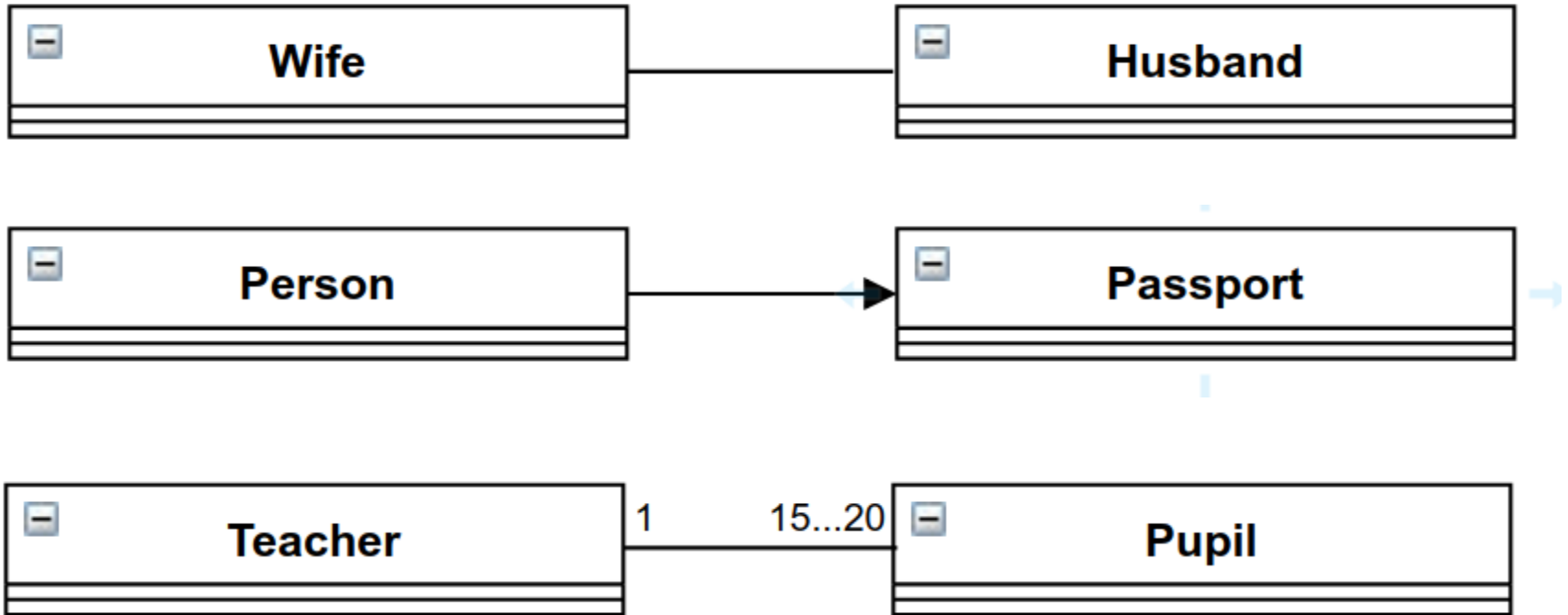
Отношение наследования (обобщение)

Диаграммы классов



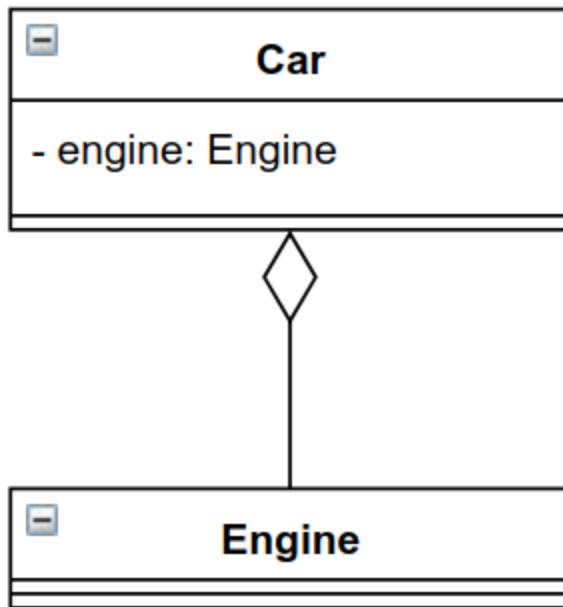
Отношение реализации

Диаграммы классов



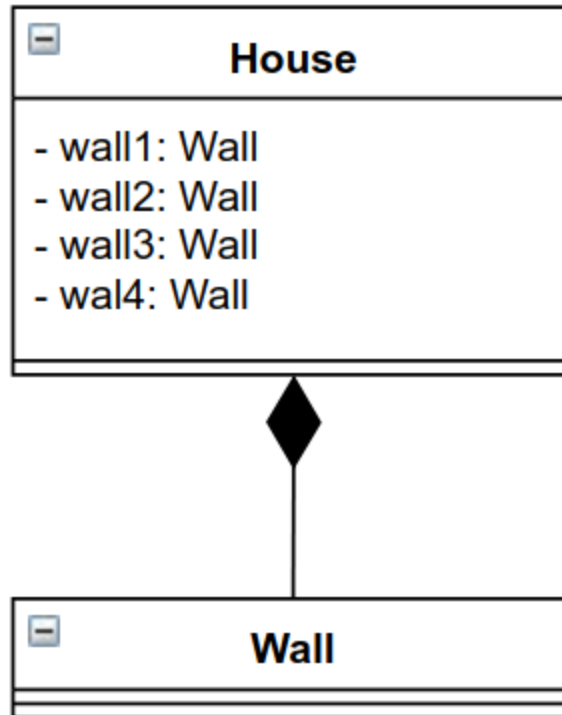
Отношение ассоциации

Диаграммы классов



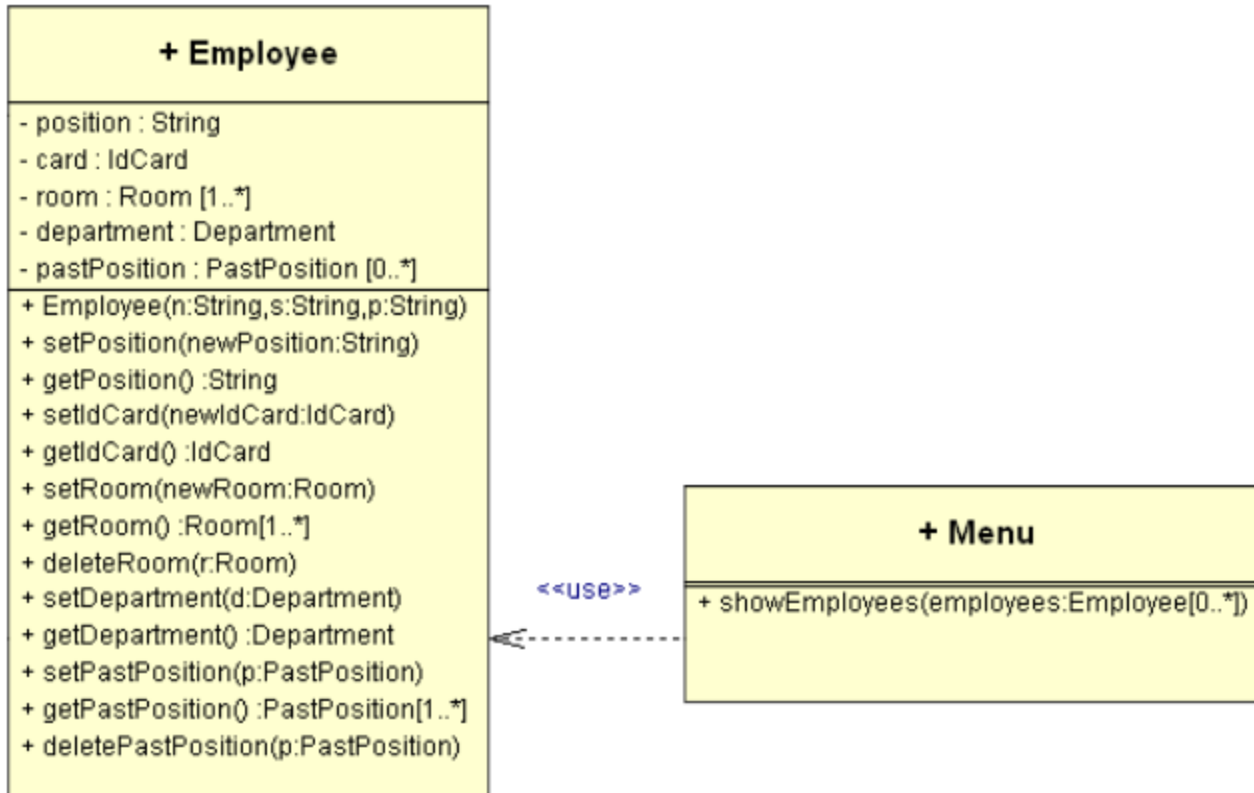
Отношение агрегации

Диаграммы классов



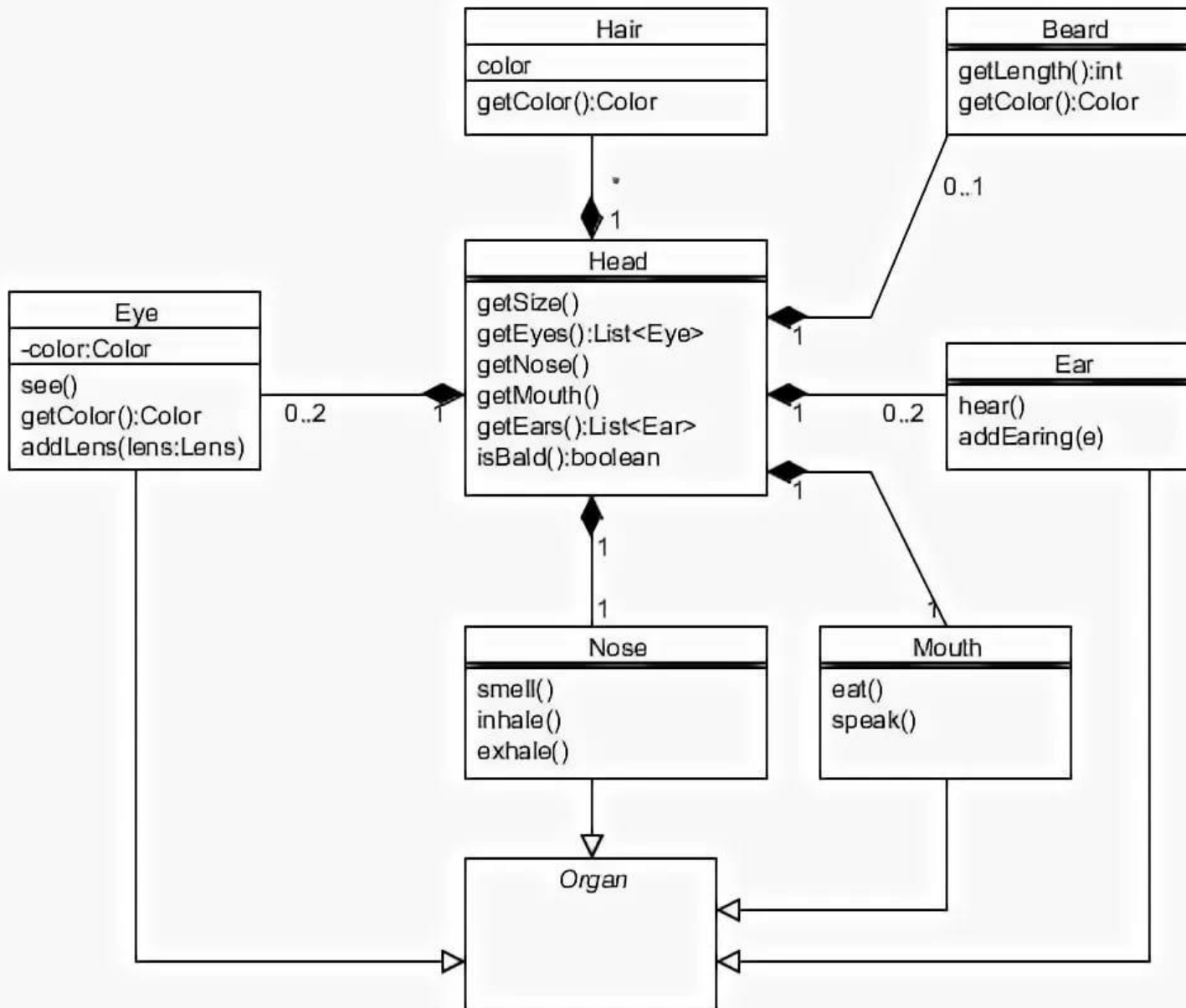
Отношение композиции

Диаграммы классов



**Отношение использования
(зависимости)**

Диаграммы классов



Диаграммы классов

Visual Paradigm Standard Edition (Group T-International Unive

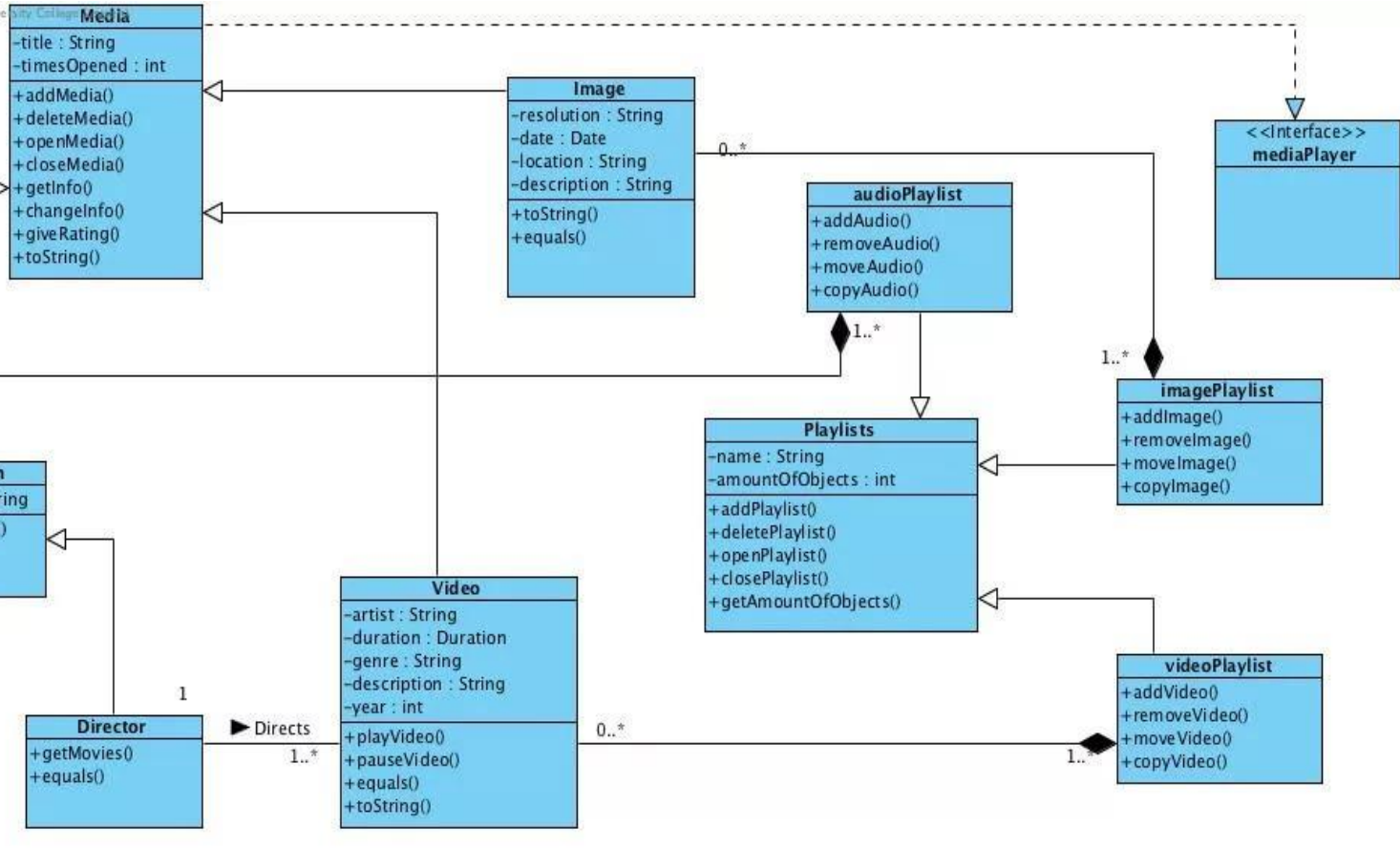


Диаграмма использования (use case diagram)

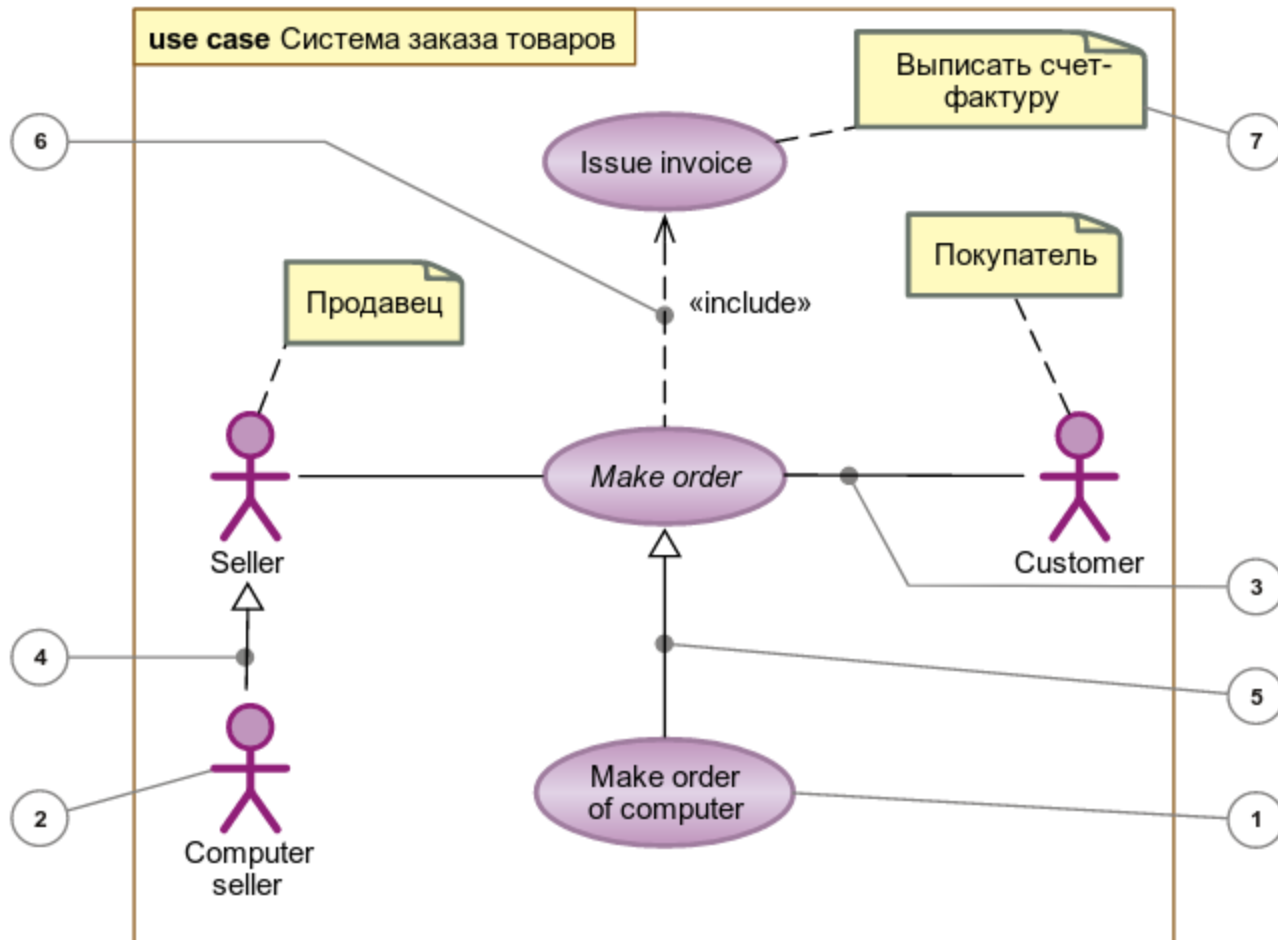
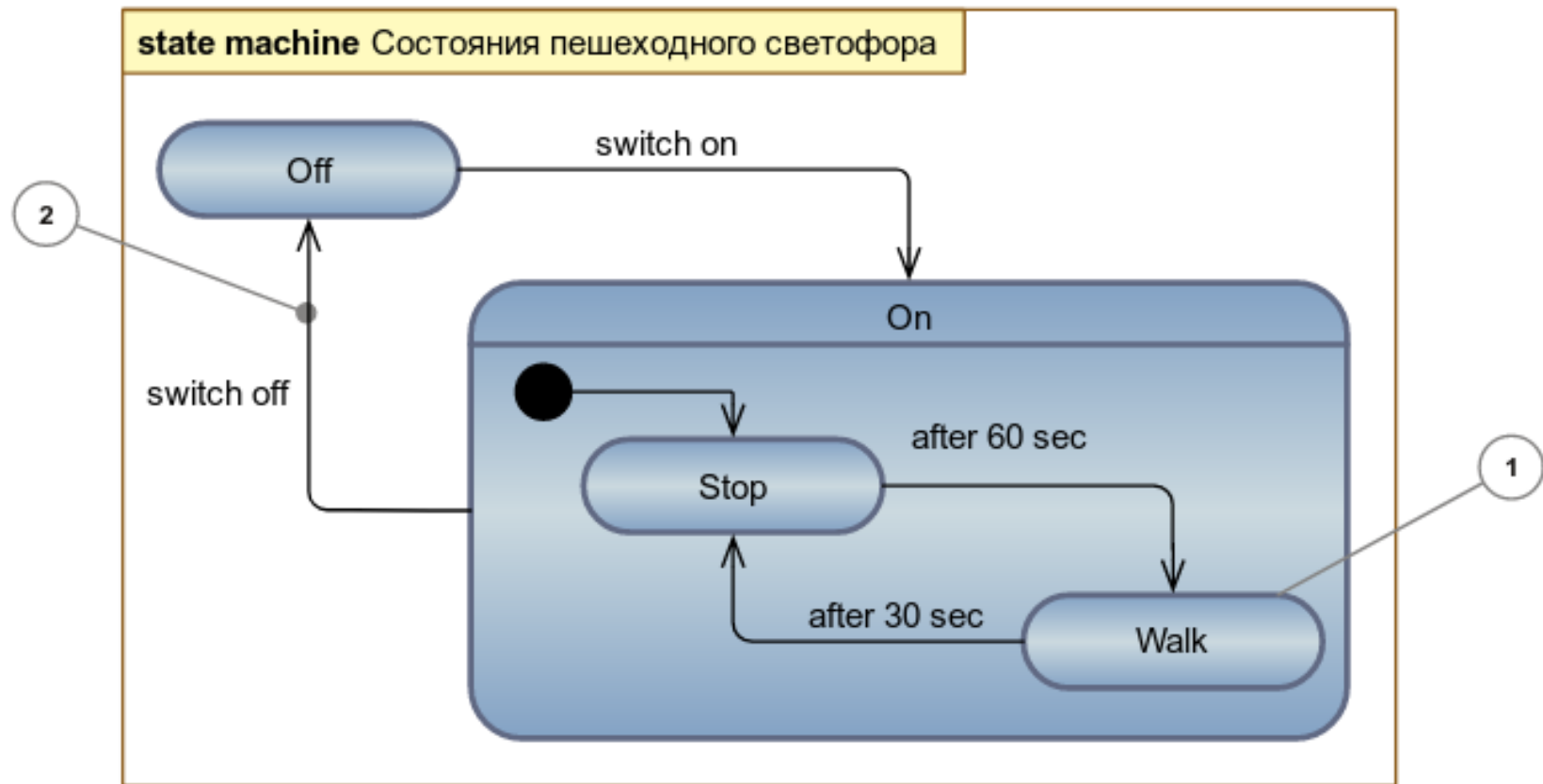
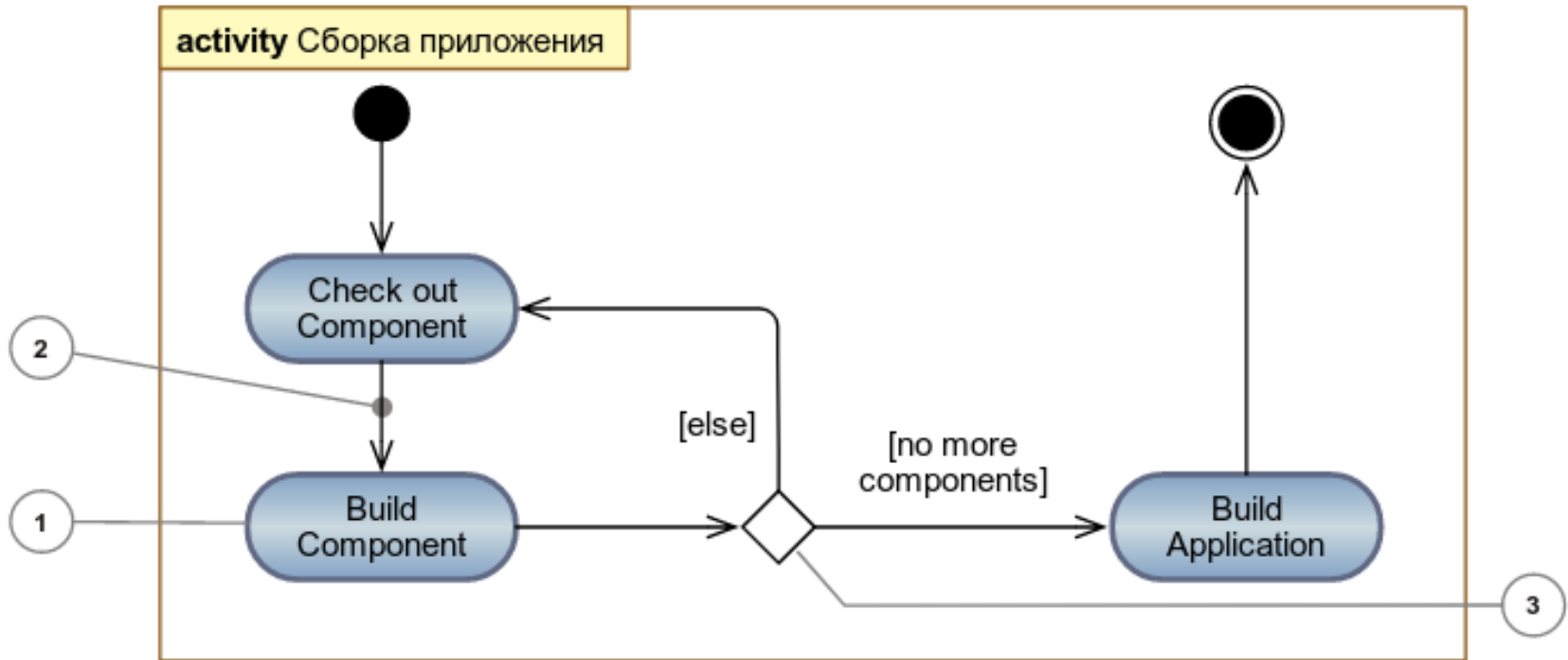


Диаграмма автомата (state machine diagram)



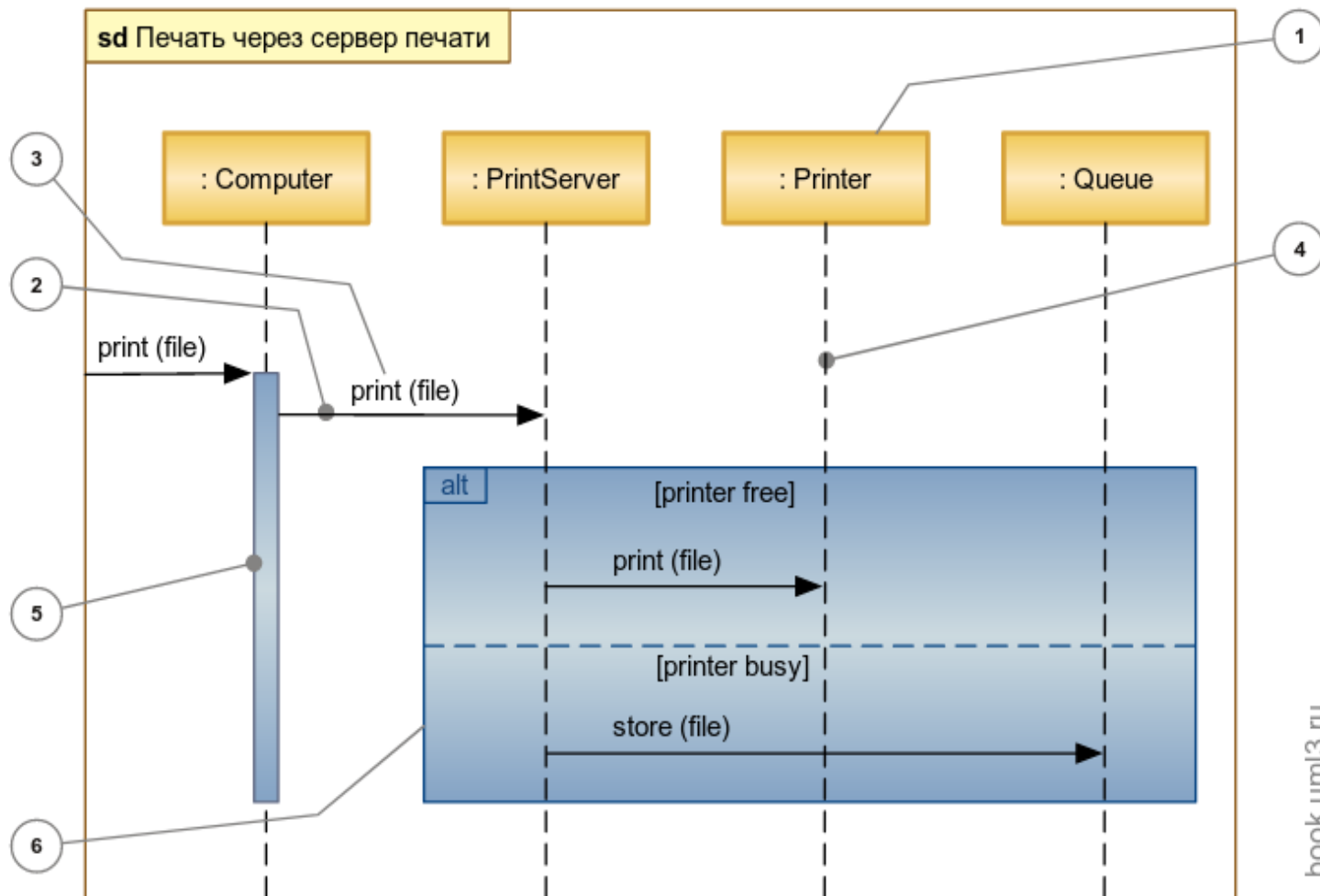
Описание поведения на основе явного выделения состояний и описания переходов между состояниями.

Диаграмма деятельности (activity diagram)



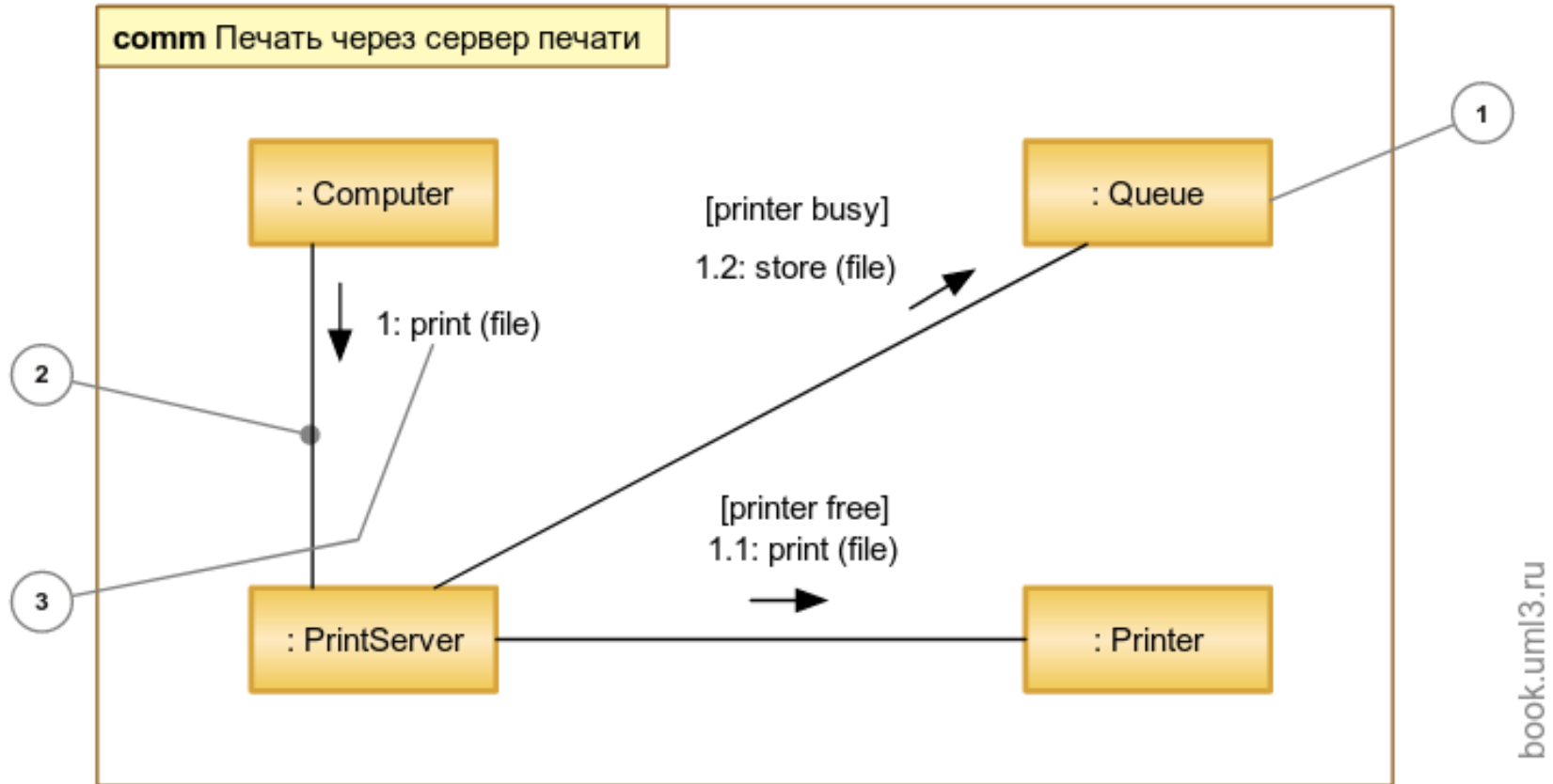
Описание поведения на основе указания потоков управления и потоков данных

Диаграмма последовательности



Описание поведения системы на основе указания последовательности передаваемых сообщений

Диаграмма коммуникации



Акцент на структуре связей между конкретными экземплярами