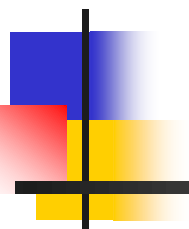


Объектно-ориентированный Анализ и Дизайн



Часть 1

Введение

Процесс разработки ПО

Анализ требований

ОО парадигма программирования

UML: Классы и пакеты



0. Введение

- Задачи курса
- Этапы семинарских занятий
- Критерии оценки
- Технологии и материалы
- С чего начать?



Цель и задачи курса

- Цель: ознакомление с современными методами объектно-ориентированной разработки программного обеспечения, позволяющими вести разработку программных систем средней и высокой сложности.
- Задачи:
 - Освоить основы языка UML
 - Изучить принципы ОО анализа и проектирования
 - Освоить основные архитектурные приемы
 - Выполнить небольшой проект в соответствии с процессом производства ПО, принятым в индустрии.



Этапы семинарских занятий

- **Этап 0:** разбиться на пары и придумать проект
- **Этап 1:** Провести анализ требований к проекту
- **Этап 2:** построить аналитическую UML модель
- **Этап 3:** Выработать архитектуру и дизайн системы
- **Этап 4:** реализовать проект на выбранной технологии
- **По всем этапам:** создать проектную документацию

Этап 0: 1 неделя.

Этапы 1-4: приблизительно по 1 месяцу



Критерии оценки

- **Отлично:** проект доведен до конца
- **Хорошо:** как минимум начат Этап 4
- **Удовлетворительно:** проект готов к старту Этапа 4
- **Неудовлетворительно:** проект НЕ готов к старту Этапа 4 = не закончен Этап 3
- **Однозначно неуд:** не закончен этап 2

При любом состоянии проекта незнание материала лекций -> снижение оценки



Технологии и материалы

- Материалы курса: <http://www.inteks.ru/OOAD/>
- UML редактор: <http://change-vision.com>
 - Astah community edition
- Текстовый процессор: MS Word | OpenOffice Writer
- Система контроля версий: SVN
 - <https://ccfit.nsu.ru/svn/ooad/2011/>
 - авторизация - доменная



С чего начать?

- Придумать проект и написать к нему Vision
- **Vision** – текст 1/2 страницы из 3 абзацев
 - Введение в предметную область
(простое описание, которое позволит непосвященному понять, о чем далее пойдет речь)
 - Известные проблемы предметной области
 - Предлагаемое решение
(какие именно проблемы из предыдущего абзаца и как именно решит ваш проект)



Разработка ПО: мифы

- Миф 1: разработка программ – это же так интересно!
- Миф 2: разработка программ – это искусство
- Миф 3: программирование – это вообще *творческая деятельность*, что-то вроде работы писателя или ученого
- Миф 4: хорошие программы пишутся только под UNIX/Windows на C++/Java/C#/Python/Ruby
(нужное подчеркнуть или вписать)
- Миф 5: eXtreme Programming/SCRUM лучше, чем RUP
- И т.д



Разработка ПО: реальность

- Реальность 1: разрабатывать программы – тяжелый и кропотливый труд. *К счастью, сравнительно хорошо оплачиваемый.*
- Реальность 2: разработка программ – это профессия, которой нужно учиться. Никаким «даром свыше» добиться в ней успехов нельзя.
- Реальность 3: программирование – не творчество, а налаженная индустрия. Со своими стандартами, правилами и процессами. Гениям-одиночкам в ней давно места нет. Профессионалам, умеющим работать в команде – есть.



Разработка ПО: реальность

- Реальность 4: профессионал не вступает в религиозные войны a-la UNIX vs. Windows, он выбирает платформу и технологию исходя из специфики проекта.
- Реальность 5: Ни один процесс разработки сам по себе успеха проекта не гарантирует. Процесс выбирается исходя из целей проекта и имеющихся ограничений. «Серебряной пули» не существует, каждый из процессов имеет и преимущества, и недостатки. Задача профессионала – четко их понимать.



Чтобы не расстраивались ...

Место для творчества и искусства в разработке ПО все же есть.

Немного, процентов 10-20, и, за исключением небольшого числа задач (вроде разработки «хитрых» алгоритмов), все они приходится на

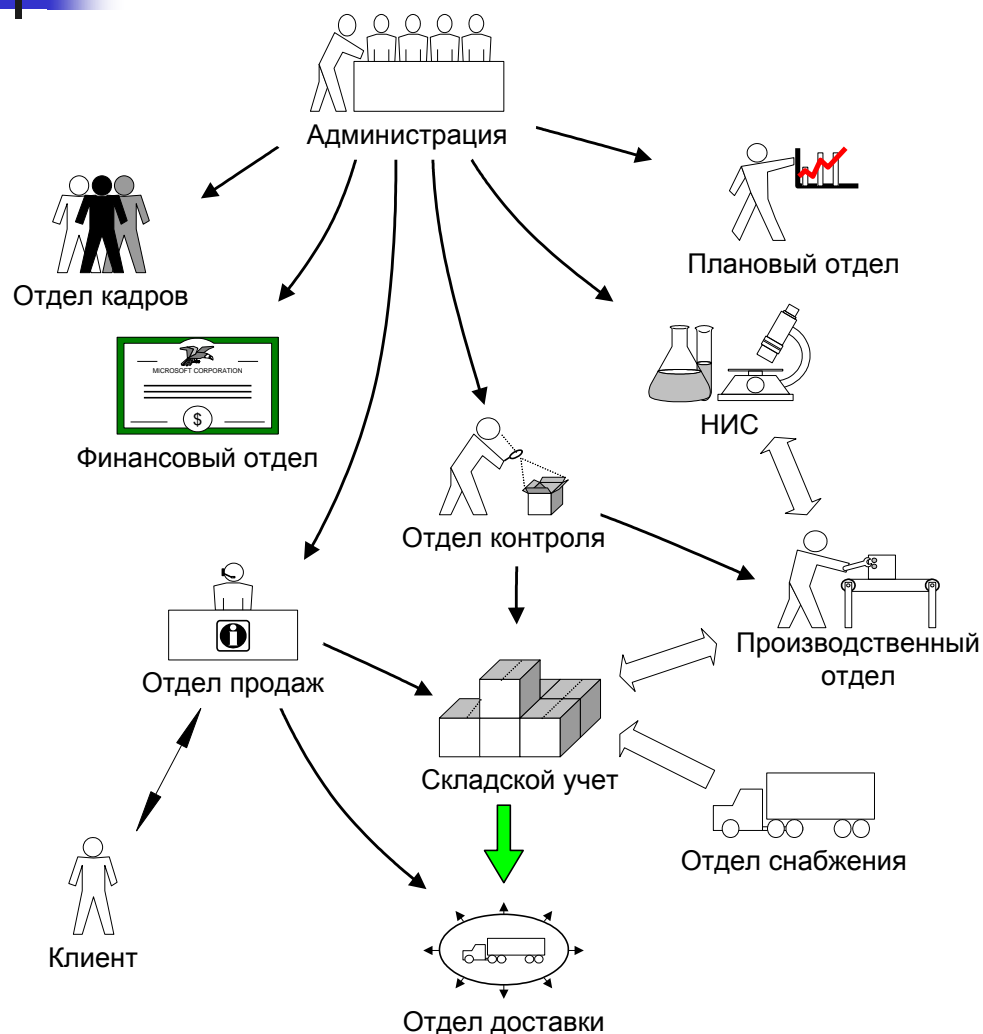
Анализ и Дизайн



1. Процесс разработки ПО

- Сложность, присущая программному обеспечению
- Процесс разработки ПО
- Роль Архитектора

Задача автоматизации



- Высокая композиционная сложность
- Наличие большого количества ролей и процессов
- Необходимость интеграции с существующими системами
- Постоянно изменяющиеся требования, связанные с развитием организации и оптимизацией процессов
- Интеграция решений задач
 - Финансовых
 - АСУ
 - Планирования
 - Управления персоналом
 - ...



О сложности

Современным информационным системам присущи как функциональная, так и композиционная сложность.

«Самолет представляет собой совокупность вещей, каждая из которых в отдельности стремится упасть на землю, но вместе, во взаимодействии, они преодолевают эту тенденцию»

G. Booch

Основным способом преодоления сложности является
декомпозиция.

1830: Ч.Бebbидж и А.Байрон – первая программа, отсутствие декомпозиции

1957: алгоритмическая декомпозиция (Фортран)

1958: функциональная декомпозиция (LISP)

1978: логическая декомпозиция и декларативные языки (Пролог)

1970: Структурное проектирование и алгоритмическая декомпозиция (Pascal, C, Algol, Cobol)

1980: Объектно-ориентированное проектирование и объектная декомпозиция (Simula(67г), Ada(80г), Smalltalk, C++(83г), Java, .NET)



ОО декомпозиция

- Программная система состоит из объектов, которые обмениваются сообщениями
- Каждый объект обладает:
 - Поведением (реакцией на сообщения)
 - Состоянием
 - Идентичностью (Индивидуальностью)
- Схожие объекты объединяются в классы



ОО Проектирование

- **OOD** – методология проектирования, соединяющая в себе процесс объектной декомпозиции и приемы представления логической, физической, а также статической и динамической моделей проектируемой системы

Г.Буч

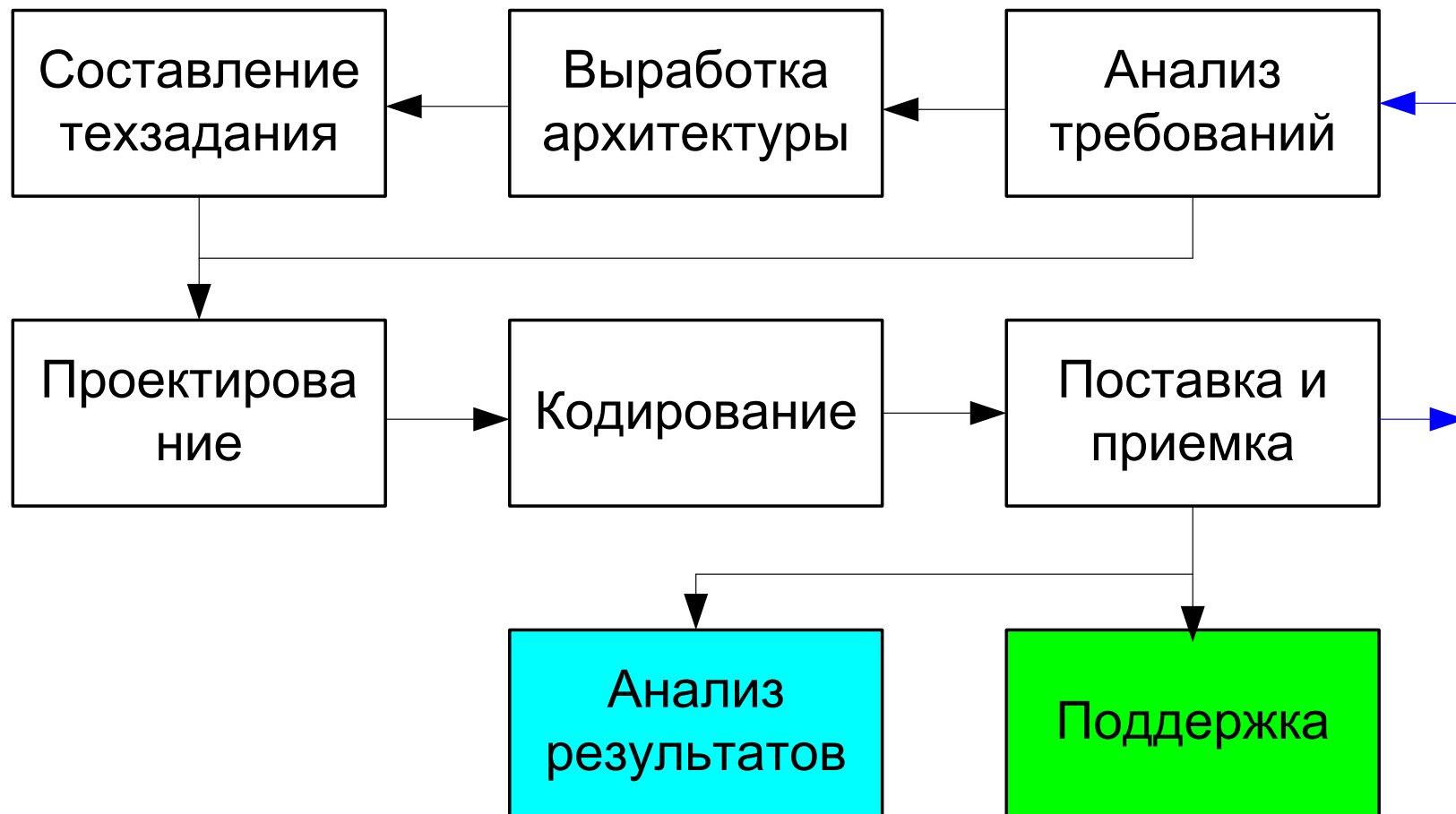


Процесс разработки ПО

*«Design and programming are human activities.
Forget it – and all is lost.»*

B. Stroustrup

Жизненный цикл IT проекта



Итеративно-инкрементная модель жизненного цикла IT-проекта



Модель жизненного цикла

- Наиболее распространенной и доказанно эффективной моделью является итеративно-инкрементная модель
- Может быть эффективна при наличии общих концепций и единой **методологии**, лежащей в основе всех составляющих процесса разработки



Методологии

- RUP-ообразные
 - RUP (Booch, Rumbaugh, Jacobson)
 - MSF (Microsoft)
- Agile (легковесные):
 - eXtreme Programming (Beck, Cunningham, Martin, Fowler, Cockburn)
 - SCRUM
 -



Архитектор / System Architect

«Идеальный архитектор должен быть писателем, математиком, знать историю, быть знатоком философии, понимать музыку, обладать знаниями в области медицины, юриспруденции и астрономии»

Витрувий, 25 г до н.э.

«Работа архитектора - это серии суб-оптимальных решений, сделанных под давлением в обстановке неуверенности и нехватки информации»

Rational Unified Process



Задачи архитектора ПО

- Анализ требований и контроль их изменений
- Выработка архитектурного решения
- Выработка плана работ совместно с РМ
- Объектная декомпозиция системы
- Контроль за соблюдением архитектуры
- Контроль качества кода и соблюдения coding rules
- Участие в процессе QA
- Контроль архитектурных рисков

Обратите внимание на частоту слова «контроль». Архитектура – во многом *управленческая* деятельность.



Артефакты

Типичные артефакты работы архитектора по фазам проекта

Анализ требований	Описание требований, Use-case модель
Выработка архитектуры	Аналитическая модель системы, draft дизайн модели
Техзадание	Требования и архитектура в Техзадании
Проектирование	SRS, SAD, Дизайн-модель системы, Модель размещения
Кодирование	Отчеты по code-review, поддержка SRS, SAD и моделей в актуальном состоянии
Поставка и приемка	Deployment Guide
Сопровождение	Анализ процесса эксплуатации, предложения по изменению системы



2. Анализ требований

- Необходимость моделирования требований
- Понятие варианта использования и актера
- Диаграммы вариантов использования
- Отношения между вариантами использования
- Сценарии вариантов использования
- Диаграммы деятельностей и состояний



Анализ требований

Анализ требований - первая фаза итеративно-инкрементного процесса разработки ПО

В RUP-образных процессах выполняется Системным Аналитиком.

На практике чаще всего выполняется будущим Архитектором проекта



Типы требований

- **Функциональные**

- Набор функций, которые система должна предоставить пользователям

- **Нефункциональные**

- Требования к производительности (время отклика, число запросов в секунду и т.п)
- Требования к отказоустойчивости (время наработки на отказ, процент времени работы от общего времени и т.д)
- Требования к нагрузочной способности (число пользователей, количество одновременных сессий и т.п)
- И т.д.



ОО Анализ

ОО Анализ – это методология, в которой *требования* к системе воспринимаются с точки зрения классов и объектов, выявленных в предметной области.

Г.Буч



О важности анализа требований

- 31% проектов – остановлены до завершения
- 53% проектов стоили 189% первоначальной оценки
- В крупных компаниях 9% проектов выполнено в срок и без превышения бюджетов.
- В мелких компаниях 16% проектов выполнено в срок и без превышения бюджетов.

Отчет Standish Group, 1994, (проанализировано 8000 проектов)



И еще о важности требований

Источники проблем в проектах

- Недостаточное количество информации от пользователей - 12,8%
- Неполные спецификации - 12,3%
- Изменения требований - 11.8 %

Итого ~37% источников проблем связаны с требованиями:
их сбором, анализом и учетом

Из отчета Standish Group, 1994, (8000 проектов)



И еще о важности требований

Источники проблем при сдаче проекта

- **Требования** - **41%**
- Проектирование - 28%
- Данные - 6%
- Интерфейс - 6%
- Окружение - 5%
- Человеческий фактор - 5%
- Документация - 2%
- Остальные - 7%

Sheldon F., "Reliability Measurement from Theory to Practice", 1992



Стоимость ошибок в требованиях

Относительная стоимость исправления ошибок в требованиях (по фазам)

- Инициация проекта - 0,1 – 0,2
- Проектирование - 0,5
- Кодирование - 1
- Компонентное тестирование - 2
- Тестирование в момент приемки - 5
- Использование и поддержка - 20

Davis A., “Software Requirements – Objects, Functions and States”, 1993



Проблемы

- Основным средством документирования требований является текст на естественном языке
- Проблемы:
 - Неоднозначность интерпретации
 - «Башмак лежит под колесом» - это о чем?
 - Нечеткая структура связей
 - Какое именно требование из этого толстого документа реализует вот этот кусок кода?
 - И обратно: какой код придется менять, если изменится требование из пункта 4.4.7 ?



Причины

- Непрерывно возрастающая сложность ПО
- Хороший метод борьбы со сложностью исследуемого объекта - формальная модель
- Проблему сложности математических вычислений решил **математический формализм** (цифры->арифметика->алгебра, мат.анализ, функциональный анализ, мат.логика ...)
- Требования к ПО, как и программный код, нуждаются в формальной модели

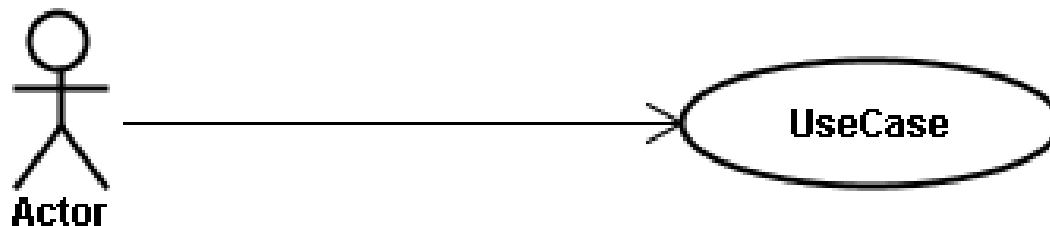


Unified Modeling Language

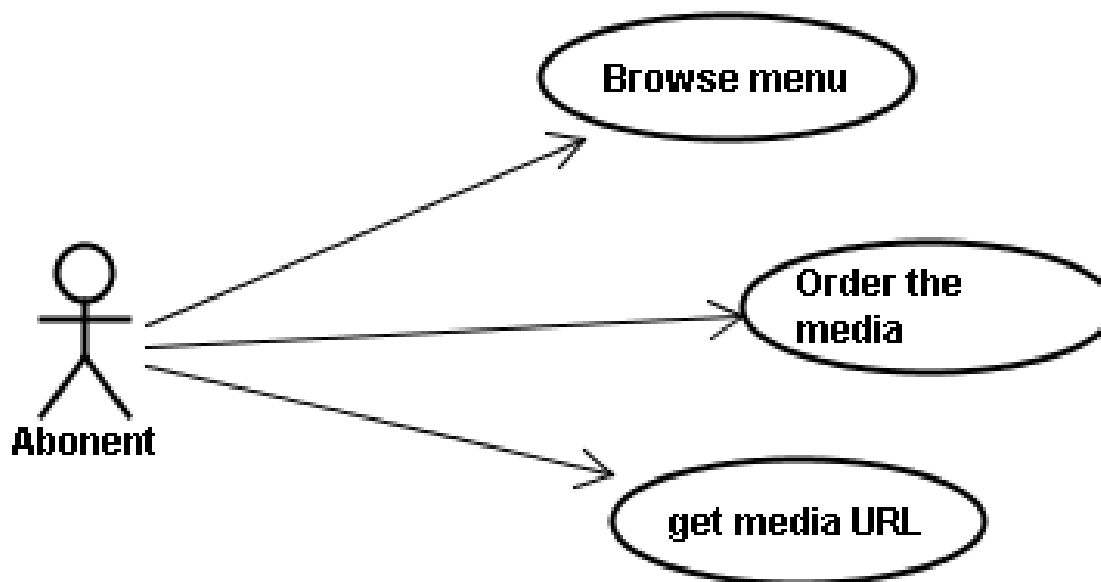
- Язык моделирования программных систем (и не только)
- Не является языком программирования
- Определяет нотацию и ее семантику
- Имеет UML-метамодель, описывающую семантику UML на языке UML
- Предоставляет возможности для расширения стандартной семантики
- OMG Unified Modeling Language Specification v 2.3
 - <http://www.omg.org/spec/UML/2.3/>

Варианты использования

- **Actor** – внешнее по отношению к системе действующее лицо (некто или нечто), взаимодействующее с системой.
- **Use case** – описание поведения системы в ответ на запрос извне (запрос Actor-а). Use-case описывает, что делает система с точки зрения Actor-а, но не как эти действия реализованы внутри.
- Use-case описывает **функциональные** требования
- Представление в UML:



Пример



Абонент сотовой сети – пользователь некоего сервиса, может выполнить 3 use-cases:

- просмотреть USSD menu,
- заказать контент (мелодию или картинку),
- получить заказ в виде URL для скачивания через WAP



Вариант использования (use case)

- Имеет **название**
- Определяет четкие **цели (value)**, которые достигаются актером в результате выполнения этого варианта использования
- Определяет как минимум один **сценарий** - последовательность событий и действий, необходимых для достижения данных целей

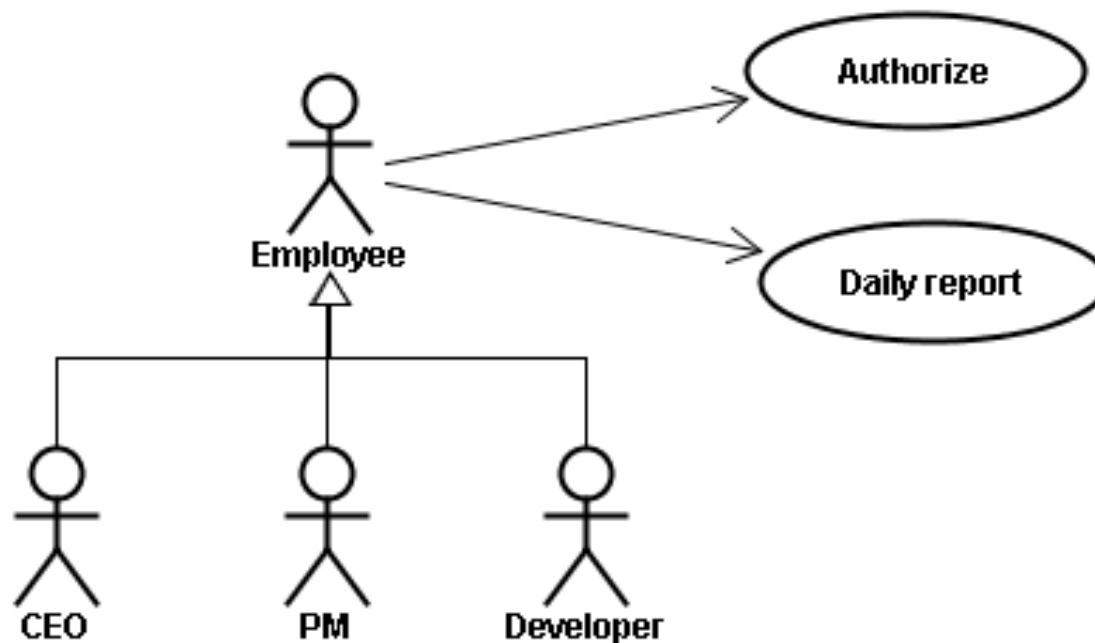


Use-case диаграммы

- Содержат:
 - Варианты использования со сценариями их выполнения
 - Актеров и их иерархию
 - Отношения между вариантами использования

Иерархия Актеров

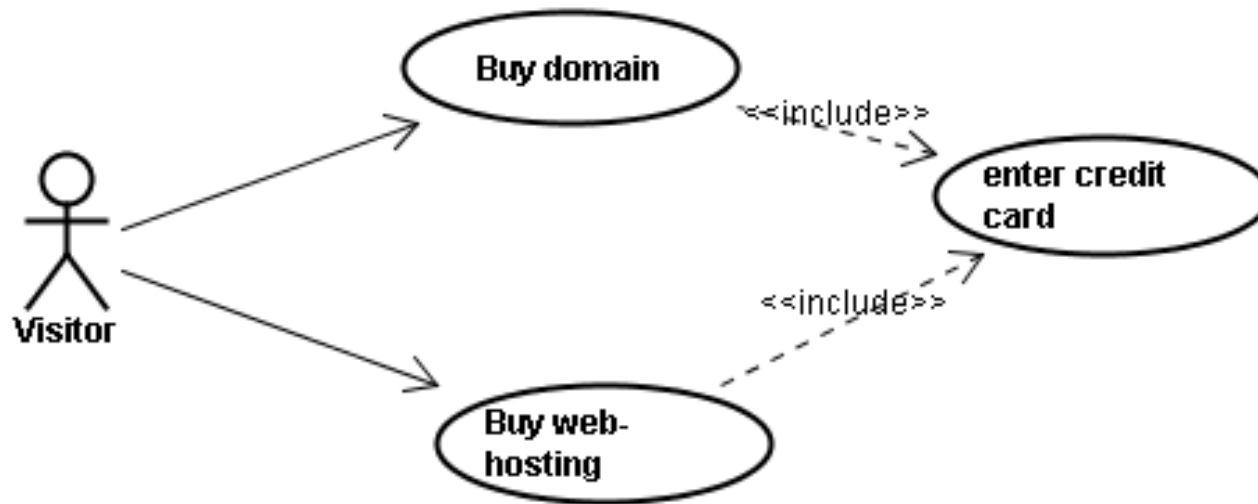
Различные Актеры могут иметь набор общих use-cases



Любой сотрудник (Employee), будь он CEO, PM или Developer, имеет возможность авторизоваться в системе и заполнить ежедневный отчет о проделанной работе.

Включаемые use-cases

- Stereotype: <<include>>
- Различные use-cases могут иметь общие части
- Абстрактный use-case не активируется actor-ами напрямую

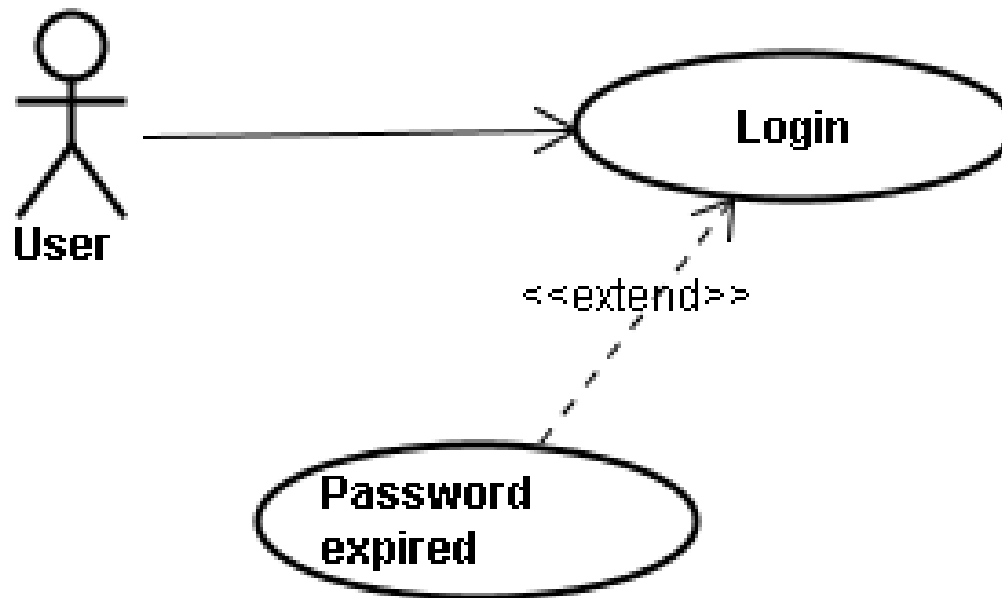


Посетитель сайта некоего регистратора доменов может попасть на страницу ввода информации о кредитной карте только попробовав что-то купить

Расширение use-cases

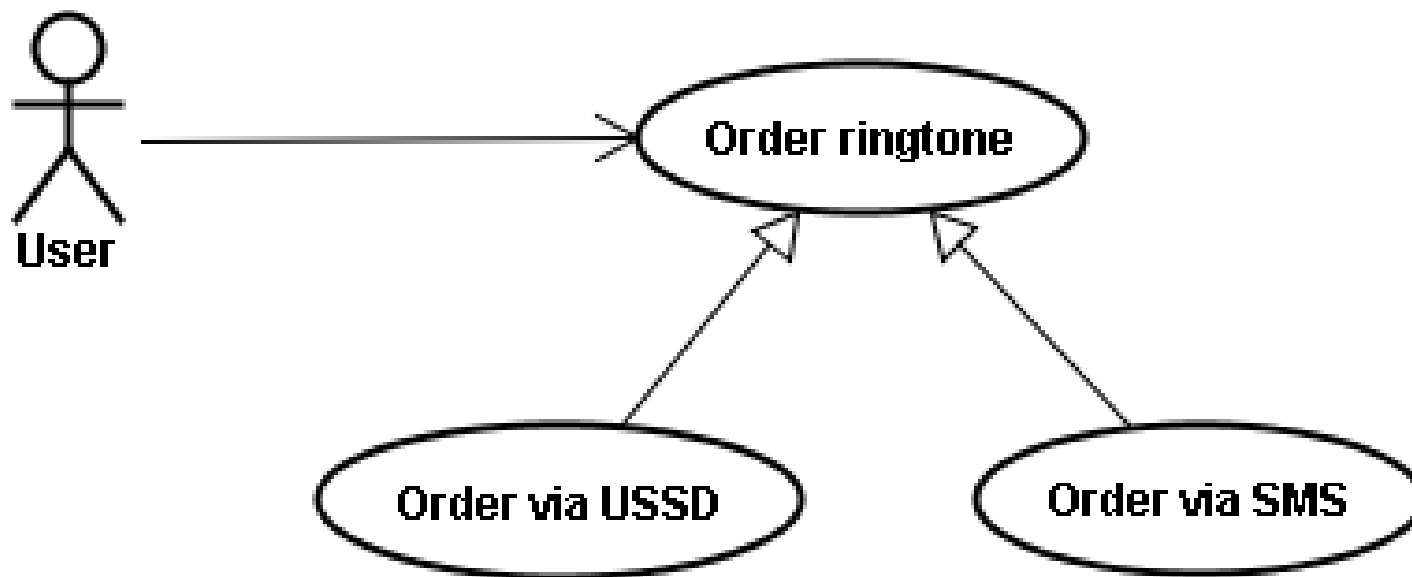
Stereotype: <<extend>>

Некоторые use-cases могут вызываться в контексте других только при некоторых условиях



Генерализация use-cases

Иногда два use-case-а могут иметь некоторую общность исполнения. Для того, чтобы показать эту общность, используется генерализация use-case-ов.





Документирование use-cases

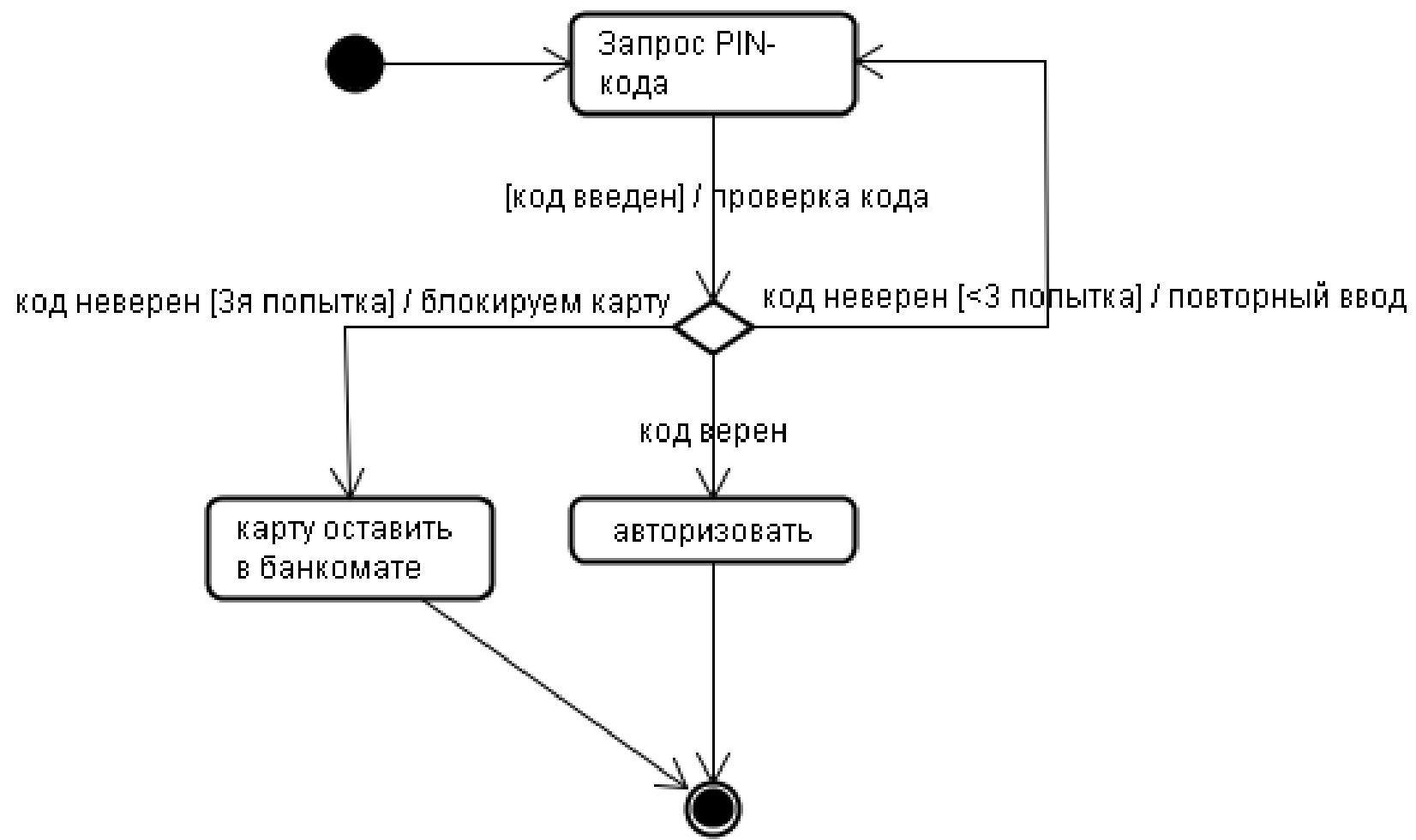
- **Имя**
- **Актер (актеры)** – действующие лица
- **Цель (value) актера** - текстовое описание
- **Предусловие** - условие старта, напр. файл должен быть открыт, прежде чем его можно будет сохранить)
- **Триггер** - событие, вызывающее начало use-case, напр. нажатие кнопки Save
- **Сценарии** - способы достижения цели
 - Основной сценарий (main success scenario, basic flow, happy path)
 - Альтернативный сценарий №1 (alternative scenario)
 - ...



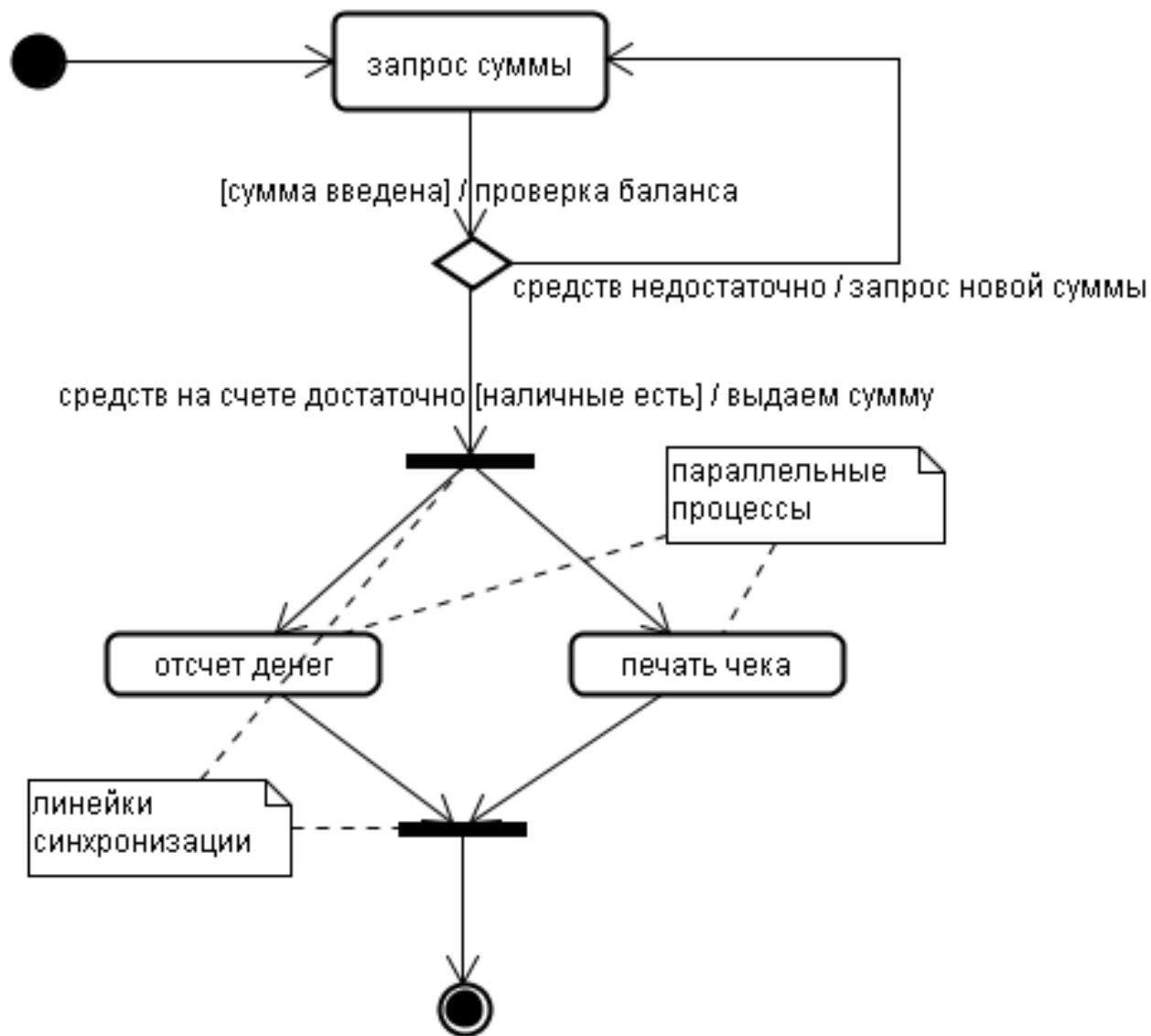
Диаграммы деятельности

- Используются для описания сценариев
- Описывают последовательности действий
- Activity - деятельность
- Transition – переходы между деятельностями
 - Event – событие вызывающее переход
 - Guard condition – условие перехода
 - Action – действие при переходе
- Decision – блок принятия решения
- Concurrent threads – параллельные деятельности
- Synchronization bar – линейка синхронизации параллельных деятельностей

Пример: банкомат. Ввод PIN кода



Пример. Банкомат - снятие денег





Типичные ошибки

- **1го рода:** сценарий принят за use-case
 - См. пример на следующем слайде
- **2го рода:** неоправданная генерализация
 - Есть актер, имеющий подклассы, но не имеющий ни одного use-case
 - Есть use-case, имеющий частные use-case, но не исполняемый ни одним актером
- **3го рода:** избыточная декомпозиция
 - Включаемый use-case имеет только один включающий
- **4го рода:** супер-абстракция (игра воображения)
 - Актеры, не имеющие собственных use-case
 - Use-case, не имеющие ни актера, ни базового или включающего use-case



use case и сценарий – не надо путать!

- Самая распространенная ошибка: путать сценарий и use-case
- Use-case **login** ОС Windows имеет 3 сценария:
 - Основной: вводим пользователя и пароль и входим в систему
 - Альтернативный 1: введенный пользователь или пароль неверен, возврат к форме ввода пользователя и пароля
 - Альтернативный 2: пароль верен, но срок его действия закончился, система выдает приглашение ввести:
 - Старый пароль
 - Новый пароль
 - Повторно новый пароль
- Важно понимать: несмотря на различия в формах ввода данных и действиях пользователя и системы - это **один** use-case, т.к во всех трех случаях пользователь достигает только одной **цели** – авторизация в системе.



Контрольные вопросы

- В чем отличие use-case от сценария?
- Приведите примеры use-case и scenario на примере системы mail.ru, MS Word, любой другой
- Mail.ru: Каким отношением могут быть связаны use-cases «Compose mail» и «Reply»?



3. ОО Программирование

- ОО парадигма программирования
- Основные понятия ООП: Классы, объекты, экземпляры
- Принципы ООП



ОО программирование

- Реши, какие требуются классы
- Обеспечь полный набор операций для каждого класса
- Явно вырази общность через наследование

Б.Страуструп

Квинтэссенция ОО- парадигмы программирования



ОО декомпозиция

- Программная система состоит из объектов, которые обмениваются сообщениями
- Каждый объект обладает:
 - Поведением
 - Состоянием
 - Идентичностью (Индивидуальностью)
- Схожие объекты объединяются в классы



Основные понятия ООП

- Класс – абстракция данных и поведения некоторого «вида» объектов
- Объект – экземпляр класса
- Instance (экземпляр) – объект, созданный во время исполнения программы
- `Cat cat;` // *объект cat – экземпляр класса Cat*



Основные принципы ООП

■ Абстракция

- Рассмотрение только существенных для решаемой задачи характеристик объекта
- Граница между существенными и несущественными характеристиками объекта называется **барьером абстракции**

■ Инкапсуляция

- Соккрытие особенностей реализации, отделение контрактных обязательств абстракции от их реализации

■ Иерархия

- Упорядочение абстракций, расположение их по уровням

■ Модульность

- Разделение системы на внутренне связанные модули, которые слабо связаны между собой



Это и есть ОО метод

- **Абстракция** - оставляет нам только существенные детали
- **Инкапсуляция** - убирает из поля зрения реализацию, оставляя для рассмотрения только поведение объектов
- **Модульность** - объединяет абстракции в группы
- **Иерархия** – распределяет абстракции по уровням, позволяя вести рассуждения на абстрактном уровне и применять результаты к частным случаям

Это и есть ОО-метод декомпозиции программной системы, ОО-способ ведения рассуждений, ОО-средство борьбы со сложностью



Принципы ООП

■ Типизация

- Способ защититься от использования объектов одного типа вместо другого

■ Полиморфизм

- способ поставить в соответствие некой грамматической конструкции контекстно-зависимую семантику

■ Параллелизм

- способность объекта обрабатывать несколько сообщений одновременно

■ Сохраняемость

- способность объекта сохранять состояние между сеансами работы приложения



4. Классы и пакеты

- Представление классов в UML
- Типы отношений между классами
- Пакеты, зависимости пакетов



Классы

- Class
 - набор объектов с общей структурой и поведением
- Interface
 - базовый класс, задающий только поведение, в UML имеет стереотип <<interface>>
- Abstract class
 - базовый класс, не имеющий экземпляров
- Parameterized class
 - параметризованный класс, шаблон
- Instantiated class
 - де-параметризованный шаблон

Примеры классов




Interface

 operation()


<<Interface>>

Interface


 operation()


AbstractClass

 attribute


 operation()


Class

 attribute

 operation()

ParameterizedClass

 attribute

 operation()

argument

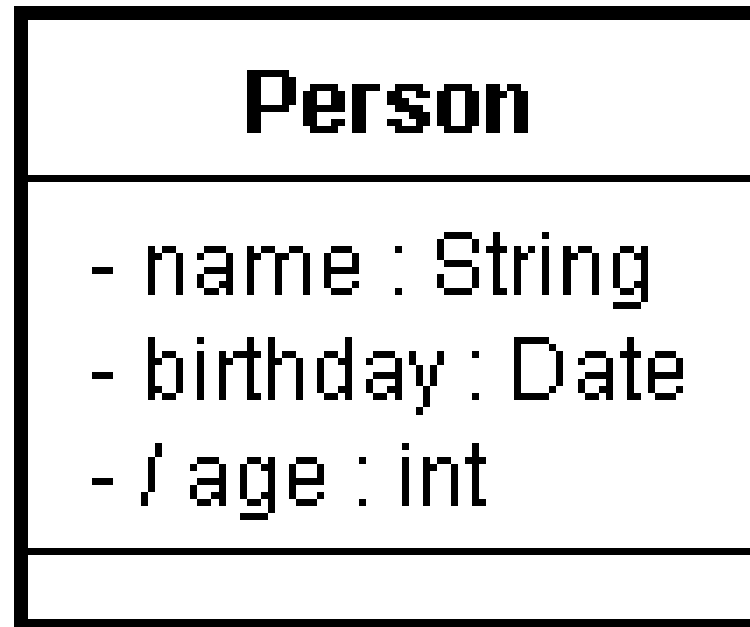


Атрибуты классов

- Attribute
 - атрибут (поле)
- Class attribute
 - атрибут класса (static)
- Derived attribute
 - производный (вычисляемый) атрибут
- Export control
 - доступ (public, protected, private)
- Containment
 - способ включения (composite, aggregate)
- Может иметь стереотип

Syntax: `<role_name>:<class_name><=default_value>`

Атрибуты классов



name, birthday – атрибуты

age – производный атрибут (вычисляется через birthday)



Атрибуты классов

DataLabels

+ NAME LABEL : String = "Your Name"

+ BIRTHDAY LABEL : String = "Birthday"

+ AGE LABEL : String = "Age"

name, birth_date и age - атрибуты класса (static)



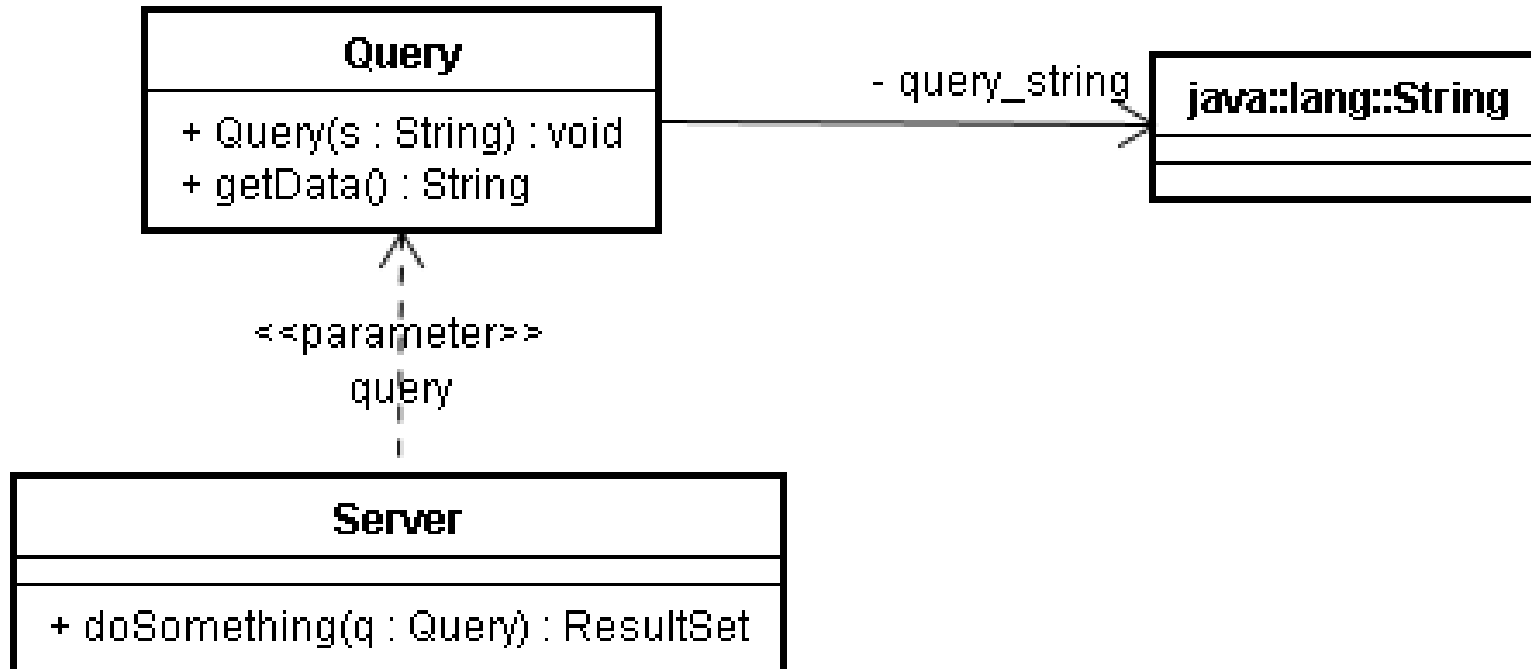
Методы(операции)

- Method (operation) – метод
- Class method – метод класса (static, final, abstract)
- Export control – public, protected, private, package
- Syntax: <stereotype> name(<parameters>) : <return type>
- Parameter: parameter_name : type

Date
- date : long
+ Date(date : long) : void + setDate(date : long) : void + getDate() : long

Диаграмма классов

- определяет типы объектов системы и статические связи между ними



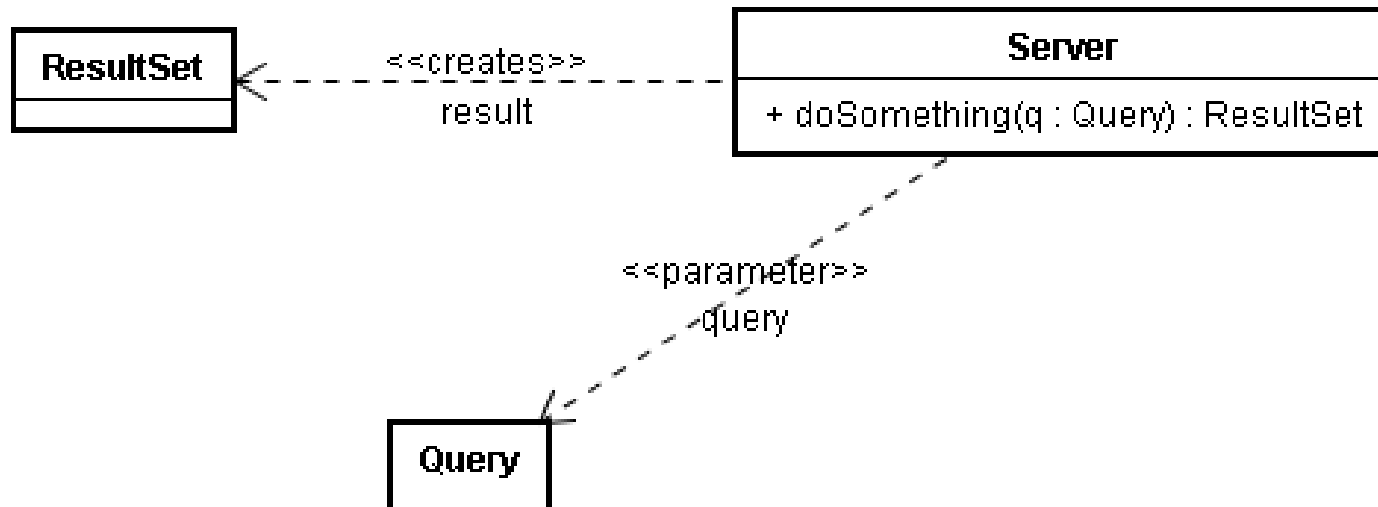


Связи между классами

- Зависимость - Dependency
- Ассоциация - Association
- Агрегация - Aggregation
- Композиция - Composition
- Генерализация - Generalization
- Реализация - Realization

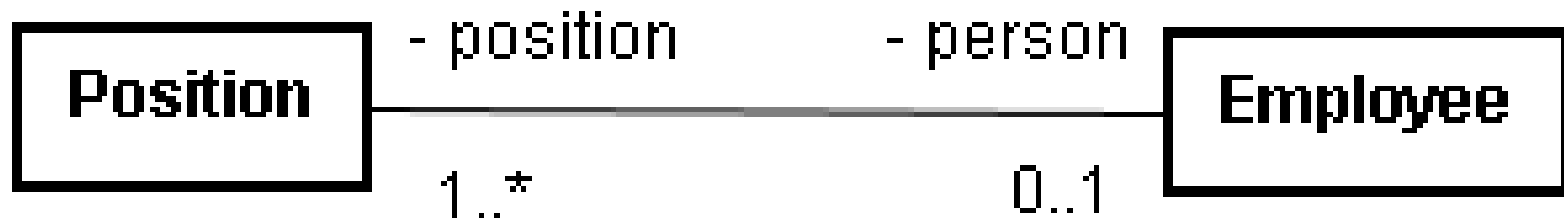
Dependency

- Определяет отношение зависимости (осведомленности)
- Имеет выделенное направление
- Может иметь стереотип
- Обладает ролью
 - Server зависит от Query, так как использует этот класс в качестве параметра метода
 - Server также зависит от ResultSet, поскольку возвращает значение этого типа



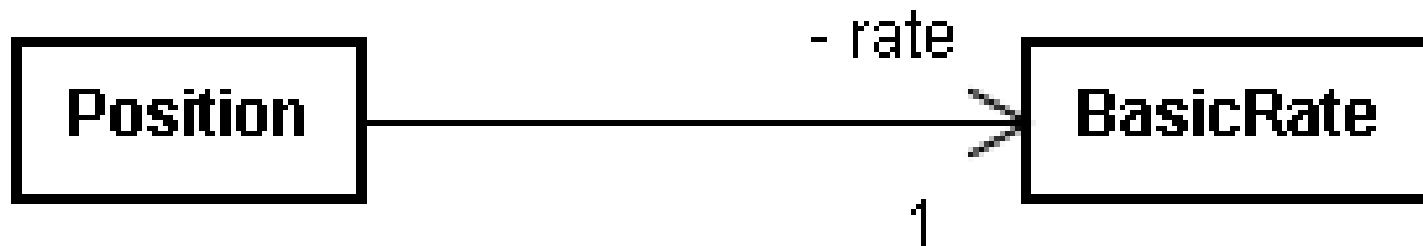
Association

- Ассоциация - отношение взаимодействия
- Подразумевает наличие зависимости
- Обладает 2-мя ролями
- Роль обладает множественностью (1, n, *, 0..n, 1..n, 1..*)
- Пример: сотрудник может занимать несколько должностей, но на одной должности находится не более одного сотрудника



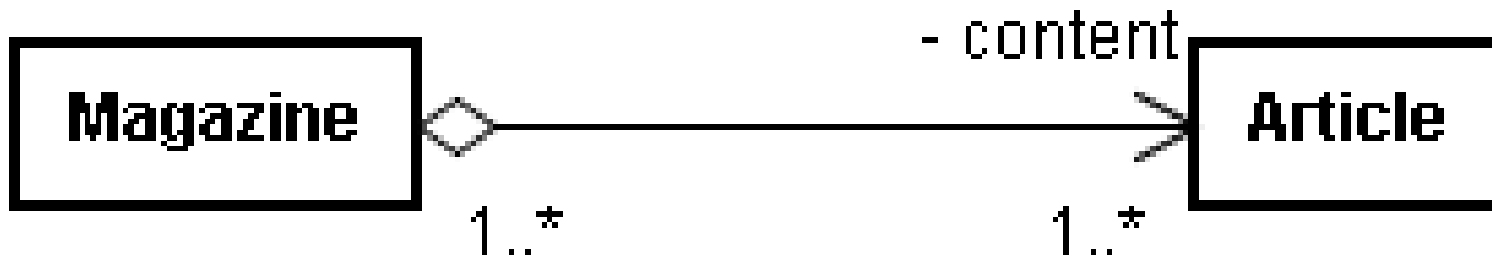
Association

- Ассоциация может иметь выделенное направление
 - Должность связана с базовым тарифом оплаты
 - Тариф оплаты никак не связан с конкретной должностью



Aggregation

- Агрегация – определяет отношение часть-целое
- Частный случай ассоциации
- Часть может принадлежать различным целым
 - Журнал состоит из одной и более статей; статья может быть опубликована нескольких журналах



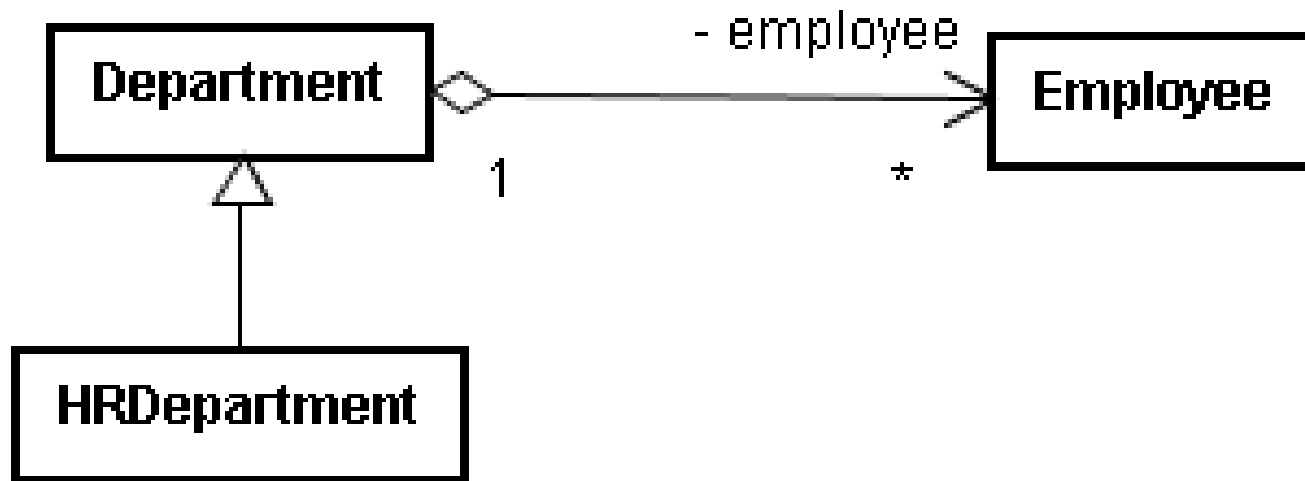
Composition

- Композиция – частный случай агрегации
- Отношение «часть - целое»
- Целое отвечает за жизненный цикл своих частей
 - Отделы не существуют без компании
- Часть принадлежит только одному целому



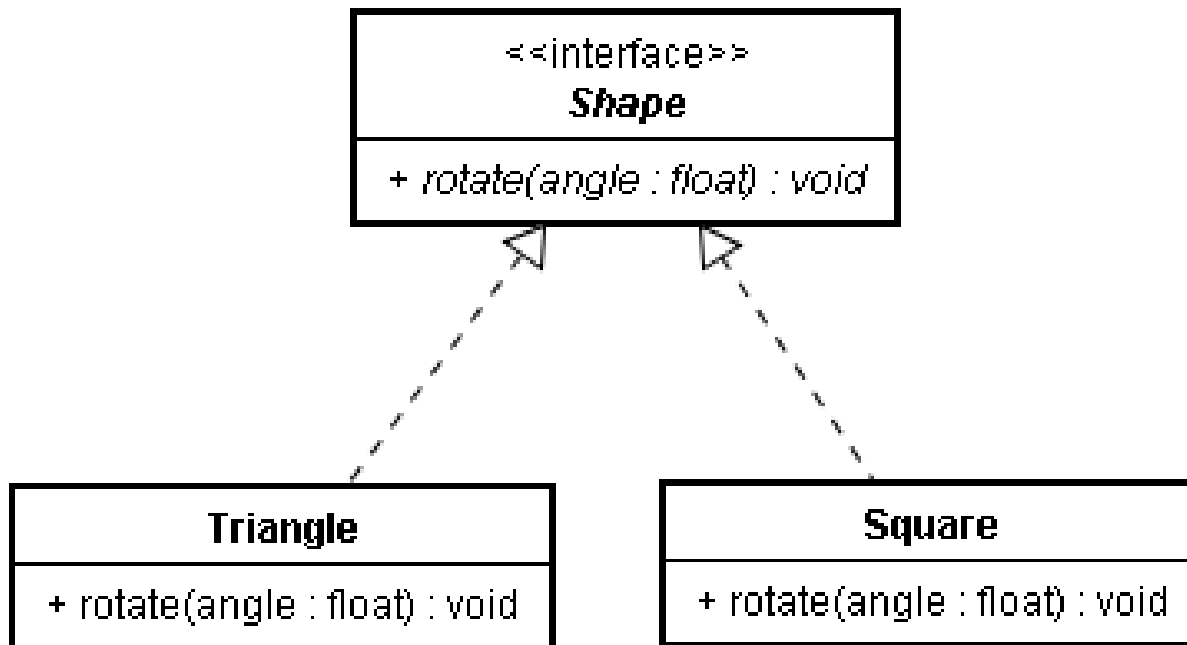
Generalization

- Генерализация - наследование, обобщение
- Отношение «частное-общее»
 - Отдел кадров – частный случай отдела



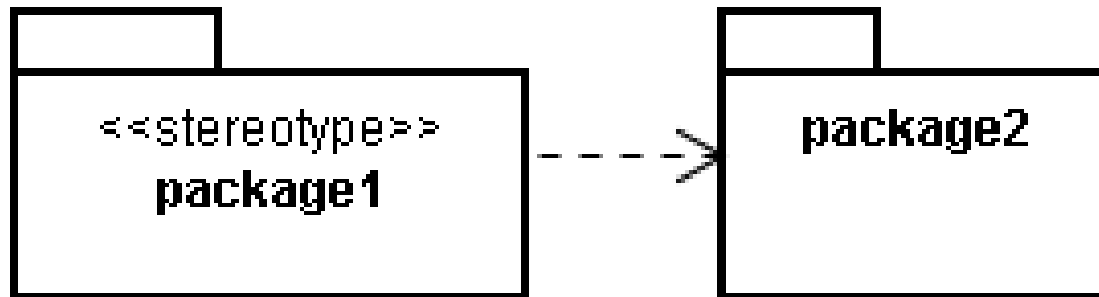
Realization

- Реализация – отношение выполнения соглашения (реализация интерфейса)
 - Треугольник и квадрат реализуют алгоритм вращения, специфицированный абстрактной сущностью «Фигура»



Пакеты в UML

- Package – пакет. Общий механизм организации элементов модели в группы
- Имеет имя
- Определяет пространство имен
- Может быть стереотипирован
- Может быть импортирован другим пакетом





Пакеты классов

- Пакет – группа тесно связанных классов.
- Пакеты должны создаваться таким образом, чтобы при использовании одного класса из пакета использовался весь пакет.

Диаграмма пакетов

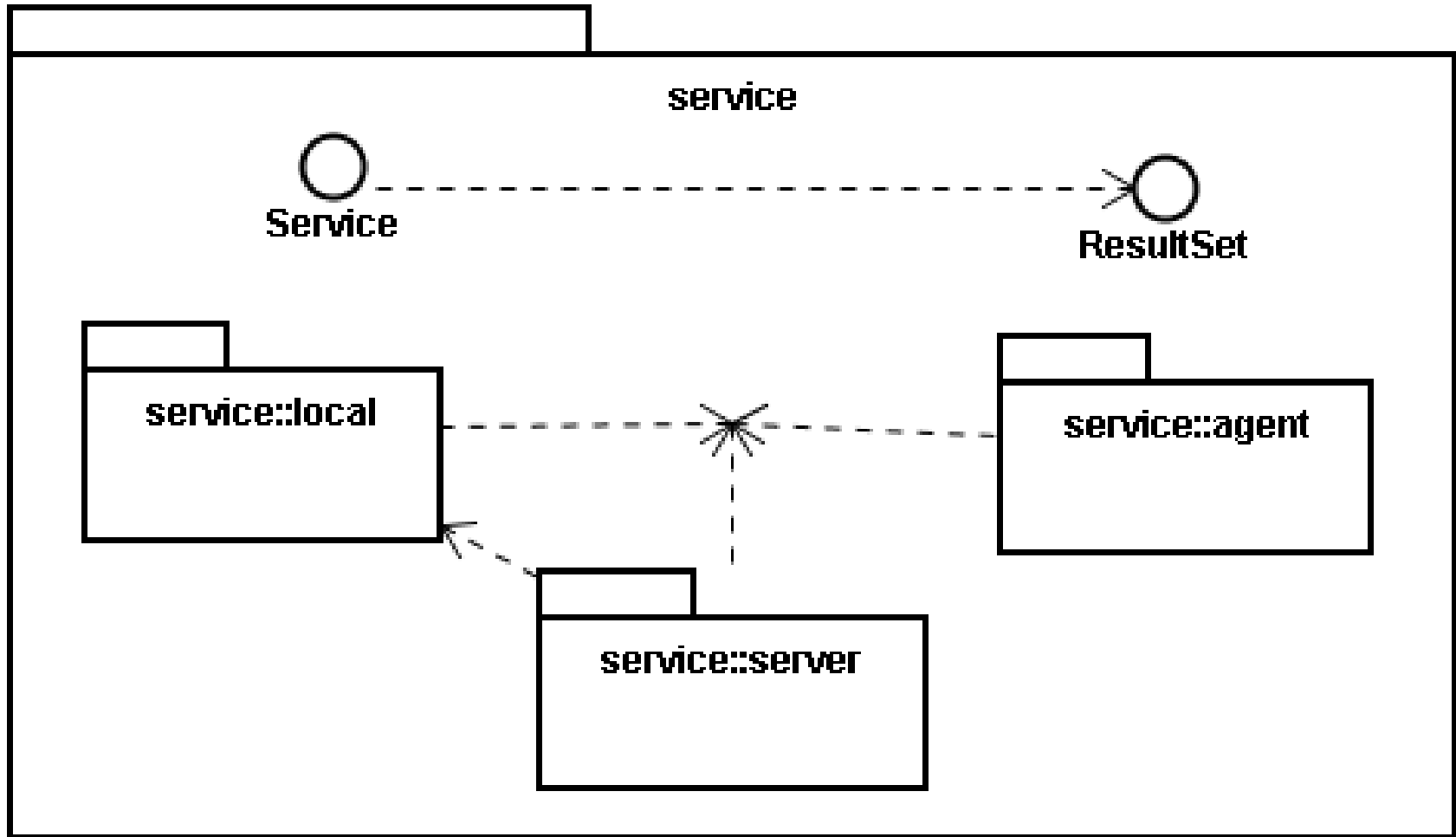
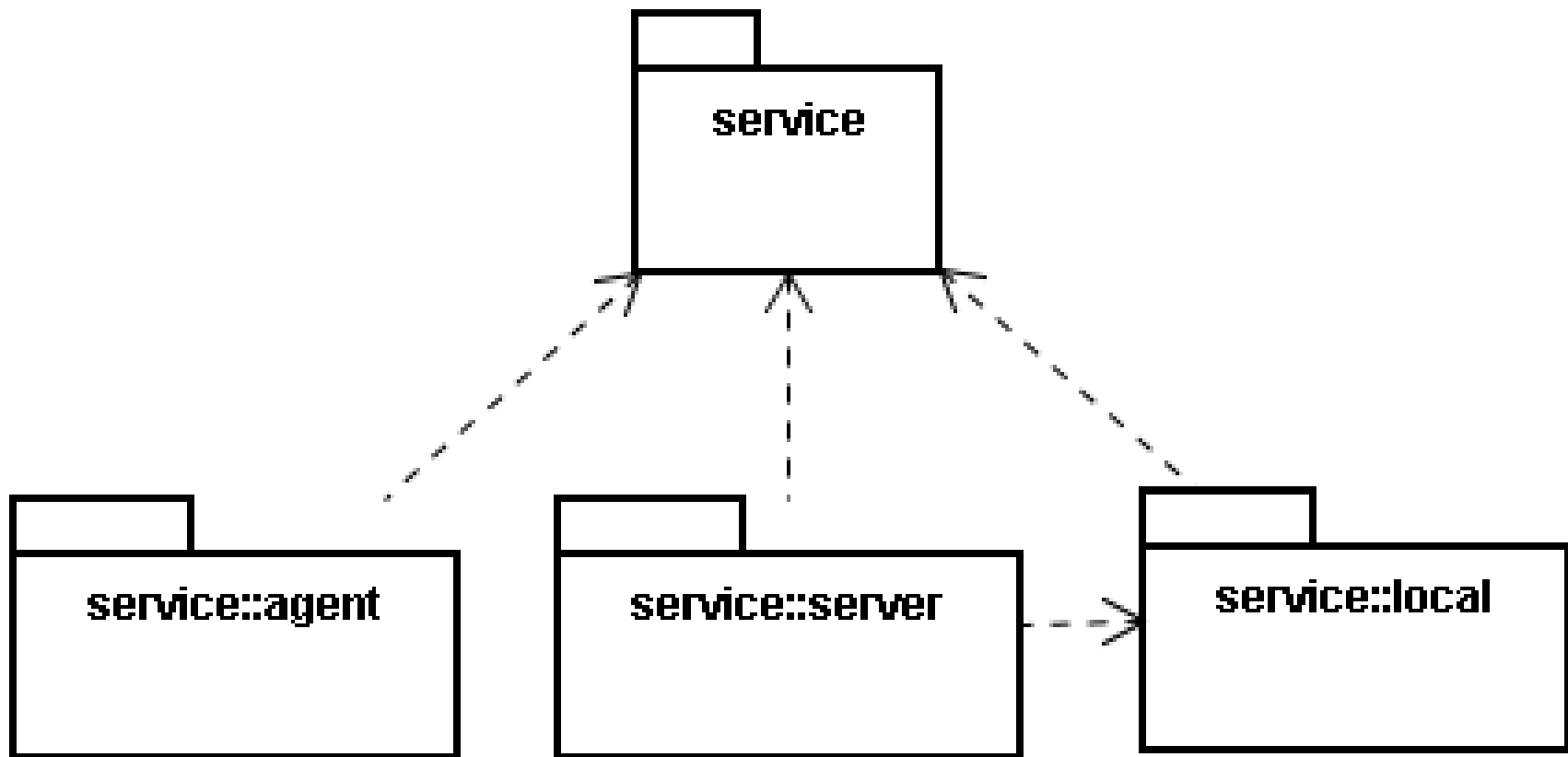
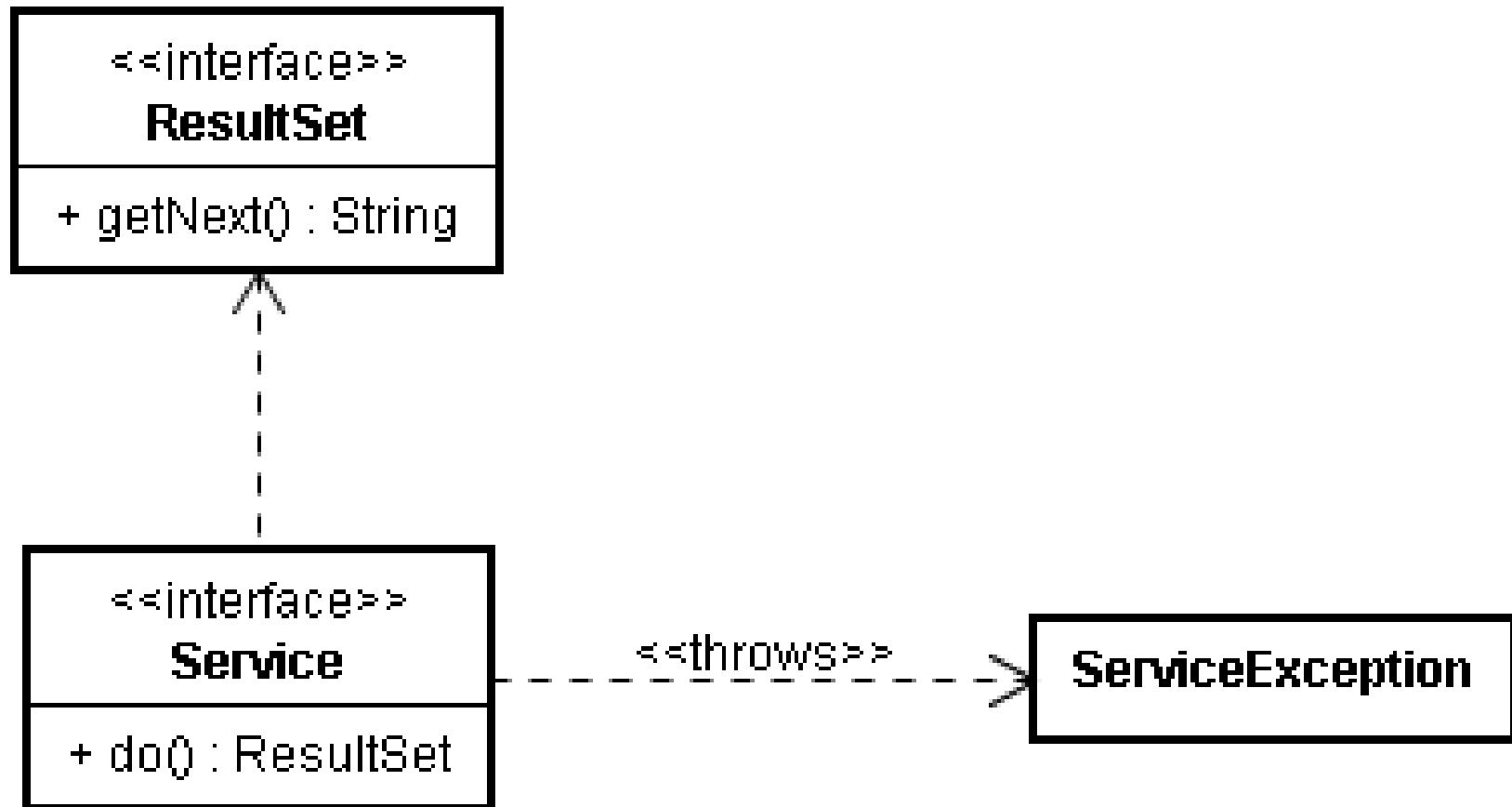


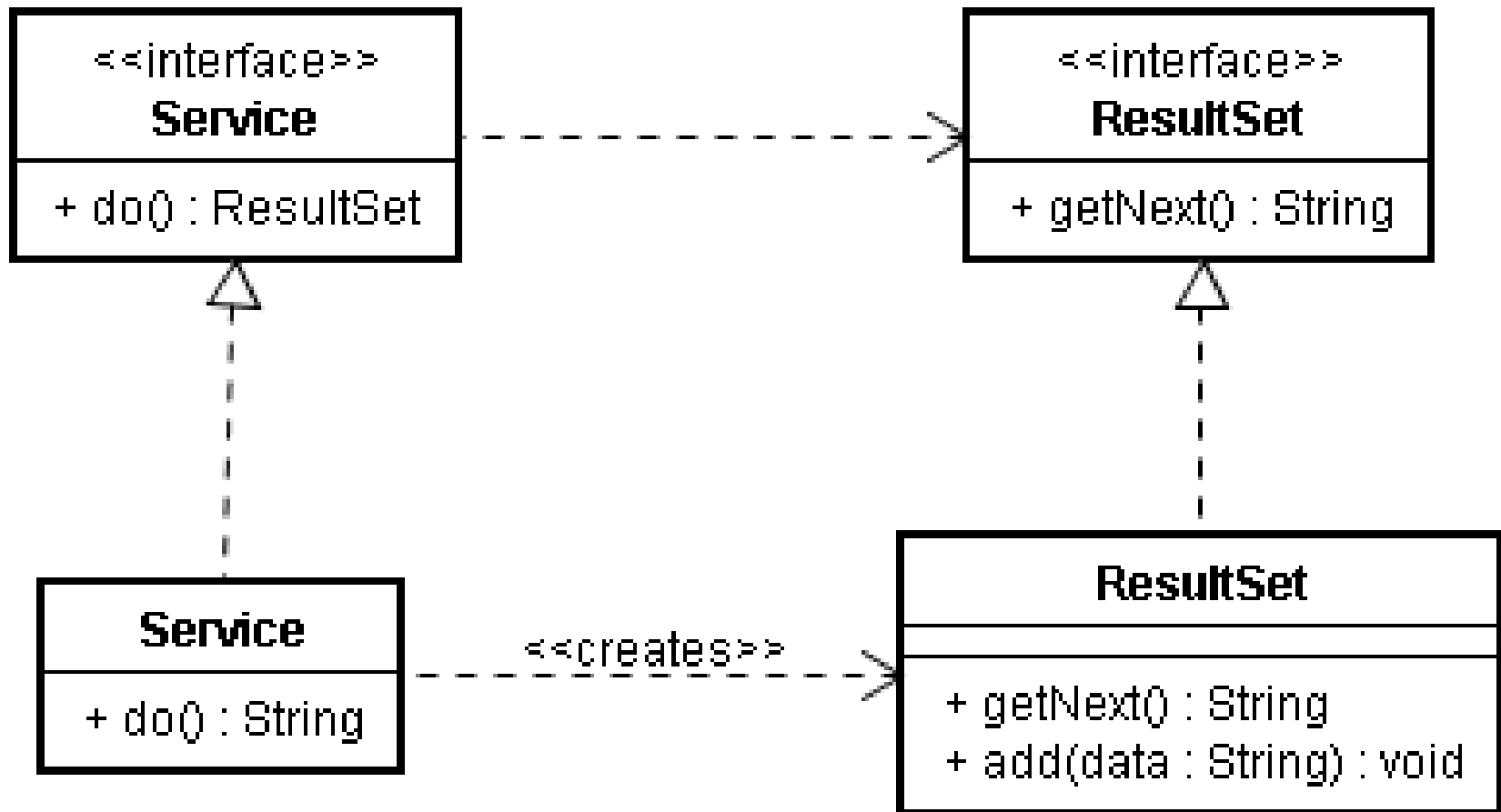
Диаграмма пакетов



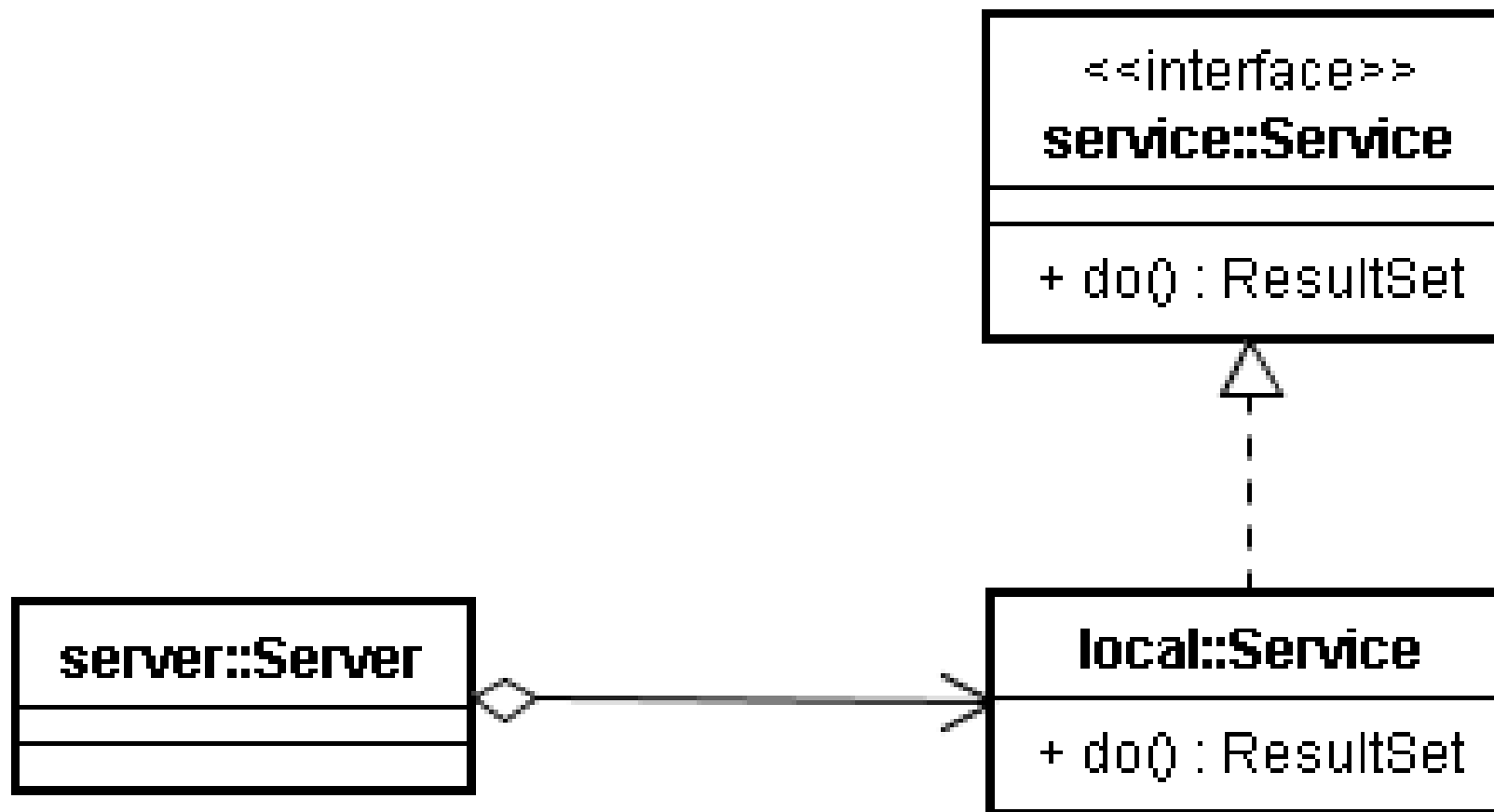
пакет service



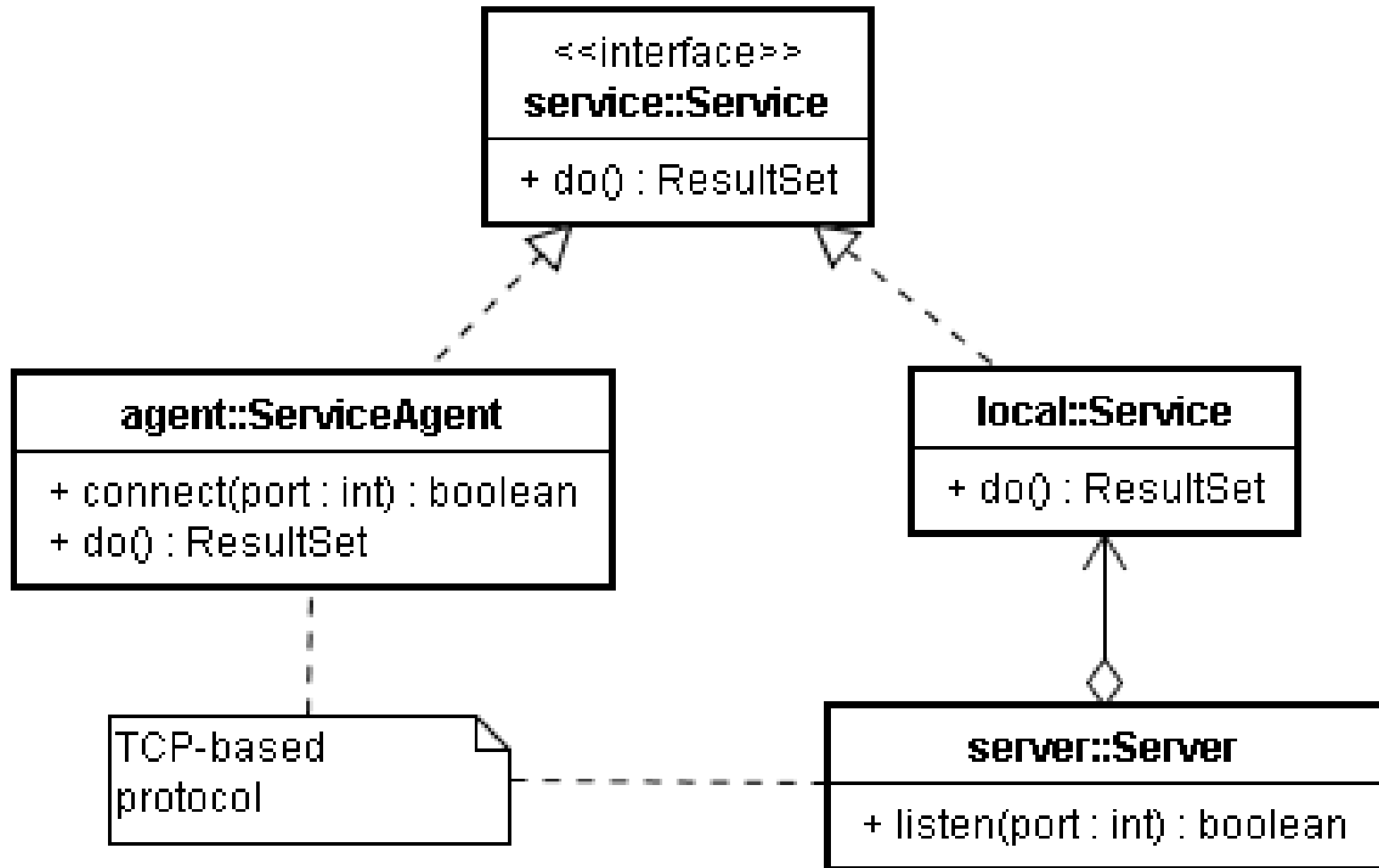
пакет service::local



пакет service::server



пакет service::agent





стереотипы пакетов

- `system` – система
- `subsystem` – подсистема
- `facade` – представление другого пакета
 - Например, пакет внешних интерфейсов подсистемы
- `framework` – набор шаблонов
- `stub` – заместитель другого пакета
 - Созданный, например, для тестирования



Контрольные вопросы

- Объясните разницу между агрегацией и композицией
- Приведите примеры генерализации абстракций
- Какой собственный стереотип пакета вы хотели бы использовать в своем проекте и для чего?