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: 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: https://ai.nsu.ru/projects/ooad/
- UML редактор: https://www.change-vision.com
  - Astah UML – free student license
- Текстовый процессор: MS Word | Google Docs
- Система контроля версий: GitLab
  - http://gitlab.ccfit.nsu.ru

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/SCRUM лучше, чем RUP/MSF/OpenUP
- и т.д

Разработка ПО: реальность

- Реальность 1: разрабатывать программы – тяжелый и кропотливый труд. *К счастью, сравнительно хорошо оплачиваемый.*
- Реальность 2: разработка программ – не искусство, а инженерия. 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 споры UNIX vs. Windows, C++ vs. Java, etc. Профессионал выбирает платформу и технологию исходя из требований к проекту.
- Реальность 5: Ни один процесс разработки сам по себе успеха проекта не гарантирует. Процесс выбирается и настраивается исходя из требований к проекту и имеющихся ограничений.

Но все не так плохо …

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

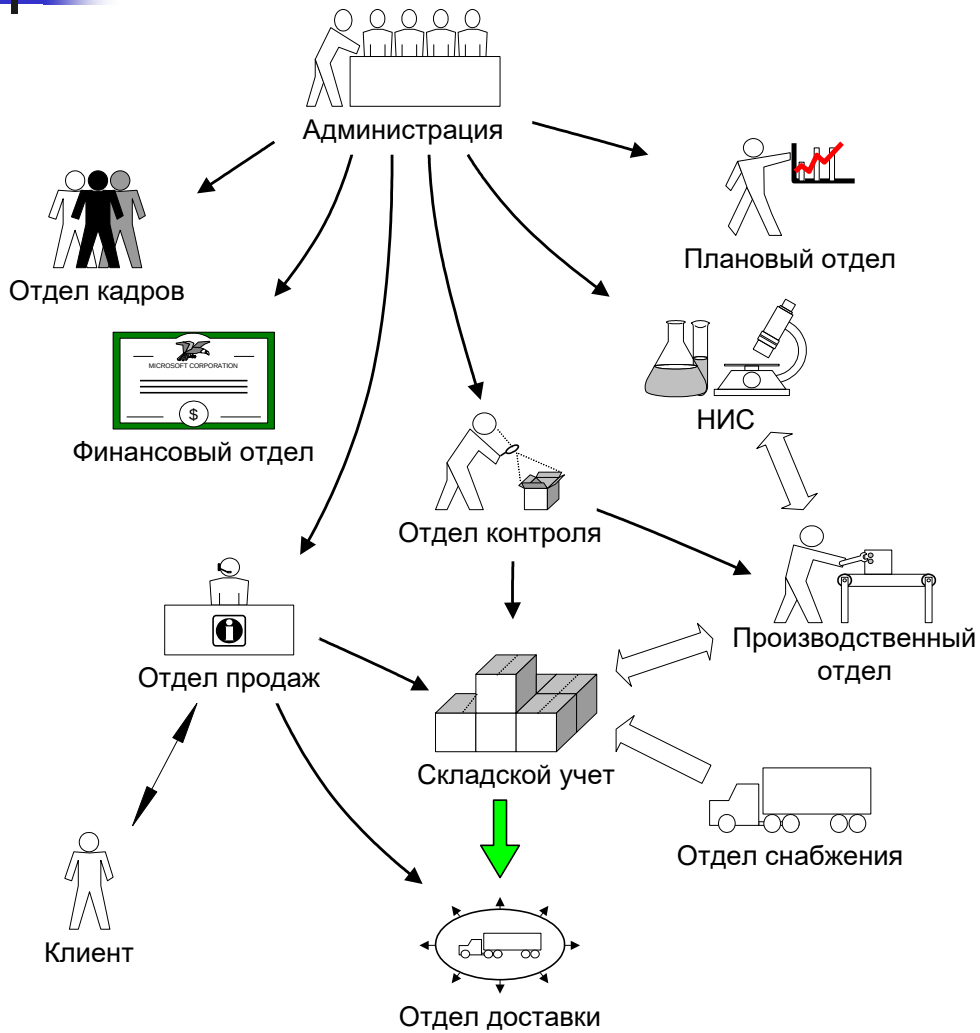
Если не учитывать довольно редкие задачи вроде разработки умных алгоритмов, это место

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
- Постоянные изменения требований, вызванные развитием организации и оптимизацией процессов
- Интеграция подсистем
  - Управления финансами
  - Планирования
  - Управления персоналом
  - Управления цепочками поставок
  - ...

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*.

О декомпозиции

1830: Ч.Беббидж 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(1967), Smalltalk(1980), Ada(1980), C++(1983), Java(1996), .NET(2000))

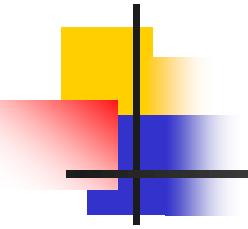
ОО декомпозиция

- Программная система состоит из объектов, которые обмениваются сообщениями
- Каждый объект обладает:
  - Поведением (реакцией на сообщения)
  - Состоянием
  - Идентичностью
- 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



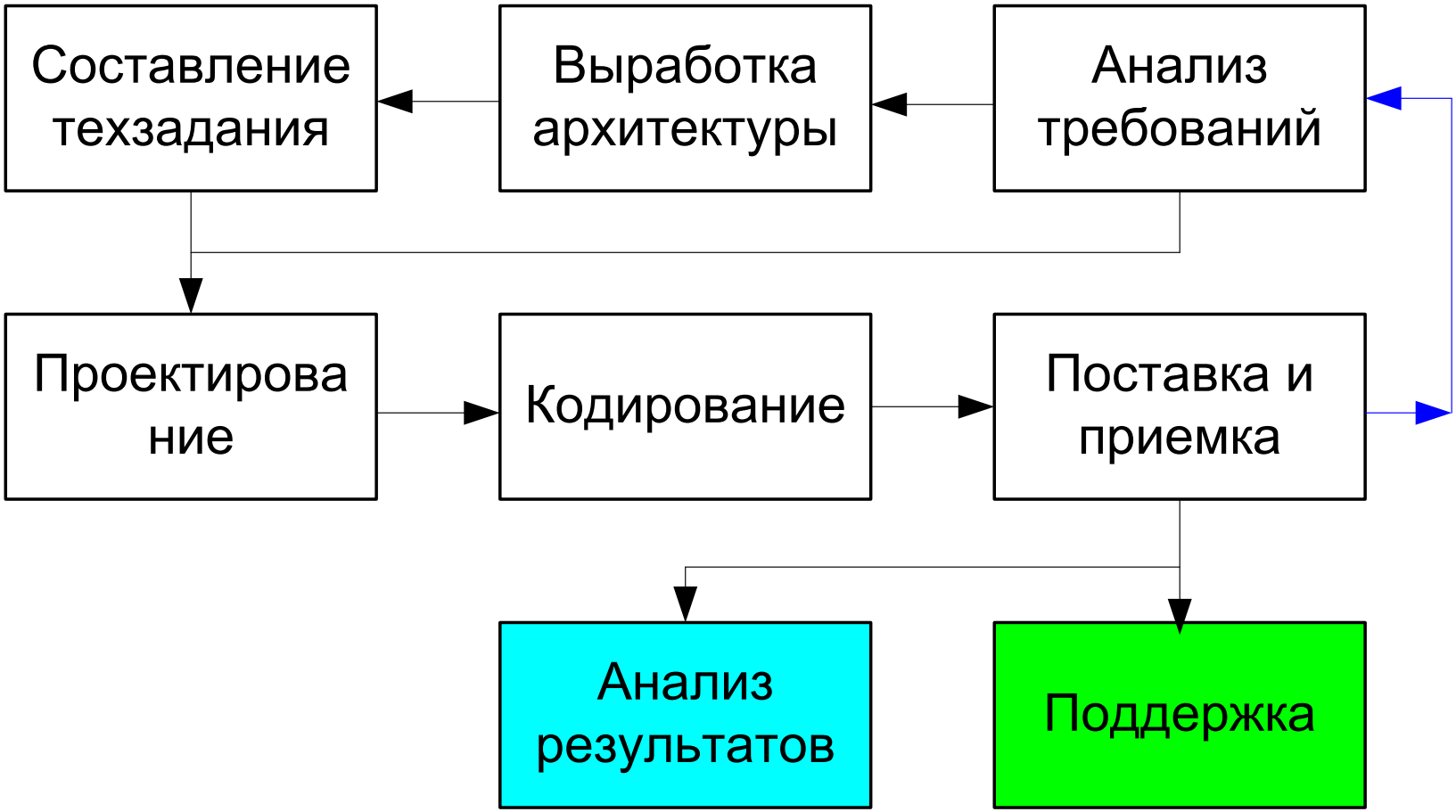
Процесс разработки ПО

«Design and programming are human activities.

Forget it – and all is lost.»

B. Stroustrup

Жизненный цикл IT проекта



Итеративно-инкрементная модель жизненного цикла ИТ-проекта

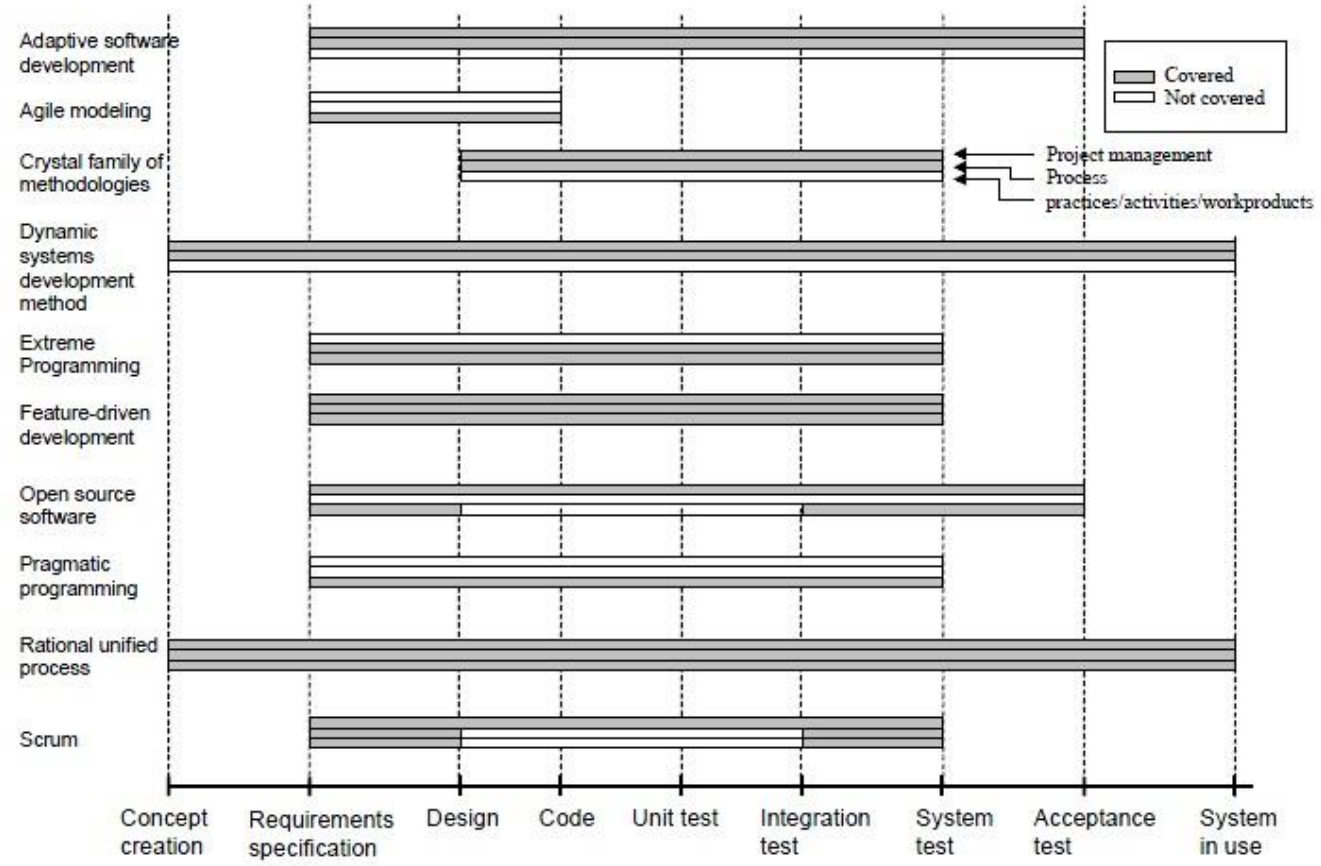
Модель жизненного цикла

- 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

- Полные
  - RUP (Booch, Rumbaugh, Jacobson)
  - MSF (Microsoft)
  - OpenUP (Eclipse Foundation)
- Agile (легковесные):
  - Agile Unified Process (AUP)
  - eXtreme Programming (Beck, Cunningham, Martin, Fowler, Cockburn)
  - Scrum, Kanban, Scrumban … etc.

Agile vs. RUP



Abrahamson P, Salo O, Ronkainen J, Warsta J (2002)
Agile software development methods: Review and analysis
http://www.vtt.fi/inf/pdf/publications/2002/P478.pdf

Профессиональные стандарты РФ в ИТ

- Утверждены в 2014г
- Определяют стандарты для должностей:
  - Системный аналитик – 06.022
  - Архитектор программного обеспечения – 06.003
  - Руководитель проектов в области информационных технологий – 06.016
  - и пр.

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 серия 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 PM
- 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	Инструкция по установке
Сопровождение	Анализ процесса эксплуатации, предложения по изменению системы

2. Анализ требований

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

Анализ требований

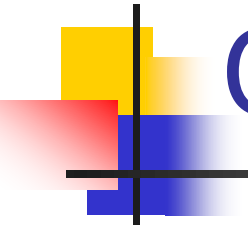
Анализ требований - первая фаза итеративно-инкрементного процесса разработки ПО

В крупных проектах выполняется Системным Аналитиком (часто, не одним)

В проектах среднего и малого размера может выполняться Архитектором

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

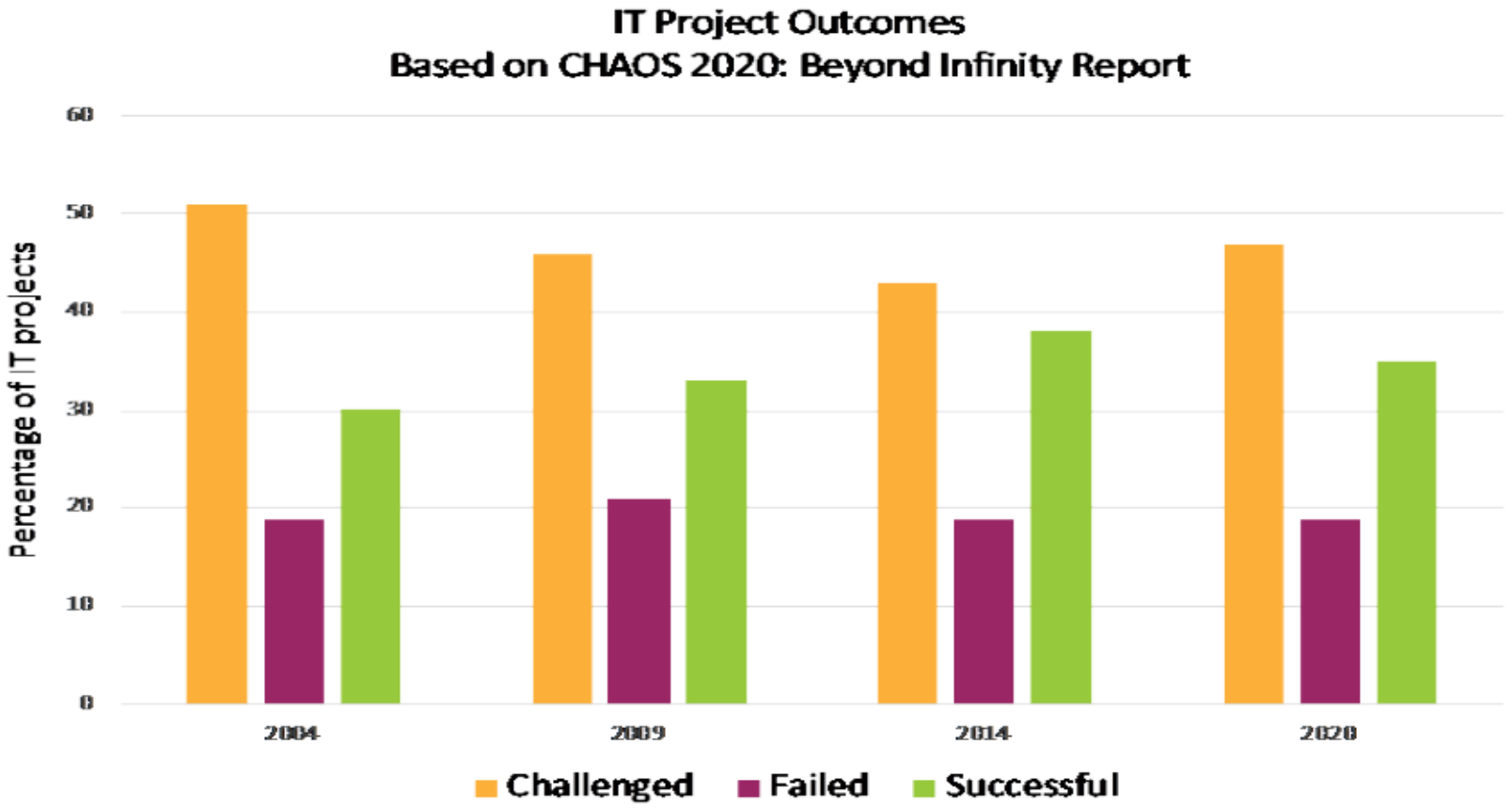
Standish Group CHAOS report 1994 (8000 проектов)

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

CHAOS report 2015:

- 19% - провалены
- 45% - имели сложности со сроками/бюджетами
- 36% - завершены успешно

CHAOS 2020 – не сильно лучше



Herb Krasner, Consortium for Information & Software Quality (CISQ)
"The Cost of Poor Software Quality in the US: A 2020 Report", p.14

И еще о важности требований

Источники проблем в проектах

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

(CHAOS report 1994)

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

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

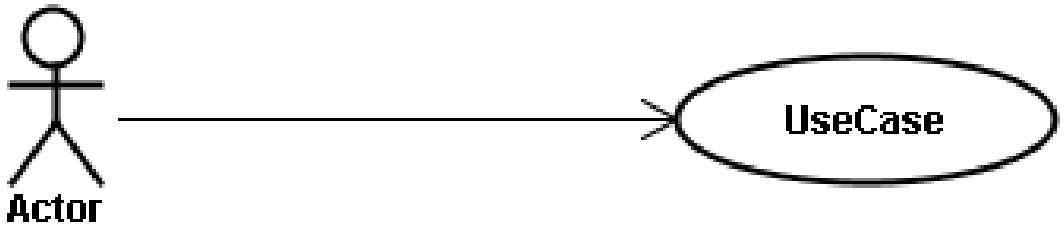
- Непрерывно возрастающая сложность ПО
- Хороший метод борьбы со сложностью исследуемого объекта - формальная модель
- Проблему сложности математических вычислений решил математический формализм (цифры->арифметика->алгебра, мат.анализ, функциональный анализ, мат.логика ...)
- Требования к ПО, как и программный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

Unified Modeling Language

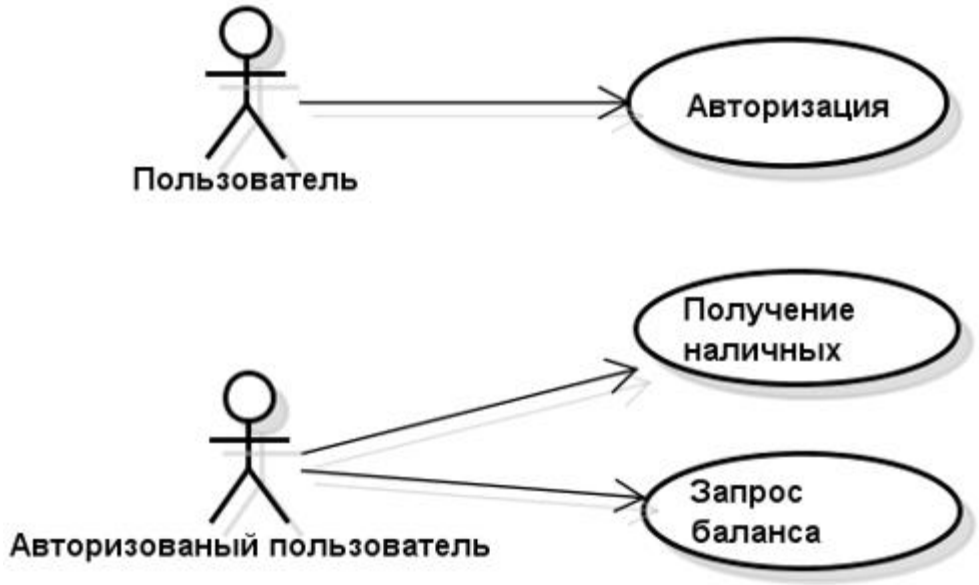
- Язык моделирования программных систем (и не только)
- Не является языком программирования
- Определяет нотацию и ее семантику
- Имеет UML-метамодель, описывающую семантику UML на языке UML
- Предоставляет возможности для расширения стандартной семантики
- OMG Unified Modeling Language Specification v 2.5
  - http://www.omg.org/spec/UML/2.5/

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
- Use-case иногда называют «прецедентами».
- Представление в UML:



Пример



Пользователь банкомата может пройти авторизацию (ввести PIN-код своей карты)
Авторизованый пользователь может выполнить операции:

- Получение наличных
- Запрос баланса

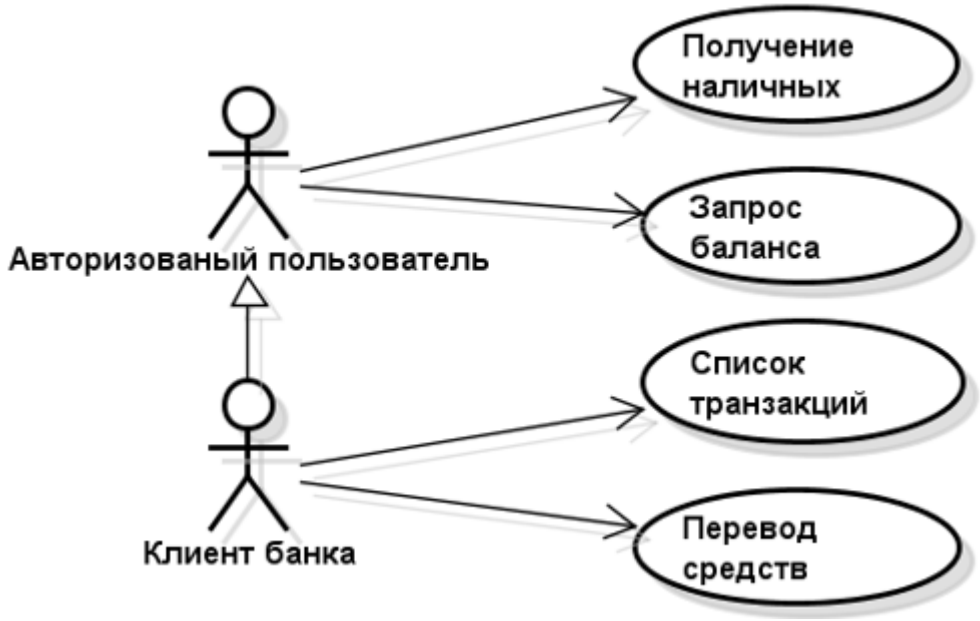
Вариант использования (use case)

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

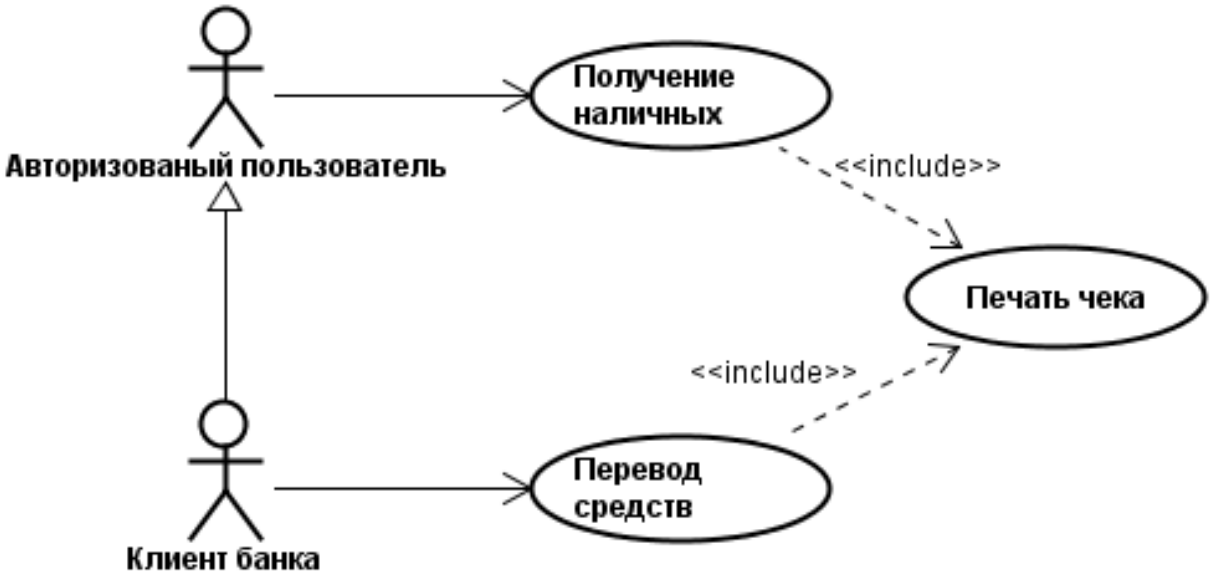
Иерархия Акторов



Различные акторы могут иметь набор общих use-case. Любой авторизованый пользователь банкомата может «получить наличные» или «запросить баланс», но если он еще и Клиент банка-владельца банкомата, ему доступны «Список транзакций» и «Перевод средств»

Включаемые use-cases

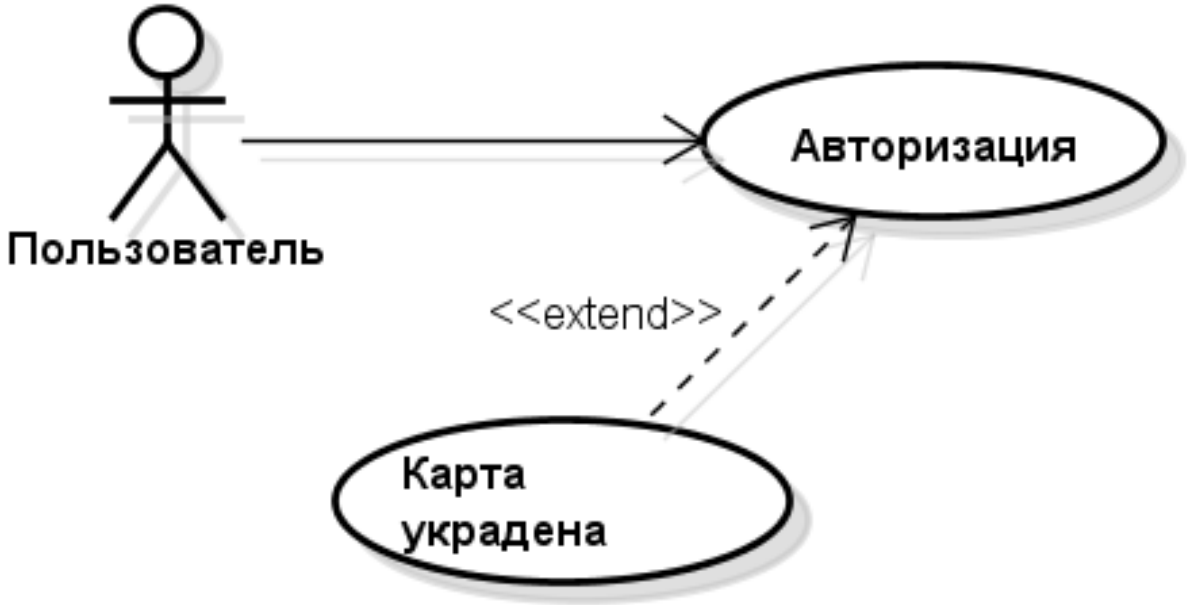
- Различные use-cases могут иметь общие части, часто исполняемые только в контексте другого use-case (абстрактный use-case)
- Stereotype: <<include>>



Пользователь банкомата при переводе средств со счета на карту или другой счет получает чек. При этом «Печать чека» не может быть исполнен иначе как в контексте выполнения другого use-case

Расширение use-case

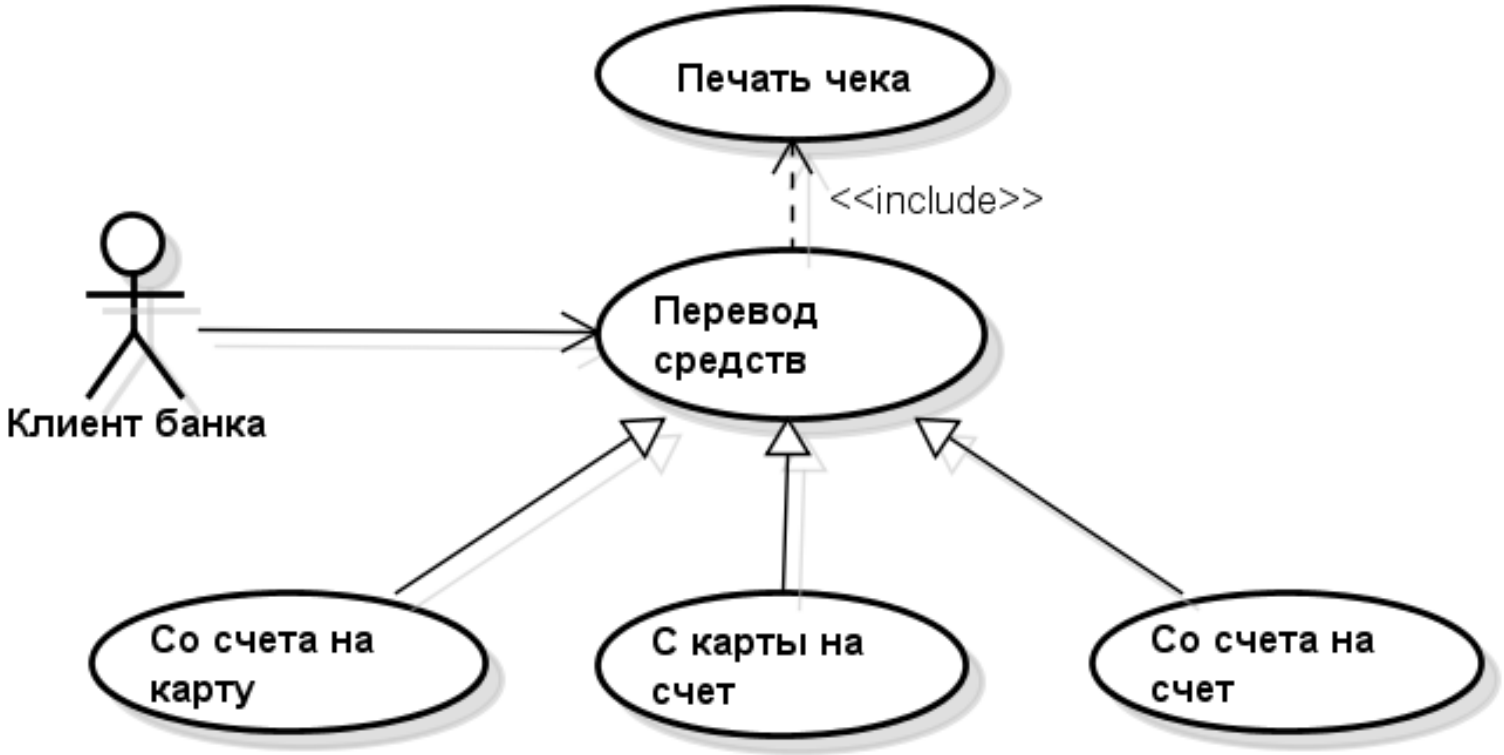
Stereotype: <<extend>>
Некоторые use-case могут вызываться в контексте других только при некоторых условиях



Если в процессе авторизации мы получаем от банка сообщение что карта украдена – нужна особая обработка этой ситуации (поднять тревогу, не отдавать карту...)

Генерализация use-case

Разные use-case могут иметь некоторую общность исполнения. Общая часть может быть генерализована в обобщающий use-case.



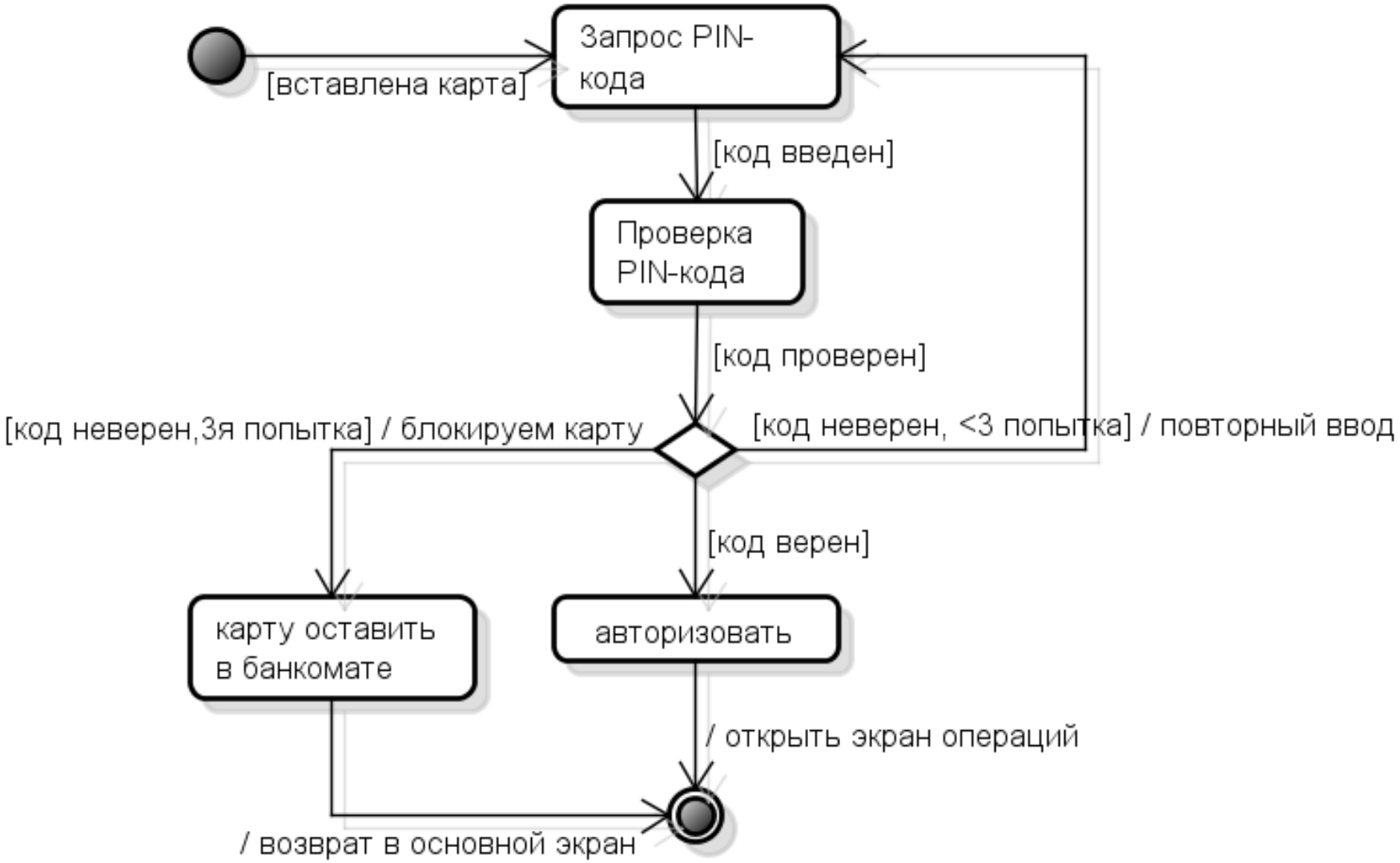
Документирование use-case

- Имя
- Актор (акторы) – роли в системе, вовлеченные в UC
- Цель (value) актора - 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)
  - Альтернативный сценарий №2 …

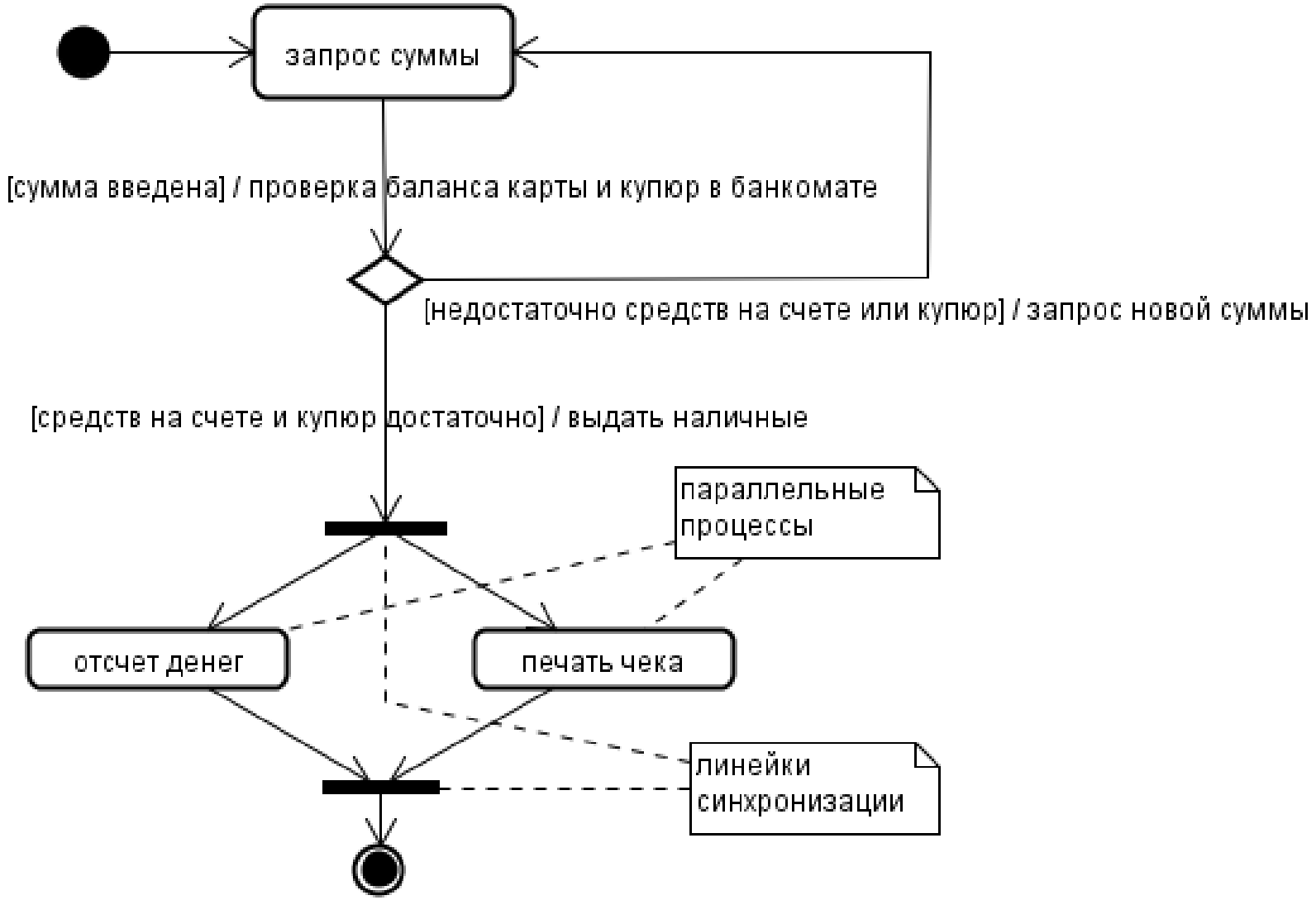
Диаграммы деятельностей

- Используются для описания сценариев
- Описывают последовательности действий
- Activity - деятельность
- Transition – переходы между деятельностями
  - Guard condition – условие перехода
  - Action – действие при переходе
- Decision node – блок принятия решения
- Fork node – переход к параллельным деятельностям
- Sync node – линейка синхронизации параллельных деятельностей

Пример: банкомат. Авторизация



Пример: банкомат. Получение наличных



Типичные ошибки

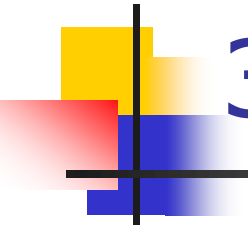
- Тип 1: сценарий принят за use-case
  - См. пример на следующем слайде
- Тип 2: избыточная декомпозиция
  - Включаемый use-case имеет только один включающий
- Тип 3: супер-абстракция (игра воображения)
  - Актор, не имеющий собственных use-case
  - Use-case, не имеющие ни актора, ни базового/производного или включающего use-case

use case и сценарий – не надо путать!

- Самая распространенная ошибка: путать сценарий и use-case
- Например, use-case login часто имеет 3 сценария:
  - Основной: вводим пользователя и пароль и входим в систему
  - Альтернативный 1: имя пользователя или пароль неверны, 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, что 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** – авторизуется в системе.
- Замечание: Сценарий 2 может, если нужно, вызывать отдельный (расширяющий) use-case «Сменить пароль»

Контрольные вопросы

- В чем отличие use-case от сценария?
- Приведите примеры use-case и их сценариев на примере известных систем (Gmail, MS Word, любой другой)
- Gmail: Каким отношением могут быть связаны use-case «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 (Уникальностью)
- Схожие объекты объединяются в классы

Основные понятия ООП

- **Класс** – абстракция (модель данных и поведения) некоторого множества объектов, обладающих одинаковым поведением
- **Объект** – сущность, обладающая поведением, состоянием и уникальностью
- **Instance** – объект, созданный во время исполнения программы
- Cat cat=new 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
  - Сокрытие особенностей реализации, отделение контрактных обязательств абстракции от их реализации
- Иерархия
  - Упорядочение абстракций, расположение их по уровням
- Модульность
  - Разделение системы на внутренне связные модули, которые слабо связаны между собой

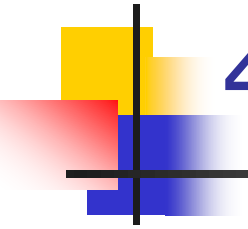
Это и есть ОО декомпозиция

- 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

Иными словами: ОО-способ ведения рассуждений, ОО-средство борьбы со сложностью

Методы 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



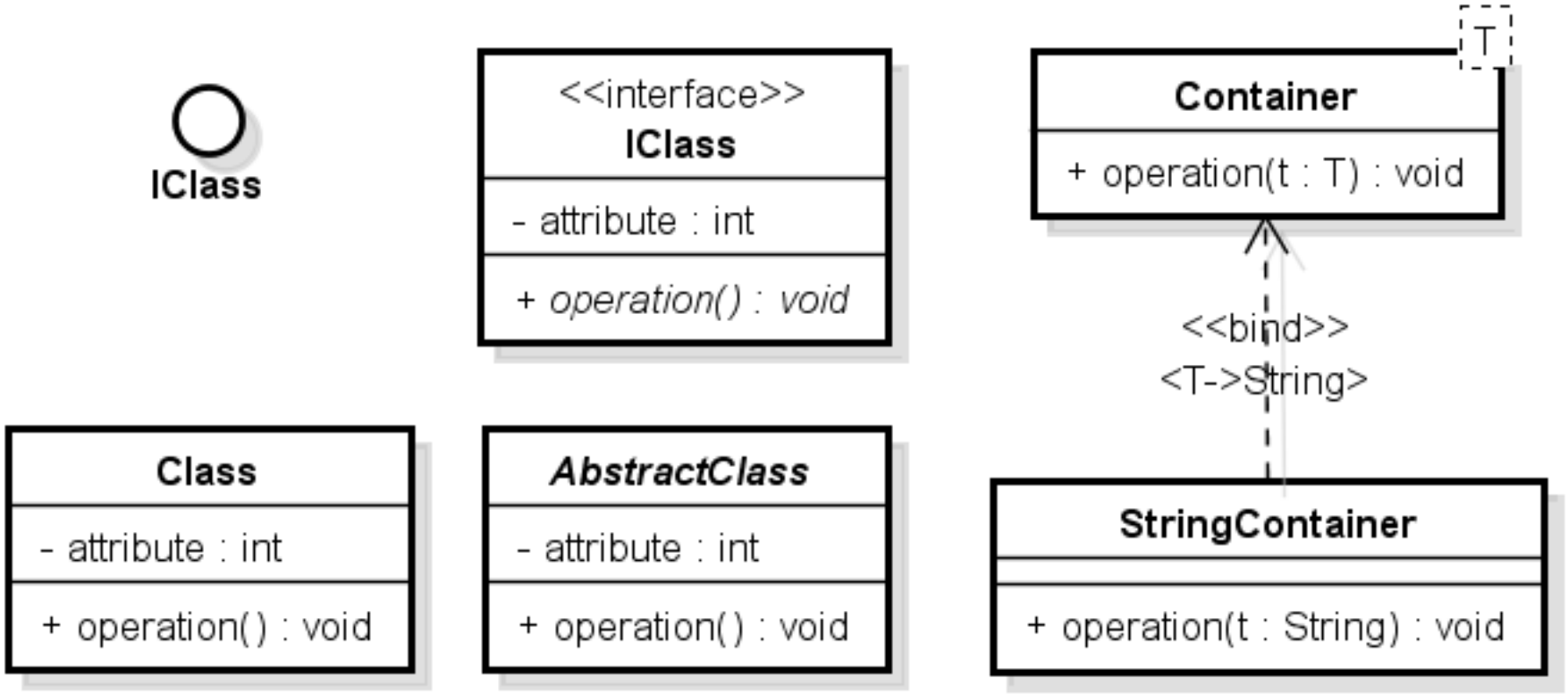
4. Классы и пакеты

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

Классы в UML

- Class
  - Абстракция данных 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

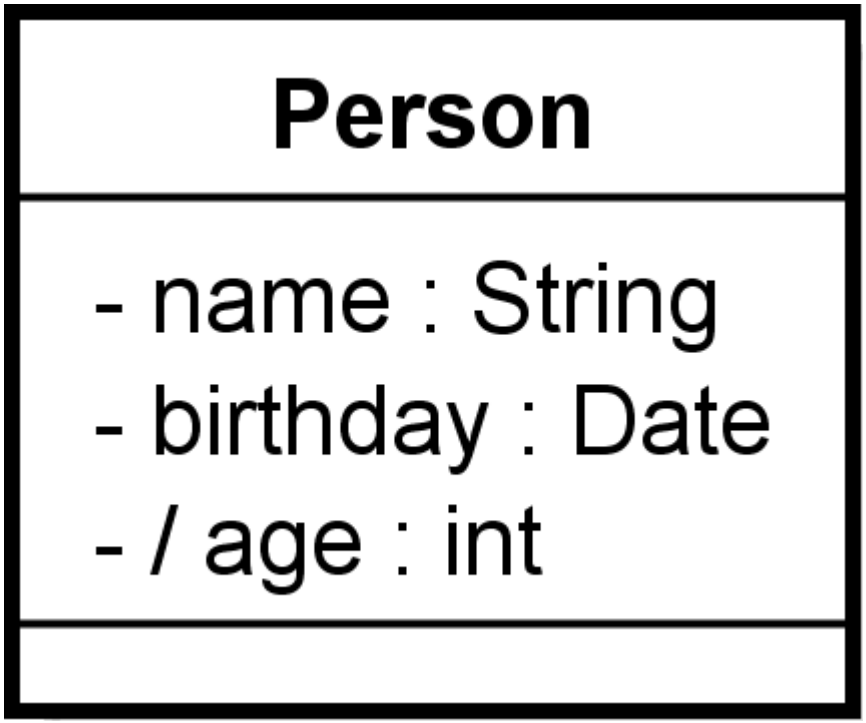


Атрибуты классов

- Attribute
  - атрибут (поле)
- Class attribute
  - атрибут класса (static)
- Derived attribute
  - производный (вычислимый) атрибут
- Export control
  - доступ (public:+, protected:#, private:-, package:~)
- Aggregation
  - способ включения (none, composite, aggregate)
- Может иметь стереотип

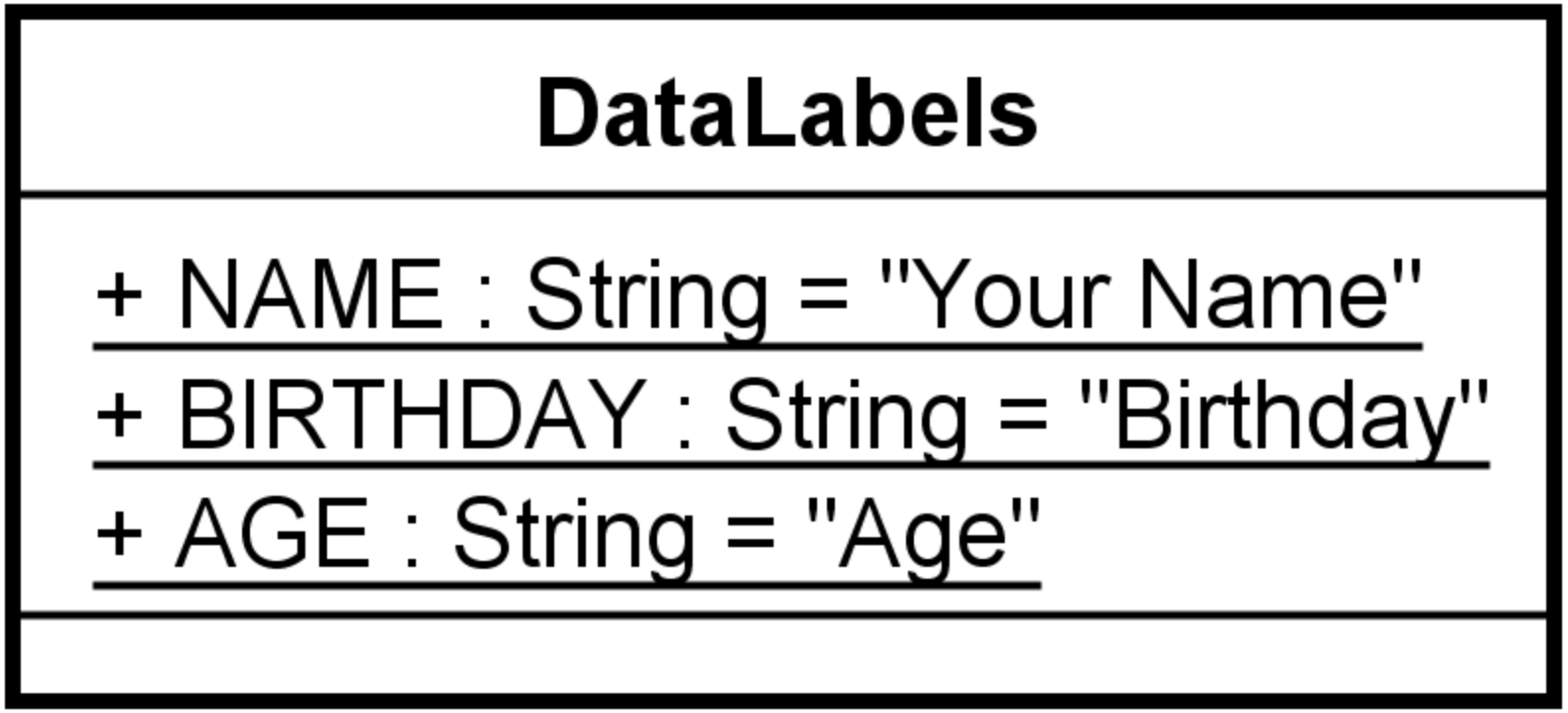
Syntax: <role_name>:<class_name><=default_value>

Атрибуты классов



name, birthday – атрибуты
age – производный атрибут (вычисляется через birthday)

Атрибуты классов



NAME, BIRTHDAY, AGE - атрибуты класса (static)

Методы(операции)

- Method (operation) – метод
- Могут быть static, abstract, final (leaf)
- Видимость: public, protected, private, package
- Синтаксис:
  - <<stereotype>> name(<parameters>) : <return type>
- Параметры: parameter_name : type

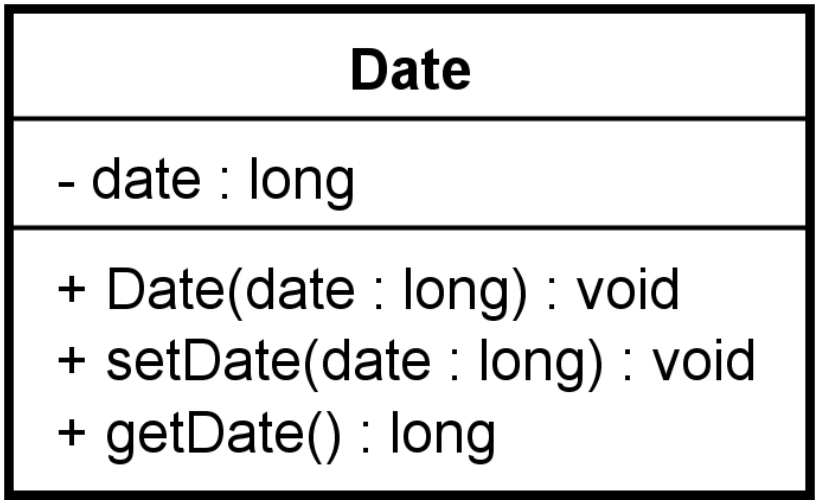
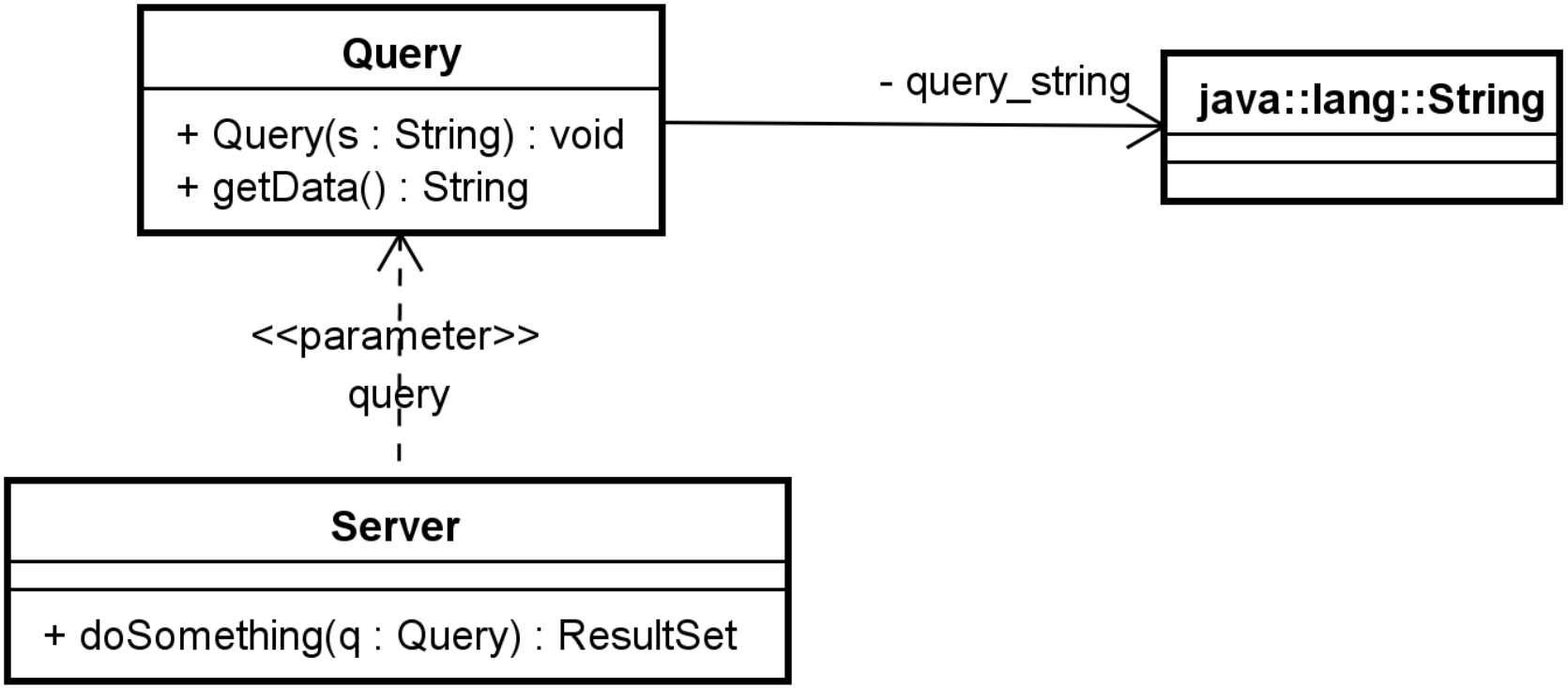


Диаграмма классов

- определяет типы (классы) объектов системы и статические связи между ними

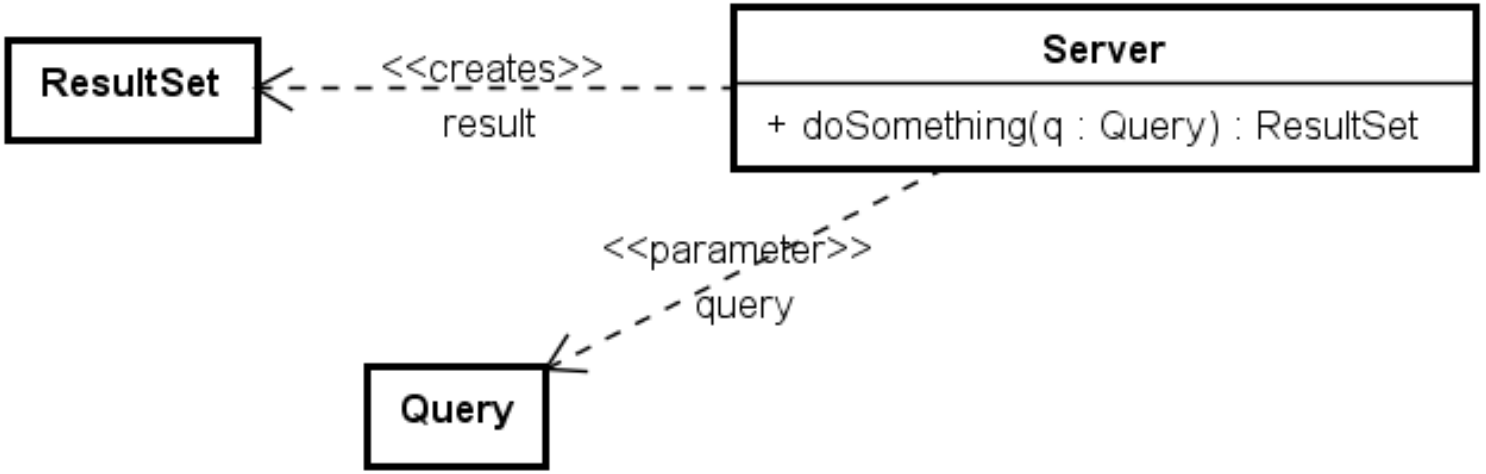


Связи между классами

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

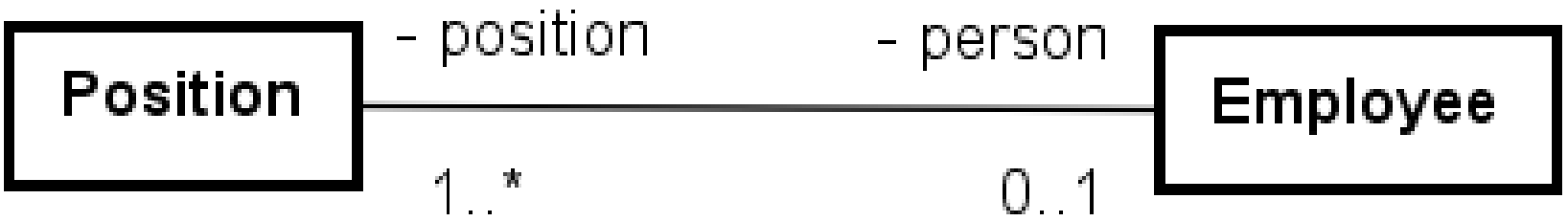
Dependency

- 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 связанности
- 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

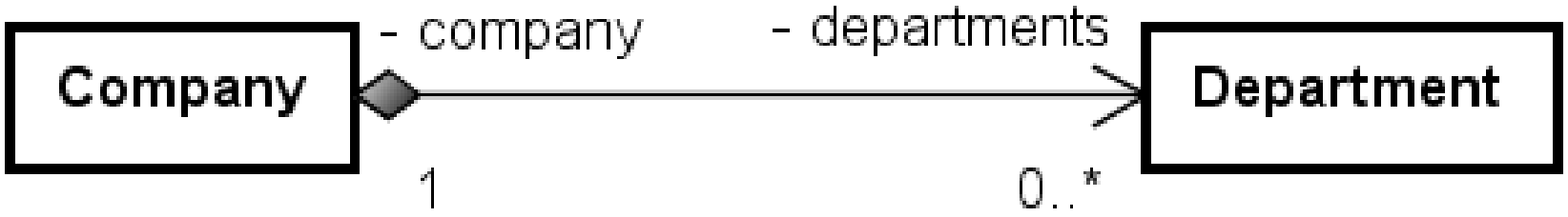
- Ассоциация может иметь выделенное направление
  - Должность связана с базовым тарифом оплаты
  - Тариф оплаты никак не связан с конкретной должностью

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 в нескольких журналах

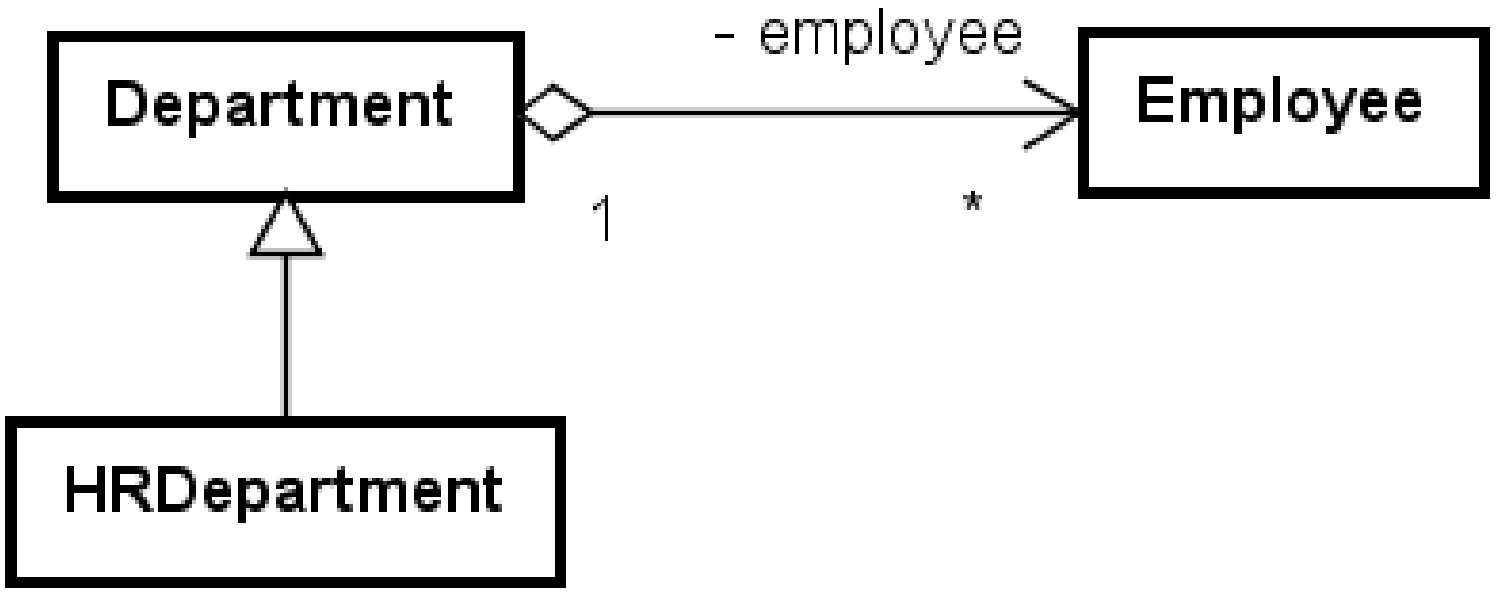
Composition

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



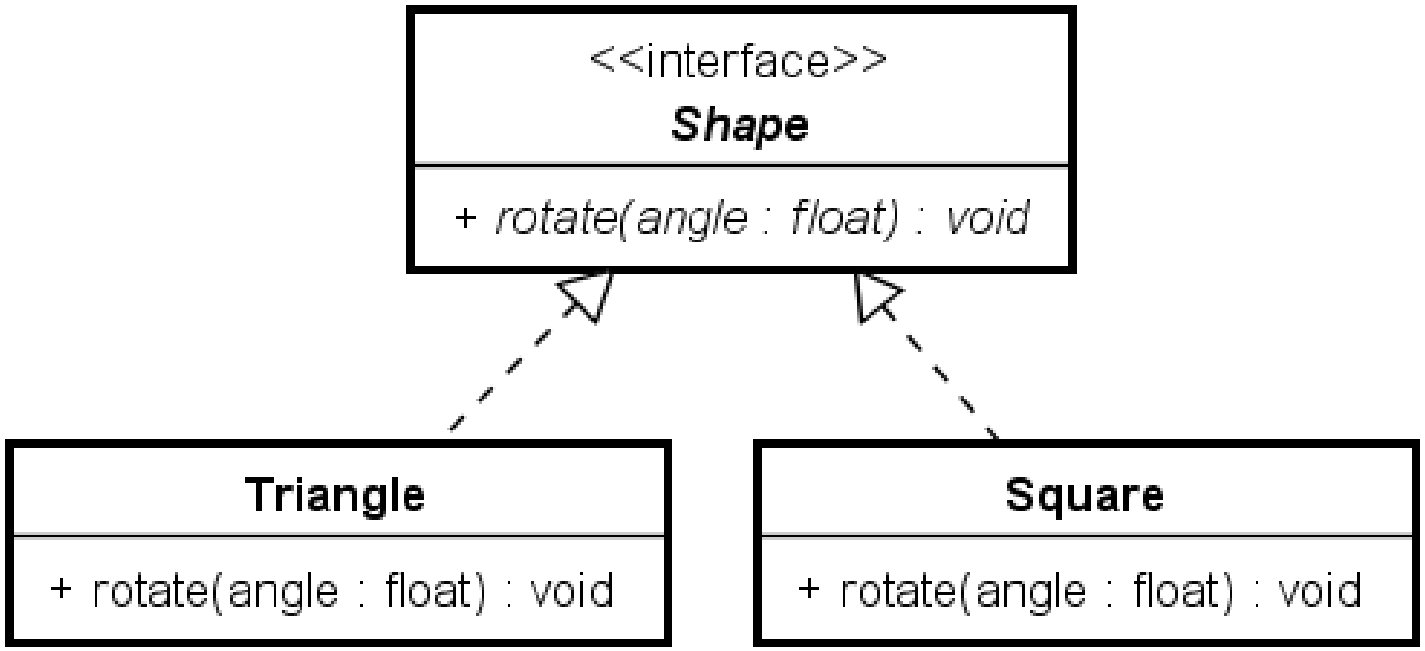
Generalization

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



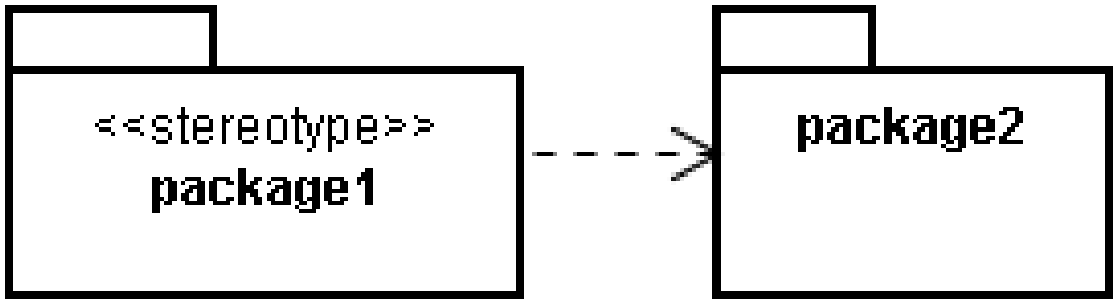
Realization

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



Пакеты в UML

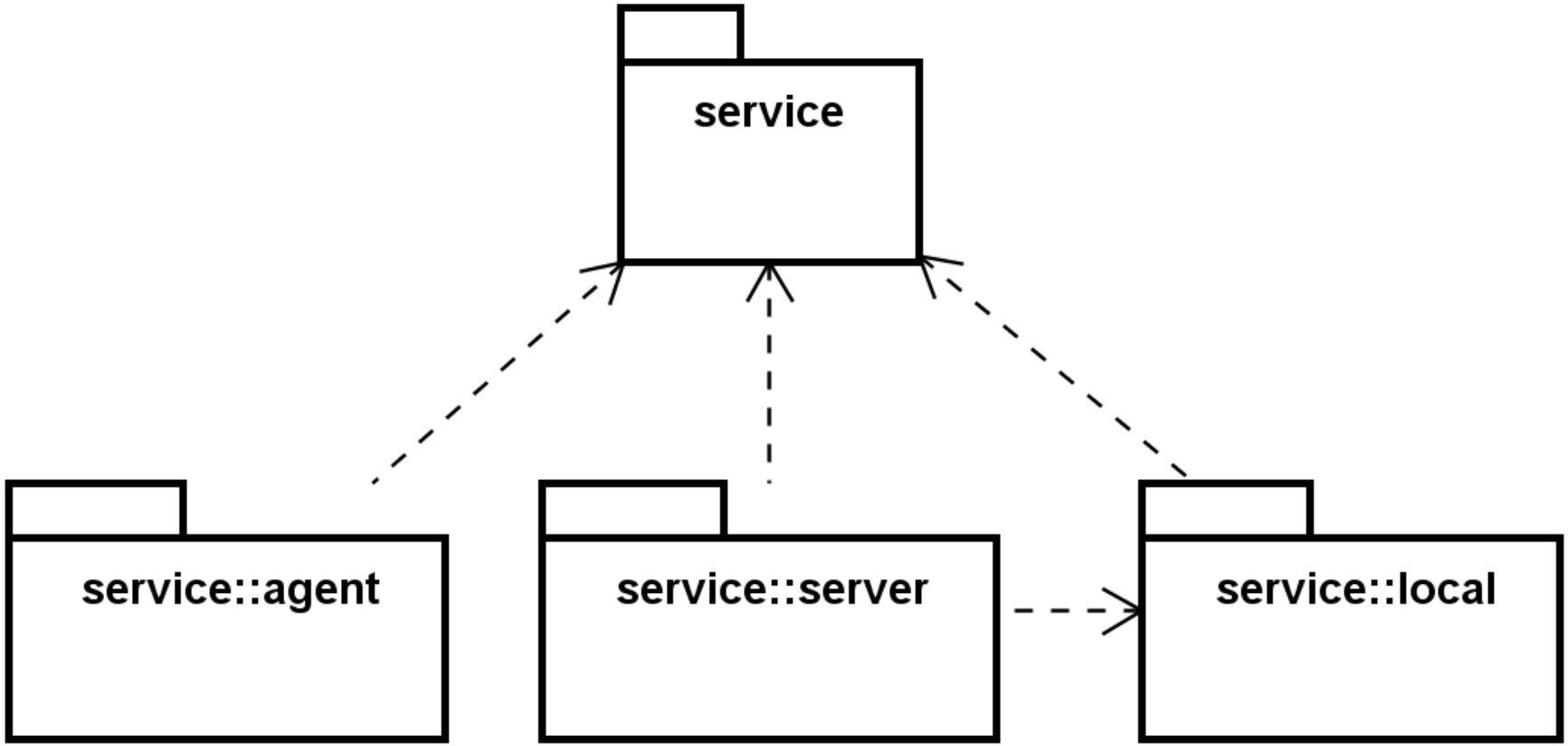
- **Package** – пакет. Общий механизм организации элементов модели в группы
- Имеет имя
- Определяет пространство имен
- Может быть стереотипирован
- Может быть импортирован другим пакетом



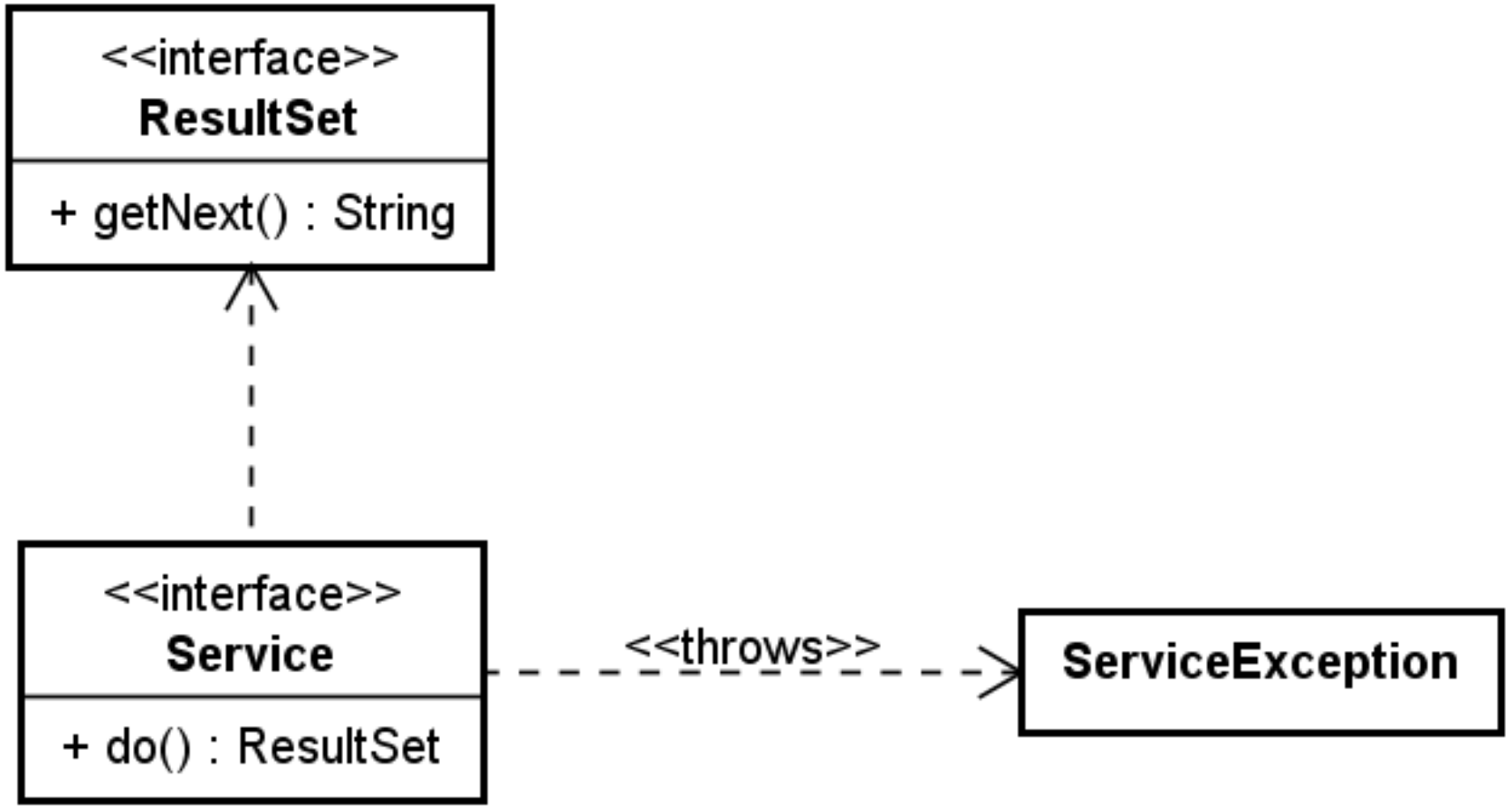
Пакеты классов

- Пакет классов – группа тесно связанных классов
- Исключение: пакеты утилитных классов
- Замечание: В контексте связей в OOAD применяются два термина, которые необходимо различать:
  - Cohesion – единство, спаянность (у классов внутри пакета должно быть высоким)
  - Coupling – сцепление, сопряжение (должно быть низким между пакетами)

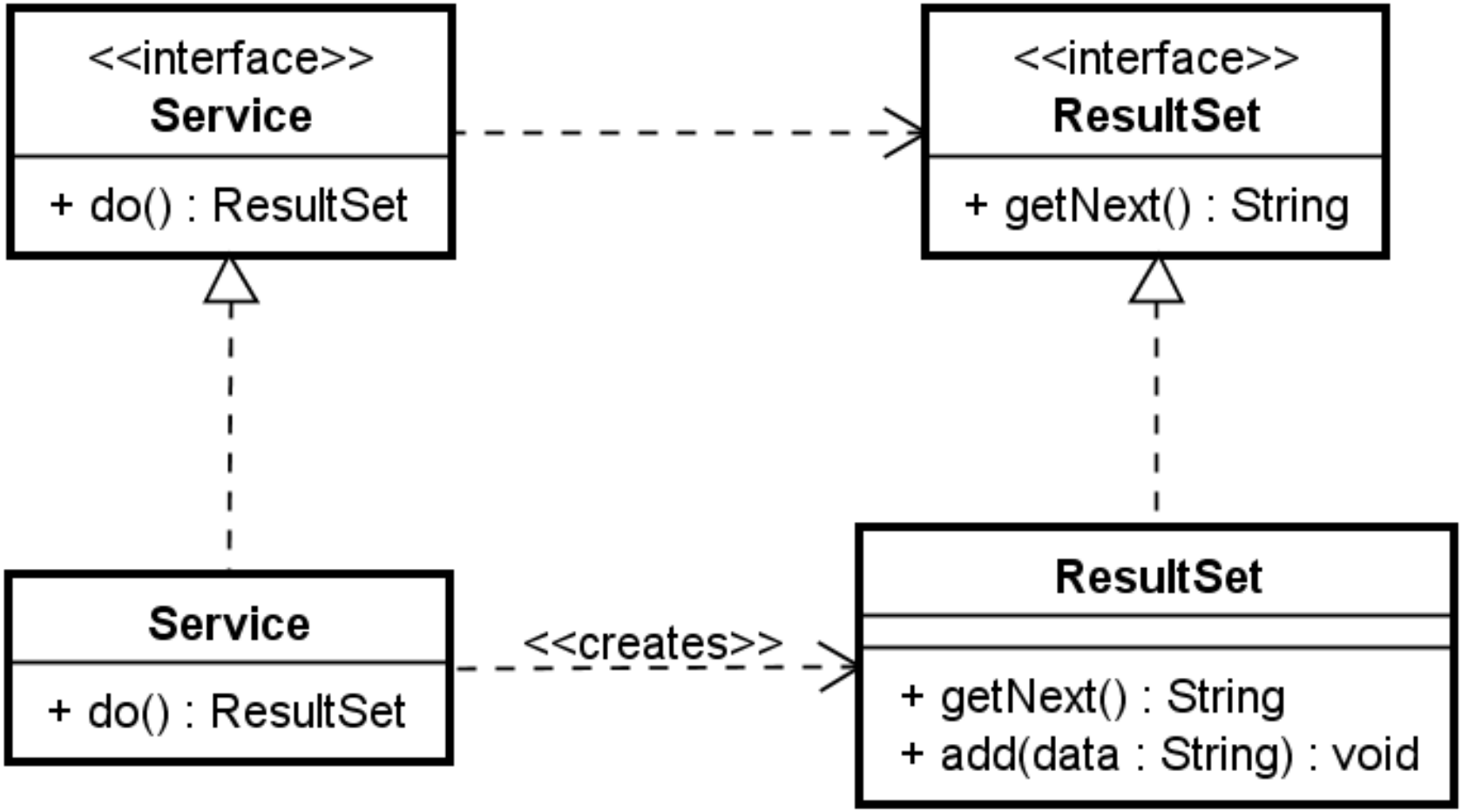
Диаграмма пакетов



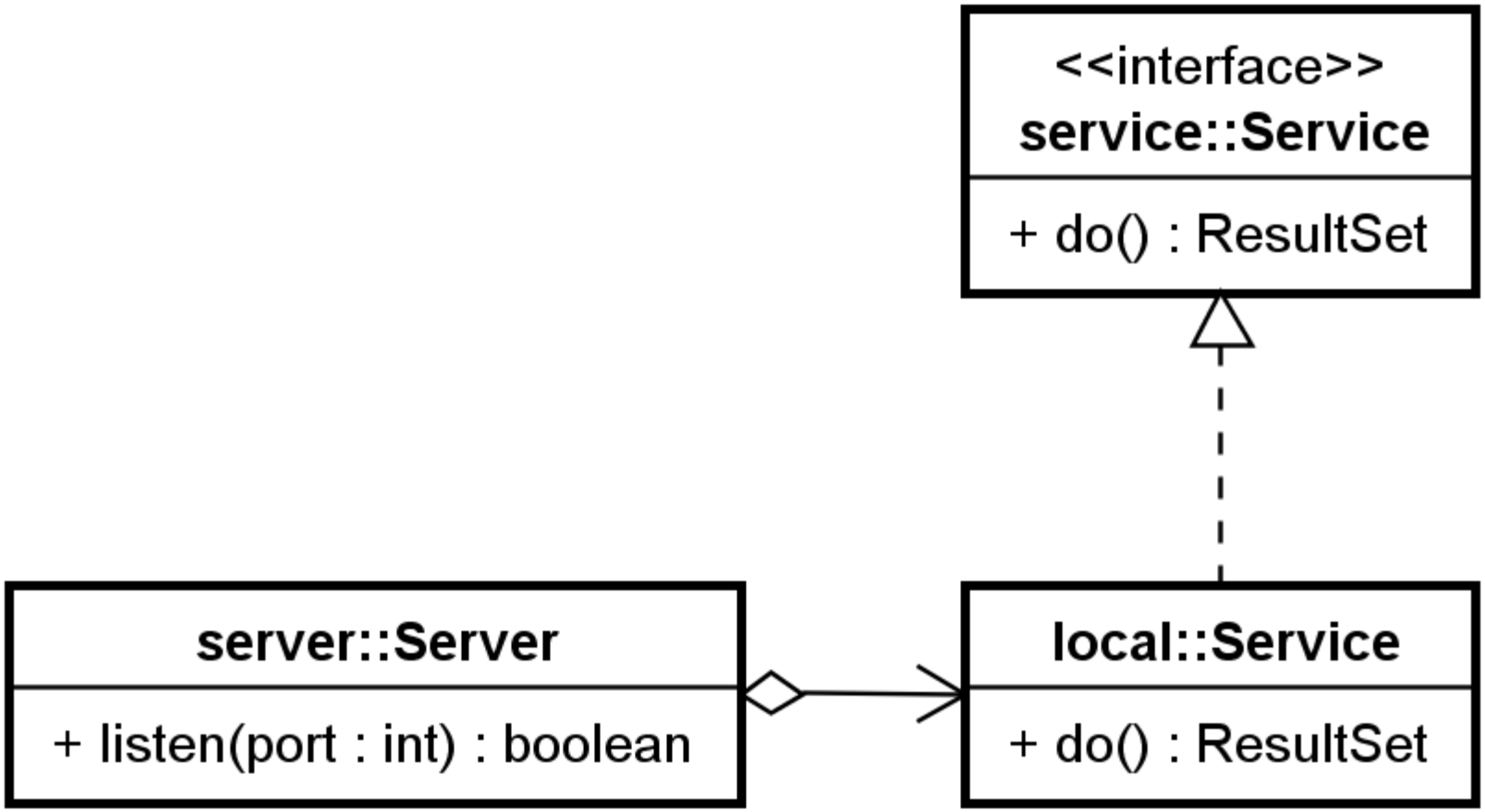
пакет service



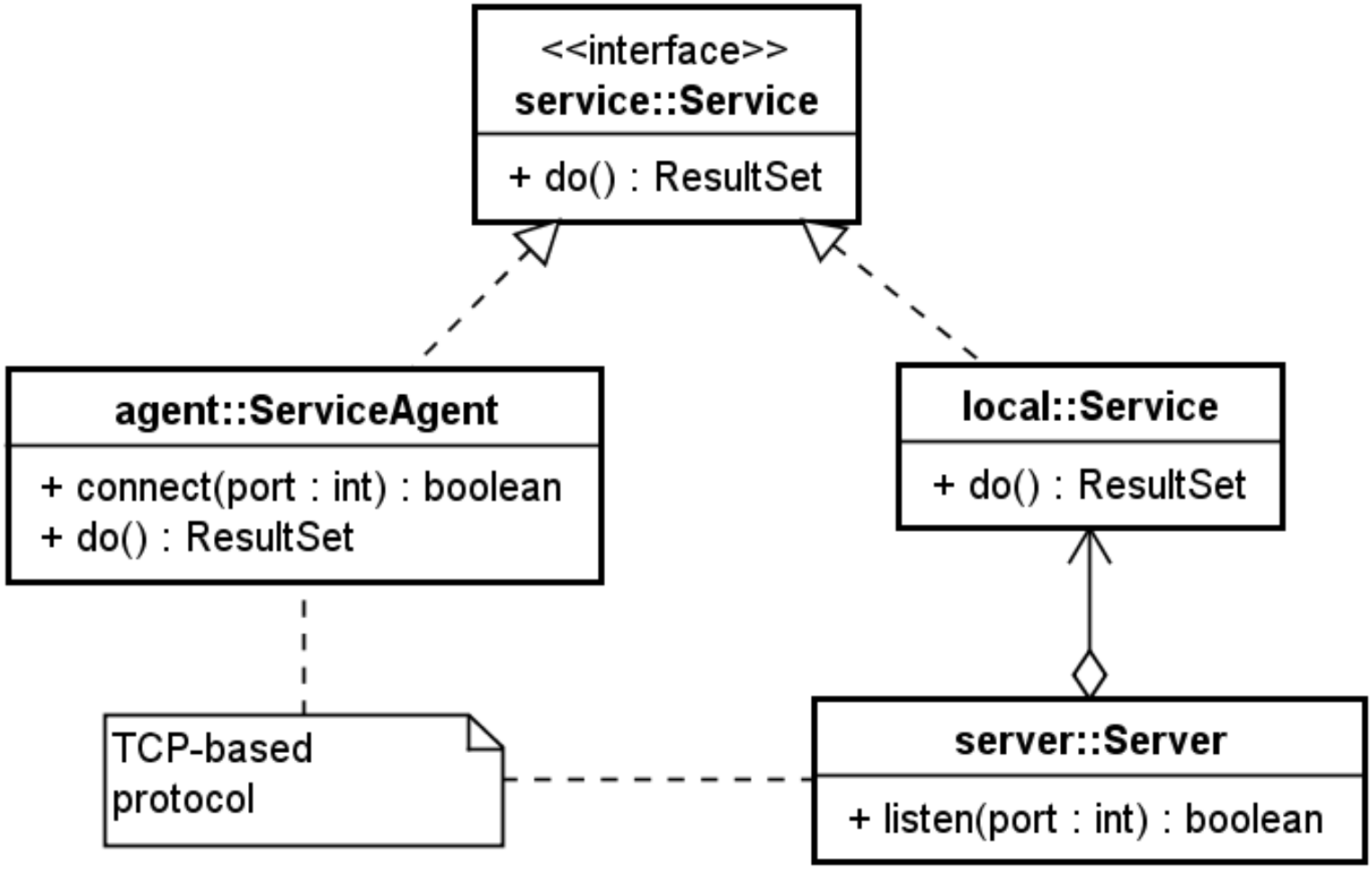
пакет service::local



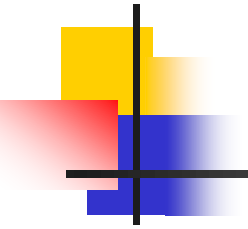
пакет service::server



пакет service::agent



стереотипы пакетов



- system – система
- subsystem – подсистема
- facade – представление другого пакета
  - Например, пакет внешних интерфейсов подсистемы
- framework – переиспользуемый набор классов
- 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?