**Жизненный цикл программного**

**Жизненный цикл программного обеспечения** (ПО) — период времени, который начинается с момента принятия решения о необходимости создания [программного продукта](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D1%8B%D0%B9_%D0%BF%D1%80%D0%BE%D0%B4%D1%83%D0%BA%D1%82) и заканчивается в момент его полного изъятия из эксплуатации[[1]](https://ru.wikipedia.org/wiki/%D0%96%D0%B8%D0%B7%D0%BD%D0%B5%D0%BD%D0%BD%D1%8B%D0%B9_%D1%86%D0%B8%D0%BA%D0%BB_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F#cite_note-1).

Частный случай [жизненного цикла системы](https://ru.wikipedia.org/wiki/%D0%96%D0%B8%D0%B7%D0%BD%D0%B5%D0%BD%D0%BD%D1%8B%D0%B9_%D1%86%D0%B8%D0%BA%D0%BB_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D1%8B) (в данном случае [программной системы](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0)).

1. Формирование требований к АС
   1. Обследование объекта и обоснование необходимости создания АС
   2. Формирование требований пользователя к АС
   3. Оформление отчета о выполнении работ и заявки на разработку АС
2. Разработка концепции АС
   1. Изучение объекта
   2. Проведение необходимых научно-исследовательских работ
   3. Разработка вариантов концепции АС и выбор варианта концепции АС, удовлетворяющего требованиям пользователей
   4. Оформление отчета о проделанной работе
3. Техническое задание
   1. Разработка и утверждение технического задания на создание АС
4. Эскизный проект
   1. Разработка предварительных проектных решений по системе и её частям
   2. Разработка документации на АС и её части
5. Технический проект
   1. Разработка проектных решений по системе и её частям
   2. Разработка документации на АС и её части
   3. Разработка и оформление документации на поставку комплектующих изделий
   4. Разработка заданий на проектирование в смежных частях проекта
6. Рабочая документация
   1. Разработка рабочей документации на АС и её части
   2. Разработка и адаптация программ
7. Ввод в действие
   1. Подготовка объекта автоматизации
   2. Подготовка персонала
   3. Комплектация АС поставляемыми изделиями (программными и техническими средствами, программно-техническими комплексами, информационными изделиями)
   4. Строительно-монтажные работы
   5. Пусконаладочные работы
   6. Проведение предварительных испытаний
   7. Проведение опытной эксплуатации
   8. Проведение приёмочных испытаний
8. Тестирование АС.
9. Сопровождение АС.
   1. Выполнение работ в соответствии с гарантийными обязательствами
   2. Послегарантийное обслуживание

**Гибкая разработка ПО**

[Гибкая](https://docs.microsoft.com/ru-ru/devops/plan/what-is-agile) разработка — термин, используемый для описания итеративной разработки программного обеспечения. итеративная разработка программного обеспечения сокращает жизненный цикл DevOps, выполняя работу с более мелкими приращениями, обычно называемыми спринтами. Как правило, продолжительность спринта составляет от 1 до 4 недель. Разработка по методике Agile часто противопоставляется традиционной или каскадной разработке, при которой крупные проекты планируются заранее и выполняются в соответствии с намеченным планом.

Для предоставления качественного рабочего кода в каждом спринте команда, использующая гибкую разработку, должна работать в ускоренном темпе. Все программирование, тестирование и все проверки качества должны выполняться в каждом спринте. Если группа не настроена должным образом, результаты могут оказаться небольшими. Хотя эти приемы предлагают отличные возможности для обучения, было бы лучше изучить некоторые ключевые уроки перед началом работы?

В этой статье рассматриваются несколько ключевых факторов успеха для гибких команд разработки:

Уточнение невыполненной работы

Интеграция с ранними и часто используемыми

Сокращение технической задолженности

Уточнение невыполненной работы

Группа гибкой разработки работает с невыполненной наработкой требований, часто называемых пользовательскими историями. Приоритеты в списке невыполненных работ расставляются таким образом, что самые важные пользовательские истории находятся в его верхней части Владелец продукта владеет списком невыполненной работы, добавляет, изменяет и переопределяет приоритеты пользовательских историй в зависимости от потребностей клиента.



Одним из самых серьезных факторов, снижающих производительность команды, работающей по методике Agile, является плохо определенный перечень невыполненной работы. Группа не может ожидать согласованного создания высококачественного программного обеспечения для каждого спринта, если у них нет четко определенных требований.

Задача владельца продукта заключается в том, чтобы убедиться в том, что каждый спринт-инженеры четко определили пользовательские истории для работы. Пользовательские истории в верхней части невыполненной работы всегда должны быть готовы для выполнения команды. Это называется уточнением невыполненной работы. Обеспечение готовности невыполненной работы для команды гибкой разработки требует усилий и дисциплины. К счастью, это хорошо.

При уточнении невыполненной работы необходимо помнить о некоторых ключевых вопросах.

Уточнение пользовательских историй часто является длительным действием. Для создания элегантных пользовательских интерфейсов, привлекательных дизайнов экрана и решений по детонкости клиентов требуется время и энергия. Тщательное владельцы продуктов расширяют пользовательские истории 2-3 спринтов заранее. Они учетируют итерации проекта и отзывы клиентов. Они работают, чтобы гарантировать, что каждая пользовательская история является тем, что гибкая группа подходит для доставки клиенту.

Пользовательская история не поддается уточнению, если только команда не говорит о ней. Группа должна проанализировать пользовательскую историю и согласиться с ее готовностью к работе. Если группа не наблюдала пользовательскую историю до первого дня спринта, это большой красный флаг.

Пользовательские истории, последующие за невыполненной работой, могут оставаться неоднозначными. Не тратить время на уточнение элементов с более низким приоритетом. Не забудьте полагаться на начало невыполненной работы.

Интеграция с ранними и часто используемыми

[Непрерывная интеграция](https://docs.microsoft.com/ru-ru/devops/develop/what-is-continuous-integration) и [непрерывная поставка](https://docs.microsoft.com/ru-ru/devops/deliver/what-is-continuous-delivery) (CI/CD) настраивает группу для быстрого развития гибкой разработки. Как можно скорее, автоматизируйте конвейеры сборки, тестирования и развертывания. Это должно быть одно из первых действий, которое Группа настраивает при запуске нового проекта.

С помощью службы автоматизации команда будет избегать выполнения процессов ручного развертывания, подверженных ошибкам и требующих много времени. Так как команды выпускают каждый спринт, просто не нужно выполнять эту задачу вручную.

CI/CD также влияет на архитектуру программного обеспечения. Это гарантирует, что вы поставляете создаваемое и развертываемое программное обеспечение. При реализации сложного развертывания команды немедленно становятся недоступными по мере сбоя сборки и развертывания. CI/CD заставляет группу устранять проблемы развертывания по мере их возникновения, гарантируя, что продукт всегда готов к поставке.

Существует несколько ключевых операций CI/CD, критиалли важных для эффективной разработки гибких задач.

Модульное тестирование. Модульные тесты являются первой защитой от ошибок человека. Модульные тесты должны рассматриваться как часть кода и возвращены вместе с кодом. Выполняемые модульные тесты должны быть частью каждой сборки. Неудачные модульные тесты означают неудачную сборку.

Автоматизация сборки. Система сборки должна автоматически получать код и тесты непосредственно из системы управления версиями при выполнении сборок.

Политики ветвления и сборки. Настройте ветви и политики сборки для автоматической сборки по мере того, как команда выполняет проверку кода в определенной ветви.

Развертывание в среде. Настройте конвейер выпуска, который автоматически развертывает построенные проекты в среде, которая имитирует рабочую среду.

Сокращение технической задолженности

Благодаря личным финансам проще не долагаться на долгие, чем на их основе. То же правило применяется к технической задолженности. Технический долг включает все, что должна сделать команда для развертывания кода, пригодного для производственного применения, и обеспечения его выполнения в рабочей среде. Примерами могут быть ошибки, проблемы с производительностью, эксплуатационные проблемы, Специальные возможности и другие.

Для поддержания поверх технической задолженности требуется. Существует множество причин отложить исправление ошибок. Он хорошо работает с функциями и не учитывает долг. К сожалению, кто бы то ни было платить за техническую задолженность раньше или позднее. Как и в случае с финансовыми обязательствами, в сфере технической задолженности больше не будет взиматься плата. Владелец смарт-продукта работает со своей командой, чтобы убедиться в наличии времени на ежемесячную оплату за каждый спринт. Несложная задача — это снижение баланса технической задолженности при разработке функций. К счастью, существуют некоторые простые методы для [создания производительных, направленных на клиентов групп](https://docs.microsoft.com/ru-ru/devops/plan/building-productive-teams).

Всегда будьте гибкими

Гибкость означает обучение на опыте и постоянно улучшаются. Гибкая разработка предоставляет больше циклов обучения по сравнению с традиционным планированием проектов из-за более тесных циклов процессов. Каждый спринт предоставляет команде новые знания.

Например:

Группа предоставляет клиенту ценность, получает отзыв, а затем изменяет их невыполненную работу на основе этих отзывов.

Они узнают, что в своих автоматизированных сборках отсутствуют ключевые тесты, и они включают работу в следующем спринте для решения этой задачи.

Они обнаружили, что некоторые функции плохо работают в рабочей среде и планируют повышение производительности.

Кто-то из участников группы слышит новый подход, и группа решила опробовать его в течение нескольких спринтов.

Teams, только начиная с гибкой разработки, должны заинтересовать больше возможностей для обучения. Они являются полезной частью процесса, так как они приводят к росту и улучшению.

**Техническое задание**

1.1. Техническое задание оформляют в соответствии с [ГОСТ 19.106-78](https://docs.cntd.ru/document/1200007647#7D20K3) на листах формата 11 и 12 по [ГОСТ 2.301-68](https://docs.cntd.ru/document/1200006582#7D20K3), как правило, без заполнения полей листа. Номера листов (страниц) проставляют в верхней части листа над текстом.

1.2. Лист утверждения и титульный лист оформляют в соответствии с [ГОСТ 19.104-78](https://docs.cntd.ru/document/1200007645#7D20K3).

Информационную часть (аннотацию и содержание), лист регистрации изменений допускается в документ не включать.

1.3. Для внесения изменений или дополнений в техническое задание на последующих стадиях разработки программы или программного изделия выпускают дополнение к нему. Согласование и утверждение дополнения к техническому заданию проводят в том же порядке, который установлен для технического задания.

1.4. Техническое задание должно содержать следующие разделы:

введение;

основания для разработки;

назначение разработки;

требования к программе или программному изделию;

требования к программной документации;

технико-экономические показатели;

стадии и этапы разработки;

порядок контроля и приемки;

в техническое задание допускается включать приложения.

В зависимости от особенностей программы или программного изделия допускается уточнять содержание разделов, вводить новые разделы или объединять отдельные из них.

**Проектирование интерфейса пользователя**

Меня часто стали спрашивать как организована работа над интерфейсом у нас в студии. Мне стало лень каждый раз описывать этот процесс, поэтому я начал готовить презентацию с описанием этого процесса, которую можно будет отправлять клиентам. Часть презентации легла в основу этой статьи.

Под интерфейсом понимается любой экранный информационный или интерактивный интерфейс. Таковыми являются:

* сайты,
* мобильные приложения,
* приложения для стационарных компьютеров,
* презентационные панели (включая touch),
* информационные стационарные экраны.

Проецируемая картинка на стену или полотно с использованием проектора и управляемая жестами или голосом тоже считается интерфейсом.

# Этапы разработки

Полный цикл разработки интерфейса включает следующие этапы:

1. Исследование
2. Пользовательские сценарии
3. Структура интерфейса
4. Прототипирование интерфейса
5. Определение стилистики
6. Дизайн концепция
7. Оформление всех экранов
8. Анимация интерфейса
9. Подготовка материалов для разработчиков



Для сокращения общего времени разработки, определение стилистики начинается после пользовательских сценариев.

**системы контроля версий**

Контроль версий, также известный как управление исходным кодом, — это практика отслеживания изменений программного кода и управления ими. Системы контроля версий — это программные инструменты, помогающие командам разработчиков управлять изменениями в исходном коде с течением времени. В свете усложнения сред разработки они помогают командам разработчиков работать быстрее и эффективнее. Системы контроля версий наиболее полезны командам [DevOps](https://www.atlassian.com/ru/devops/what-is-devops), поскольку помогают сократить время разработки и увеличить количество успешных развертываний.

Программное обеспечение контроля версий отслеживает все вносимые в код изменения в специальной базе данных. При обнаружении ошибки разработчики могут вернуться назад и выполнить сравнение с более ранними версиями кода для исправления ошибок, сводя к минимуму проблемы для всех участников команды.

**[](https://www.youtube.com/watch?v=xQujH0ElTUg)**

**Настройка репозитория в Visual Studio**

**Создаем репозиторий**

1. Открываем шаблон
2. Вводим название репозитория
3. Выбираем необходимые параметры
4. Создаем репозиторий

**Клонируем репозиторий в среде разработки**

1. Указываем путь
2. Клонируем репозиторий

**Работа в master ветке**

**Работаем с кодом**

1. Создаем новый проект в репозитории
2. Вносим изменения в код

***Добавляем изменения в репозиторий***

1. Указываем название commit'a
2. Делаем commit

3. Делаем push изменений

**Важно**

Commit является основным объектом в системе управления версиями и содержит информацию о внесенных изменениях

***Создаем новую ветку***

1. Переходим к созданию ветки
2. Указываем ее имя
3. Создаем ветку

4. Делаем push новой ветки

**Важно**

Ветка в Git является указателем на один из commit'ов, которым чаще всего является последний commit

**Изменяем код в master ветке**

1. Добавляем код метода для вывода списка отелей
2. Создаем новый commit и делаем push изменений

**Важно**

В новой ветке код данного метода будет **отсутствовать**, т.к. ветка была создана перед реализацией метода

**Работа в новой ветке**

**Работаем с кодом**

1. Реализуем верстку названий отелей
2. Делаем commit

**Как откатить изменения?**

1. Открываем историю commit'ов

2. Откатываем изменения

**Важно**

После отката изменений новая ветка вернется к предыдущему состоянию

**Как слить ветки?**

1. Выбираем ветку для слияния
2. Сливаем ветки

**Проектирование архитектуры ИС**

Архитектура информационной системы (ИС) – ее концепция, которая определяет модель, структуру, функционал и взаимосвязь компонентов.  
  
Имея в штате системного архитектора, компания Mobidom может спроектировать архитектуру информационной системы любого назначения – как для решения бизнес-задач, так и для применения в решении государственных задач заказчиками федерального и регионального уровней.

радиционными архитектурами ИС являются:

* Файл-серверные системы (файловый сервер), в составе которых на стороне сервера осуществляется хранение информации и программного кода, а на стороне клиента и только здесь происходит обработка данных. Основная проблема систем этого типа – разделение мест хранения и обработки информации, в следствие чего серьезно снижаются надежность и производительность системы.
* Двухслойные (двухуровневые) клиент-серверные системы (сервер СУБД) – надежные, многопользовательские, имеющие централизованную БД, оперирующие данными на уровне логической схемы. На стороне клиента – интерфейс и бизнес-логика, на стороне сервера – хранение и управление файлами данных, выполнение запросов и обработка, хранение процедур. Ключевая проблема систем с такой архитектурой – исполнение бизнес-логики на стороне клиента, что порождает необходимость при изменении алгоритмов в работе системы обновлять пользовательское программное обеспечение на каждом клиенте.
* Промежуточная клиент-серверная архитектура, основное отличие которой от классической клиент-серверной модели заключается в разделении бизнес-логики, часть которой остается на стороне клиента, а часть – выполняется на стороне сервера.
* Трехуровневая (трехслойная) клиент-серверная архитектура (сервер приложений) – отличаются разделением пользовательского интерфейса (уровень клиента), бизнес-логики (сервер приложений) и управления данными (сервер БД). Основное преимущество этого типа архитектуры – оптимизация нагрузки на все компоненты системы, что дает повышение производительности и надежности в работе ИС. Ключевой и, пожалуй, единственно важный недостаток такой архитектуры – повышение расходов на обслуживание и администрирование серверной части.

Новые типы архитектур, хотя и выделяются как более современные, фактически являются развитием архитектуры трехуровневого клиент-серверного типа. Они также базируются на трех компонентах: клиент, веб-сервер и сервер СУБД. В настоящее время среди них наибольшее распространение получили:

* Архитектуры на базе Интернет/Интранет-технологий:
  + клиентом выступает браузер, который разделяет уровень представления информации с веб-сервером;
  + на стороне веб-сервера находится бизнес-логика (CGI, API), частично разделенная со следующим уровнем – сервером СУБД (процедуры);
  + на стороне сервера СУБД – хранение и управление БД, выполнение запросов и обработка.
* Архитектуры на базе Интернет/Интранет-технологий с мигрирующим ПО:
  + клиентом выступает браузер, который разделяет уровень представления информации с веб-сервером;
  + на стороне клиента находится бизнес-логика (приложения и фреймворк) и обеспечивается прямой доступ и соединение с БД (с сервером СУБД);
  + на стороне сервера СУБД – хранение и управление БД, выполнение запросов и обработка.

Разработка архитектуры ИС

В рамках разработки архитектуры определяются:

* что будет делать система;
* из каких компонентов (частей, модулей) она будет состоять;
* где именно компоненты будут располагаться;
* каким образом компоненты будут взаимодействовать.

Для архитектуры определяются и описываются:

* Базовые параметры и характеристики архитектуры.
* Логическая и физическая структура.
* Взаимодействие системных компонентов (подсистемы и модули, синхронность и асинхронность их взаимодействия, каналы коммуникации и их характеристики, протоколы и интерфейсы, тип программного обеспечения промежуточного слоя, форматы файлов, которыми система будет оперировать, и другие особенности).
* Необходимые элементы ИТ-инфраструктуры для реализации выстраиваемой архитектуры ИС – платформа (среда), аппаратный комплекс, СУБД, инструментарий прикладное ПО.
* Возможные риски, ограничения, стоимость владения, экономическая обоснованность.

**Паттерны проектирования.**

### **Условные обозначения**

#### **Отношения между классами**

*  — агрегация (aggregation) — описывает связь «часть»–«целое», в котором «часть» может существовать отдельно от «целого». Ромб указывается со стороны «целого».
*  — композиция (composition) — подвид агрегации, в которой «части» не могут существовать отдельно от «целого».
*  — зависимость (dependency) — изменение в одной сущности (независимой) может влиять на состояние или поведение другой сущности (зависимой). Со стороны стрелки указывается независимая сущность.
*  — обобщение (generalization) — отношение наследования или реализации интерфейса. Со стороны стрелки находится суперкласс или интерфейс.

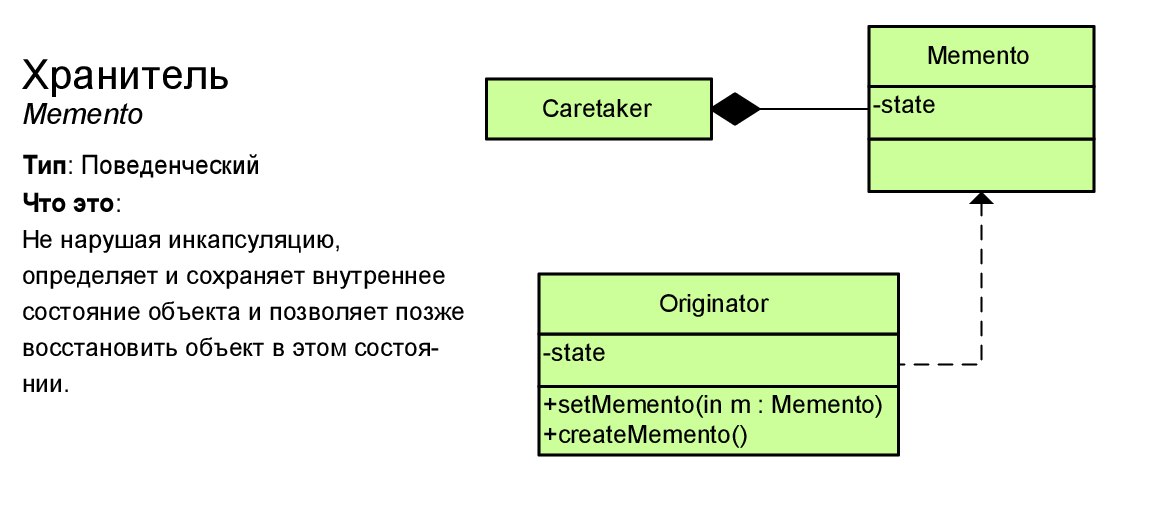
#### **Виды паттернов**

*  — поведенческие (behavioral);
*  — порождающие (creational);
*  — структурные (structural).

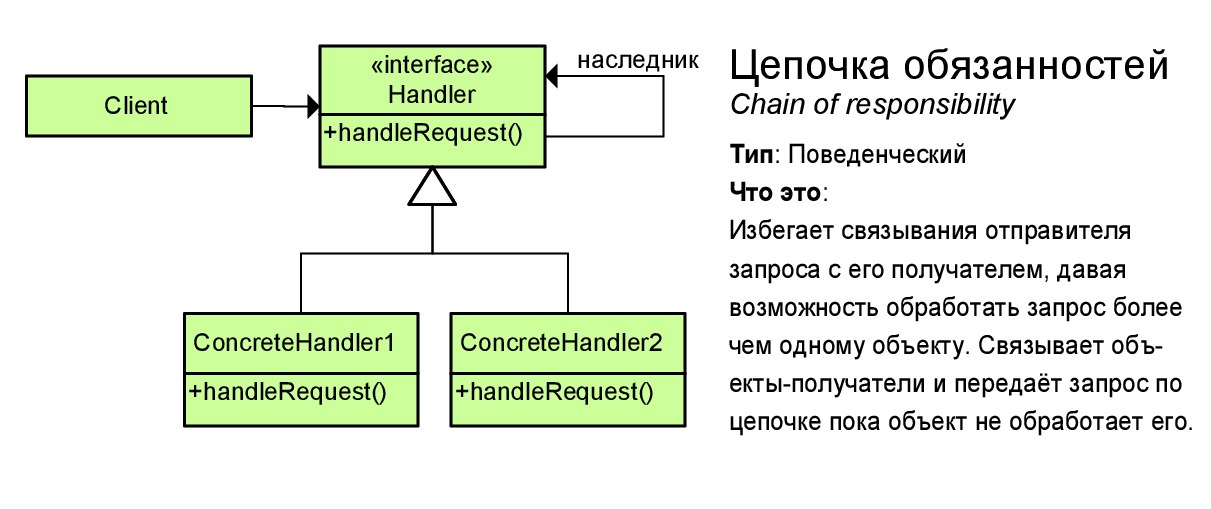
###### **Список шаблонов**



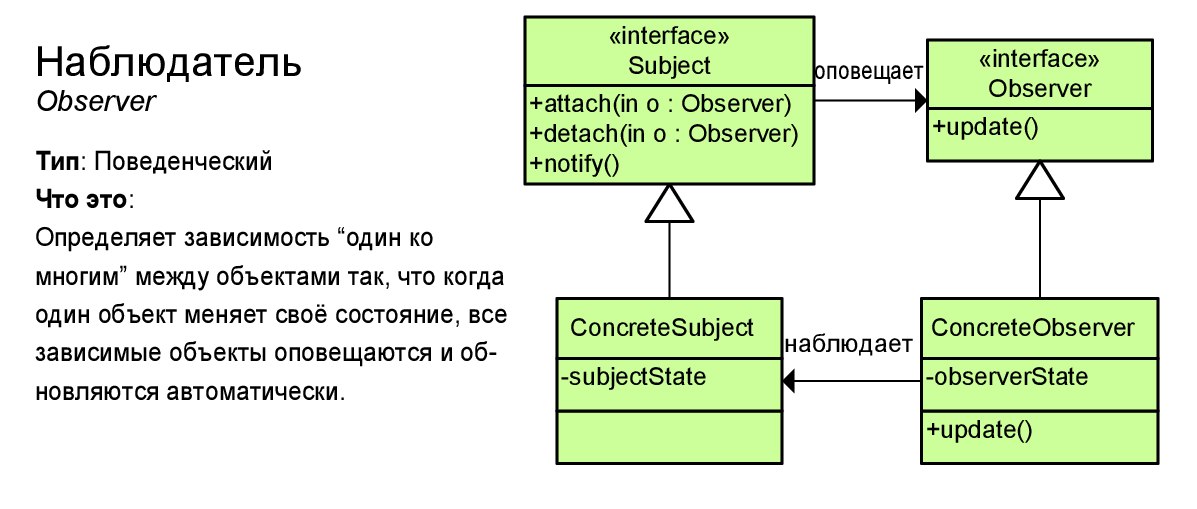
###### **Хранитель (memento)**



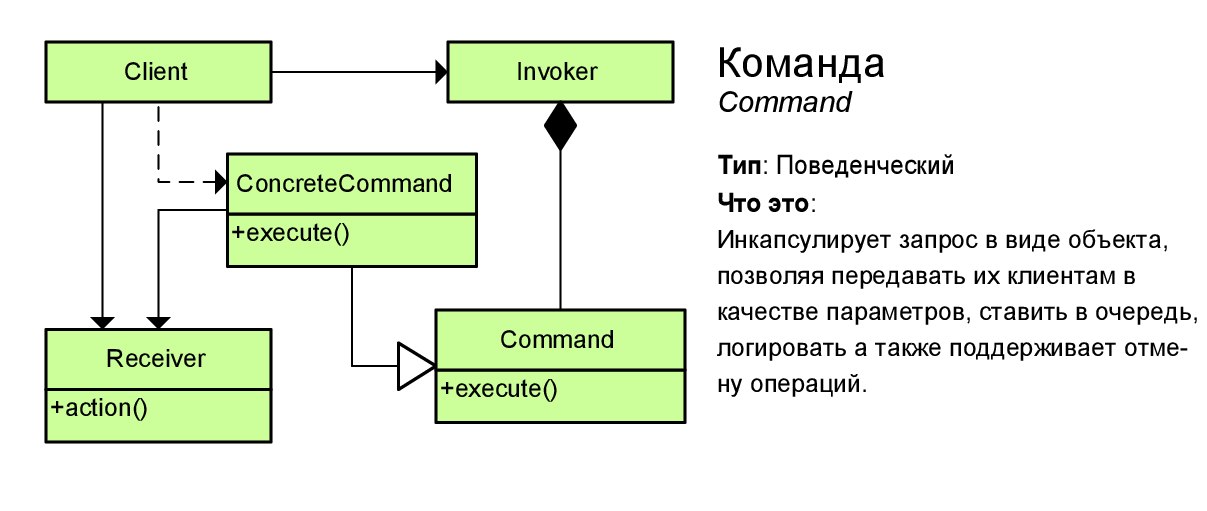
###### **Цепочка обязанностей (chain of responsibility)**



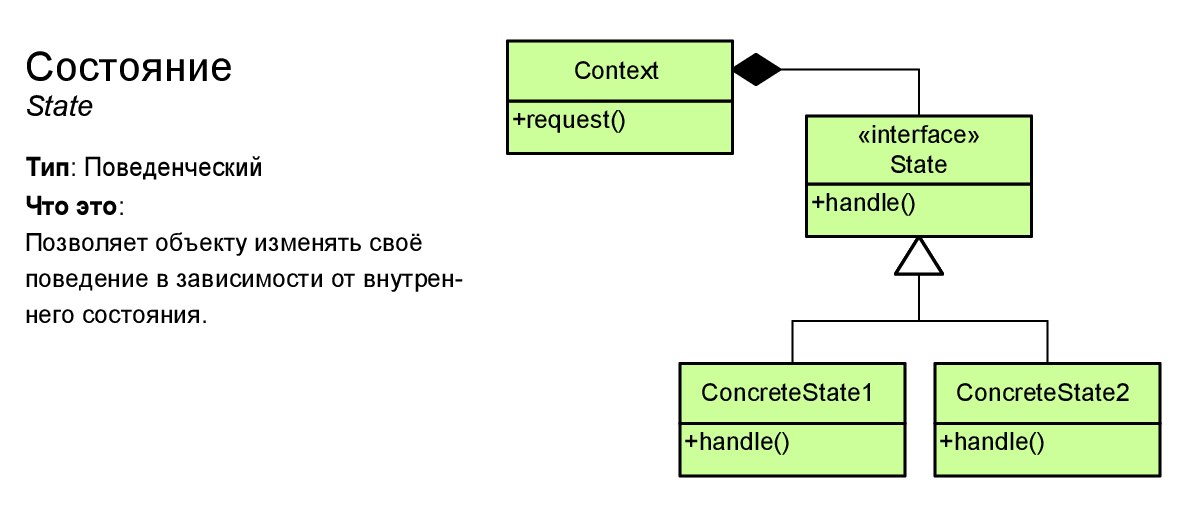
###### **Наблюдатель (observer)**



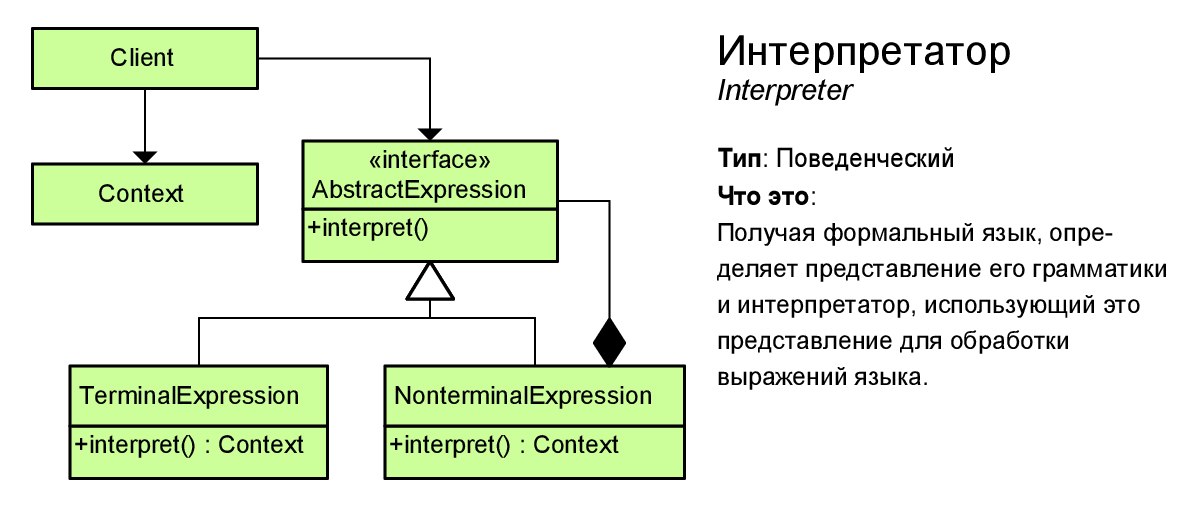
###### **Команда (command)**



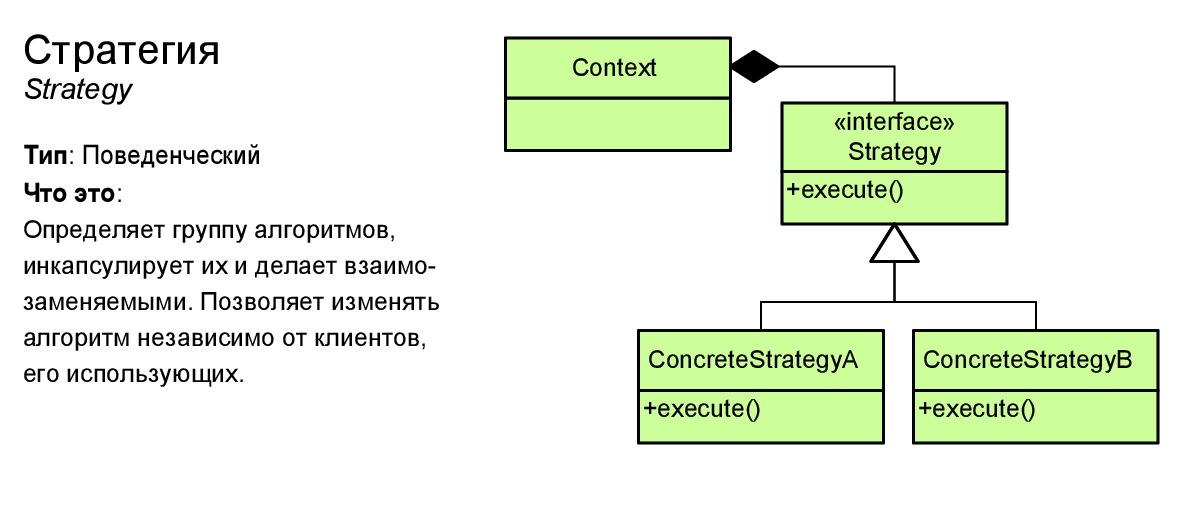
###### **Состояние (state)**



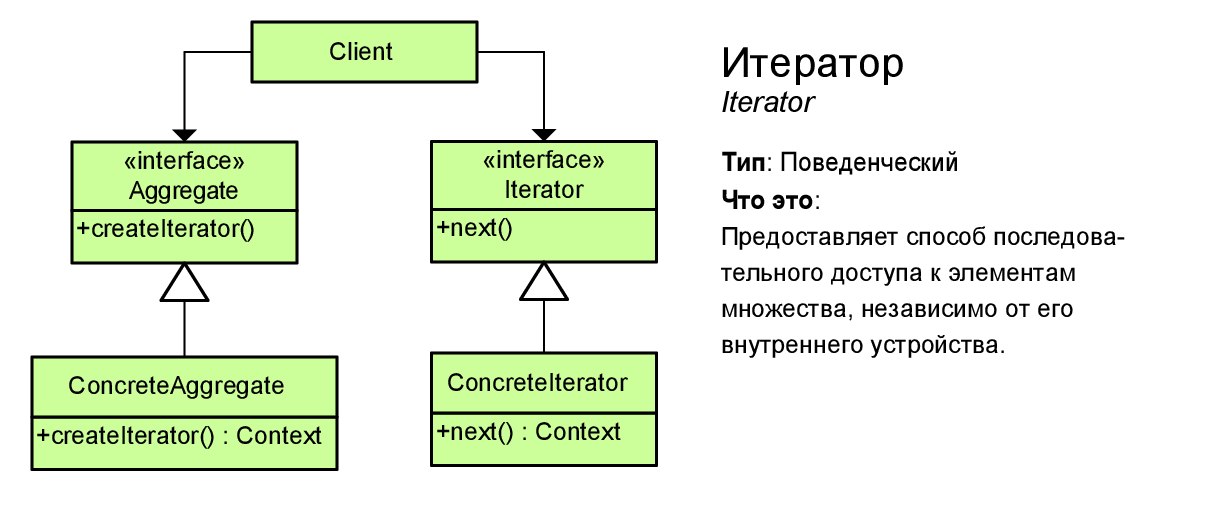
###### **Интерпретатор (interpreter)**



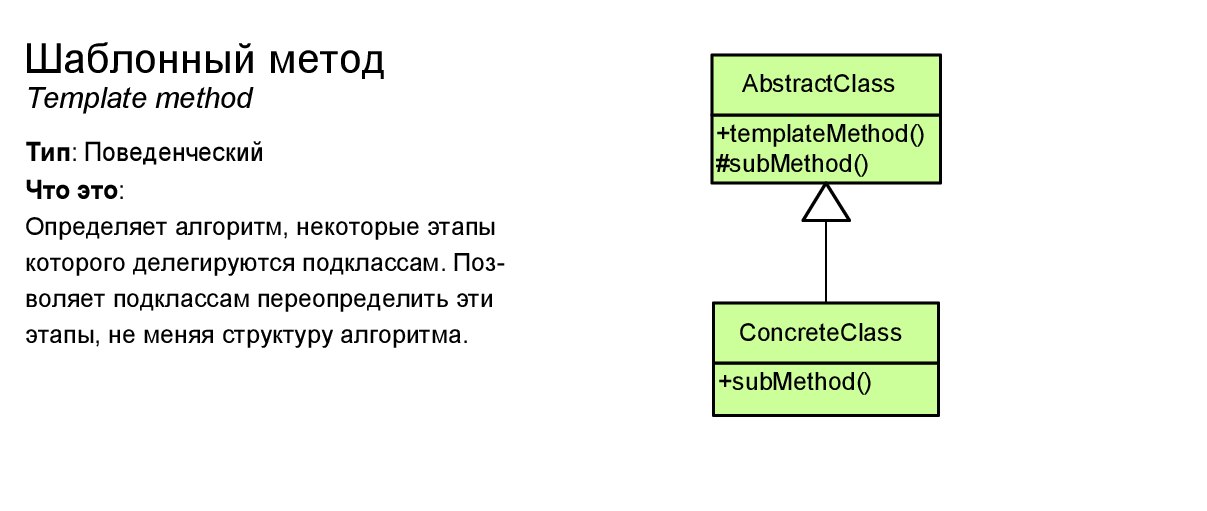
###### **Стратегия (strategy)**



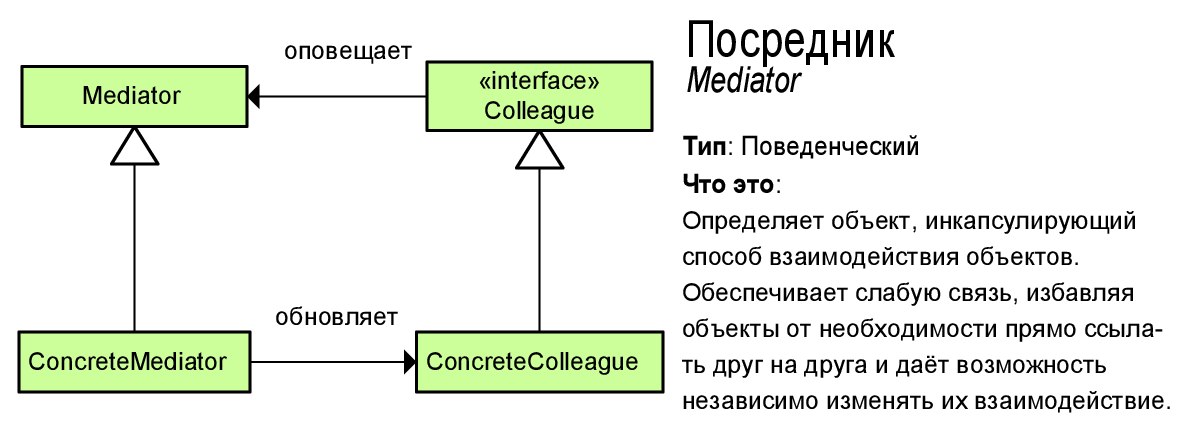
###### **Итератор (iterator)**



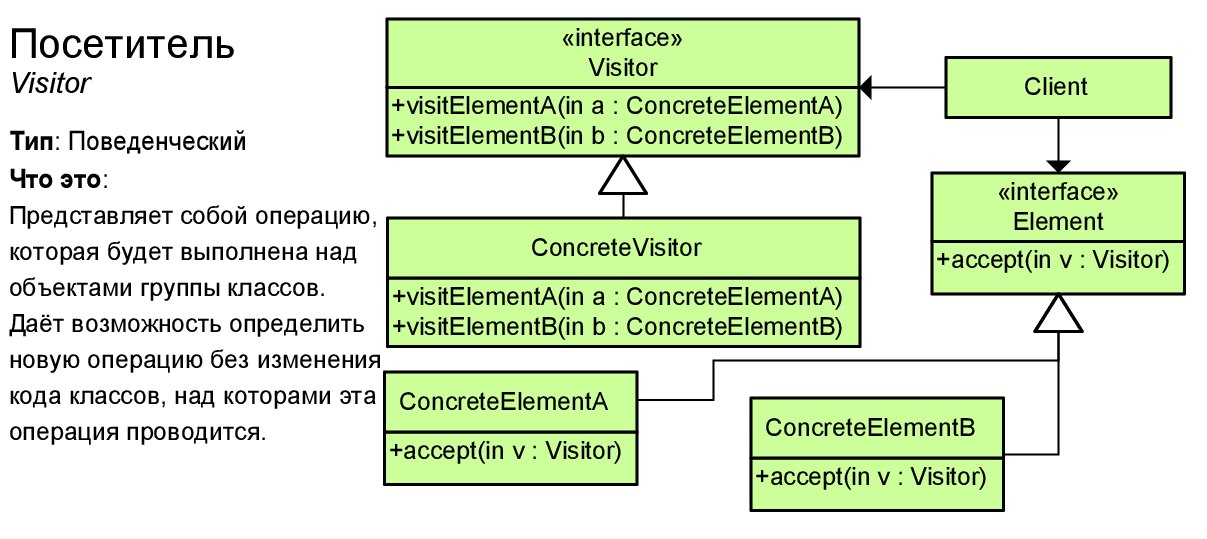
###### **Шаблонный метод (template method)**



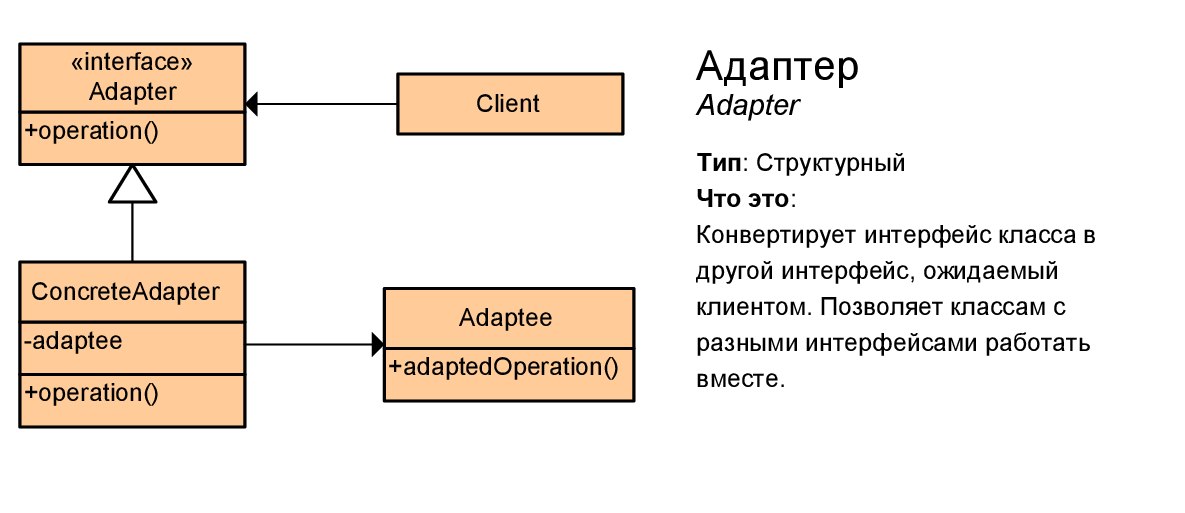
###### **Посредник (mediator)**



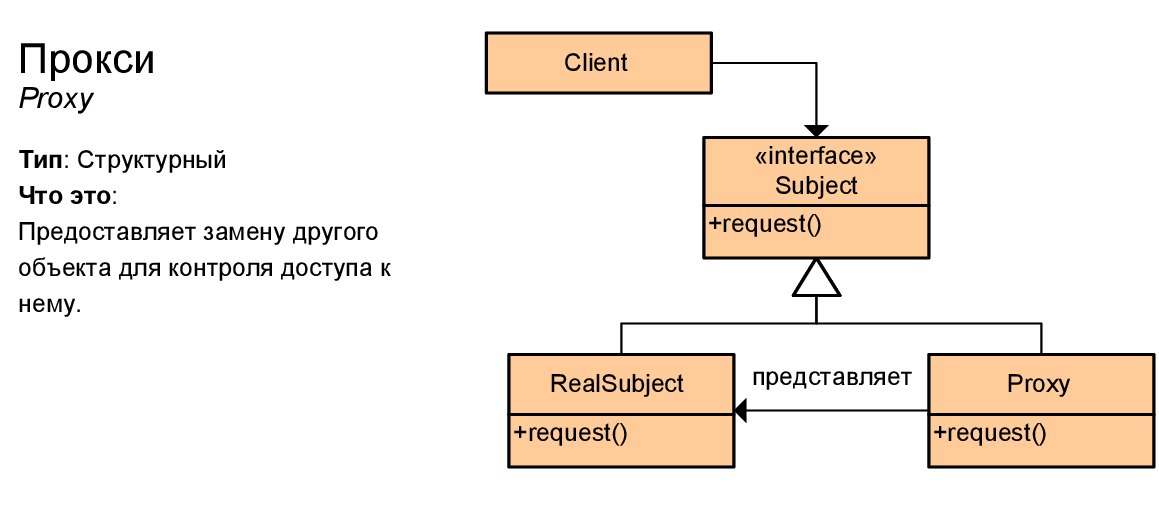
###### **Посетитель (visitor)**



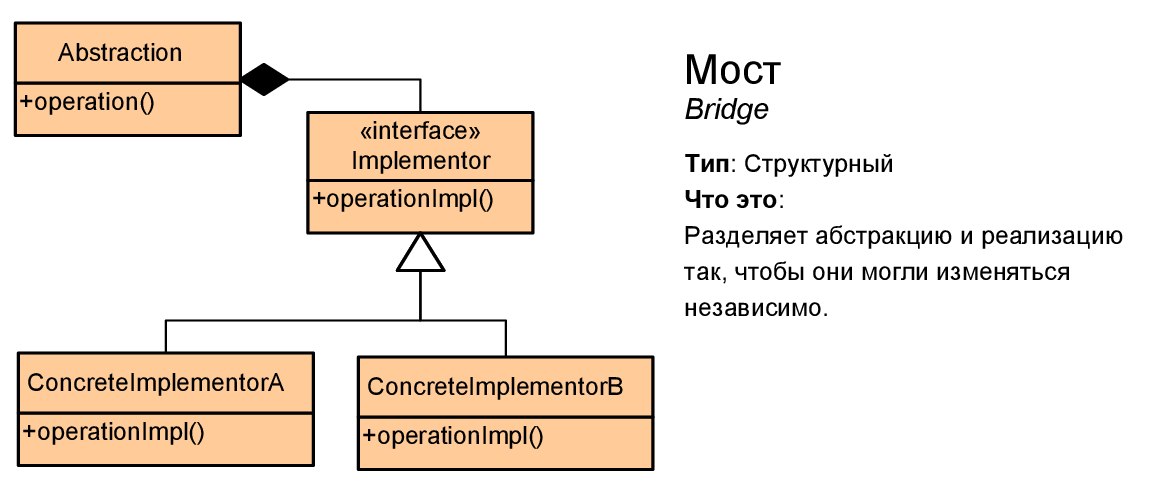
###### **Адаптер (adapter)**



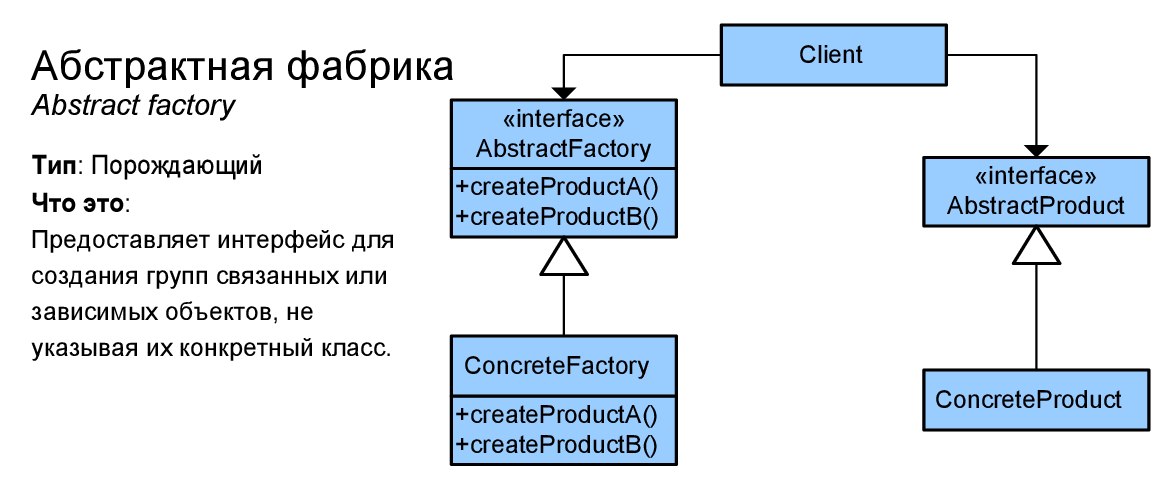
###### **Прокси (proxy)**



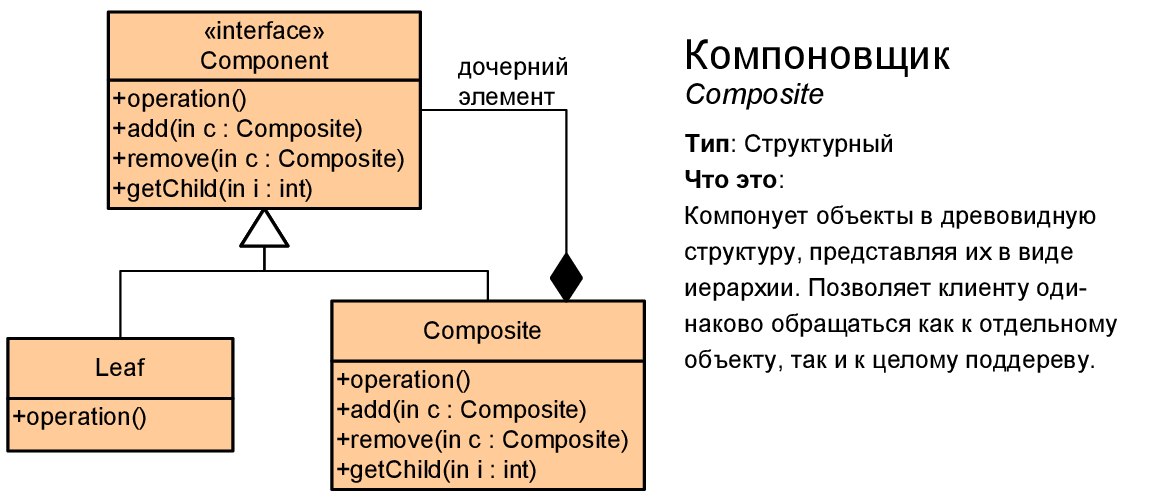
###### **Мост (bridge)**



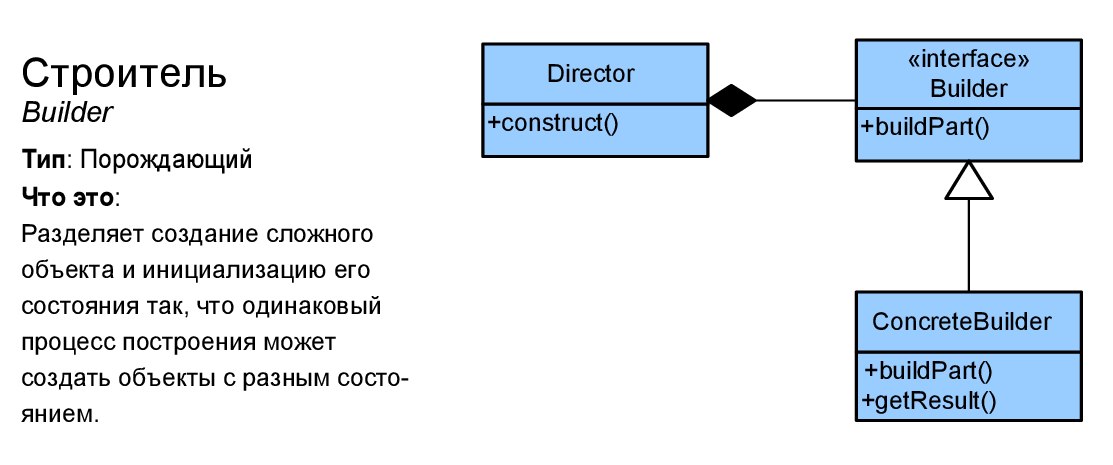
###### **Абстрактная фабрика (abstract factory)**



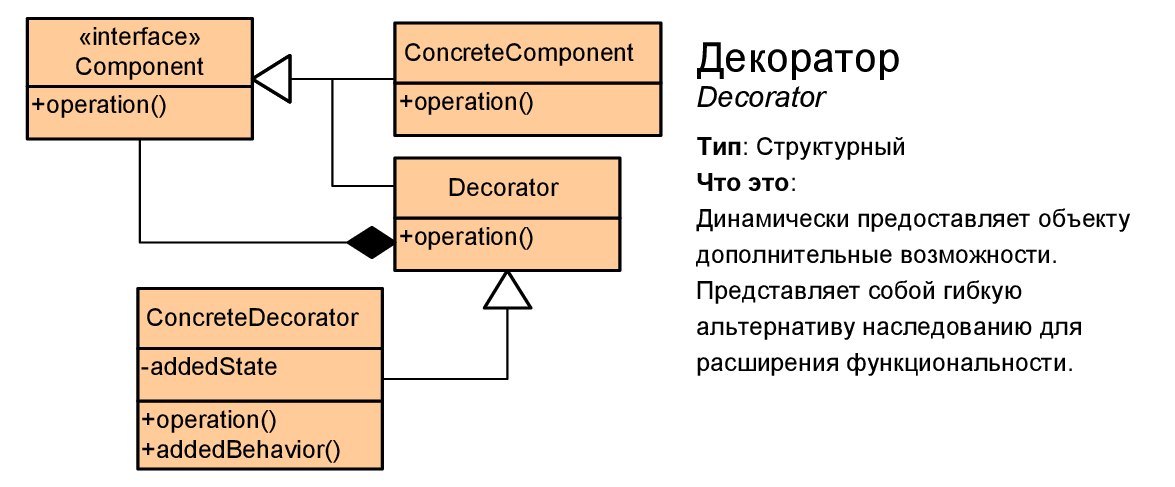
###### **Компоновщик (composite)**



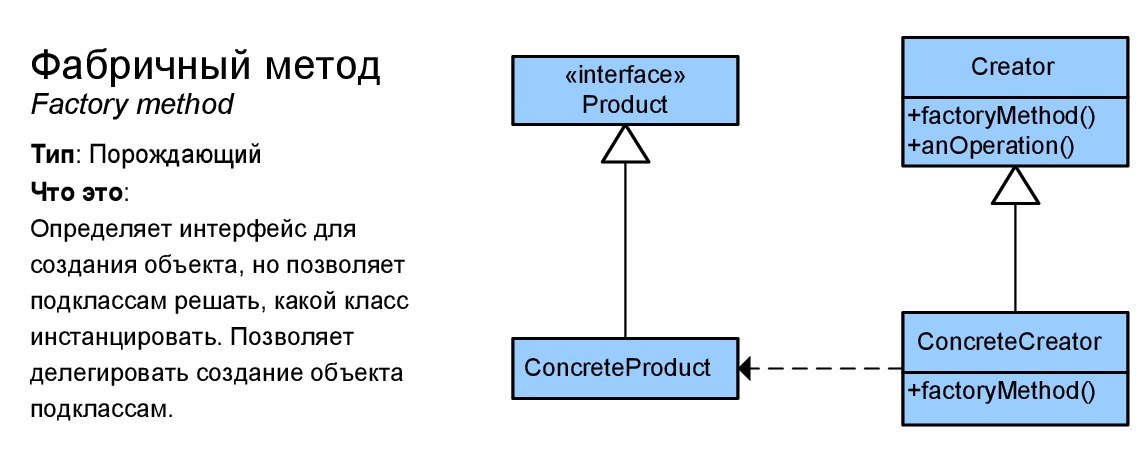
###### **Строитель (builder)**



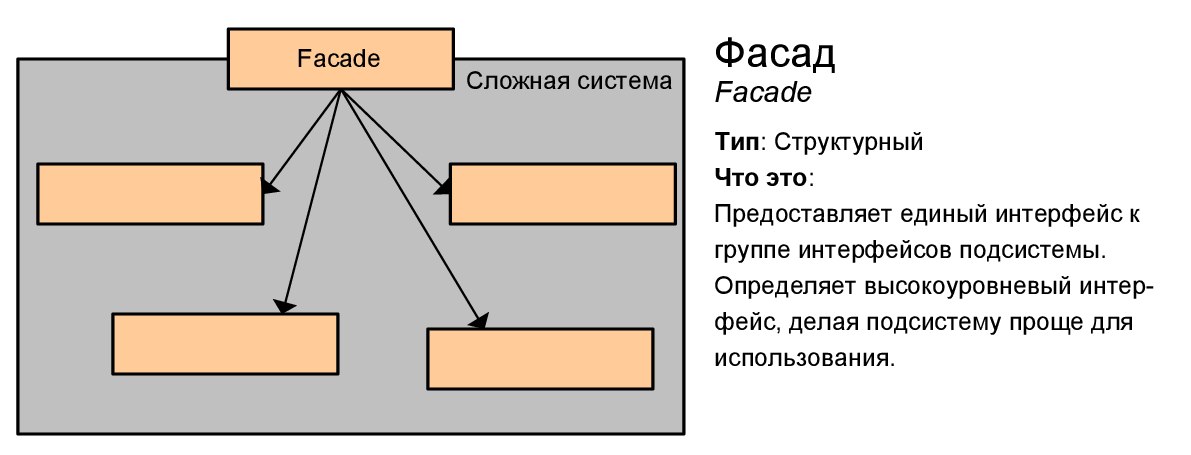
###### **Декоратор (decorator)**



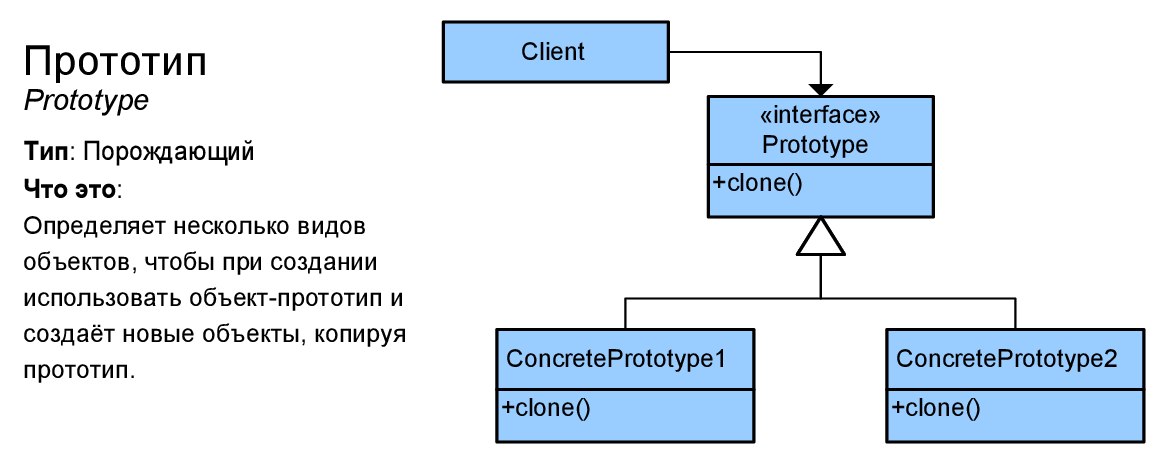
###### **Фабричный метод (factory method)**



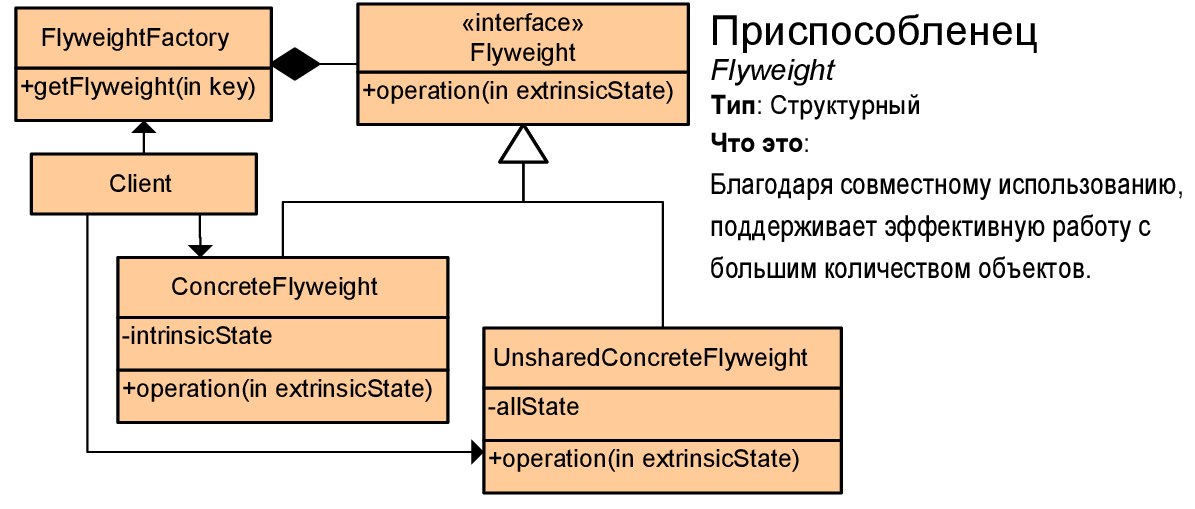
###### **Фасад (facade)**



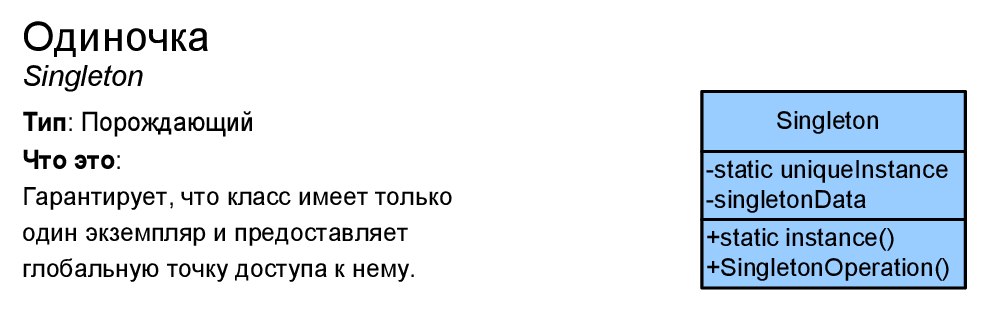
###### **Прототип (prototype)**



###### **Приспособленец (flyweight)**



###### **Одиночка (singleton)**



**Структурные методы анализа и проектирования ПО: IDEF0, DFD, IDEF1x**

DFD — общепринятое сокращение от англ. data flow diagrams — диаграммы потоков данных. Так называется методология графического структурного анализа, описывающая внешние по отношению к системе источники и адресаты данных, логические функции, потоки данных и хранилища данных, к которым осуществляется доступ. Диаграмма потоков данных (data flow diagram, DFD) — один из основных инструментов структурного анализа и проектирования информационных систем, существовавших до широкого распространения UML. Википедия

По моему мнению, определение из русскоязычной Википедии, несколько перегружено информацией и, в результате, излишне сложно для понимания. Кроме того, лично я считаю, что DFD и UML — это разные инструменты, а потому некорректно утверждать, что DFD — это просто предшественник UML.  
  
Для себя я вывел следующую формулировку:

DFD – это нотация, предназначенная для моделирования информационный систем с точки зрения хранения, обработки и передачи данных.

IDEF0 - методология функционального моделирования. С помощью наглядного графического языка IDEF0, изучаемая система предстает перед разработчиками и аналитиками в виде набора взаимосвязанных функций (функциональных блоков - в терминах IDEF0). Как правило, моделирование средствами IDEF0 является первым этапом изучения любой системы;

IDEF1X (IDEF1 Extended) – методология построения реляционных структур. IDEF1X относится к типу методологий “Сущность-взаимосвязь” (ER – Entity-Relationship) и, как правило, используется для моделирования реляционных баз данных, имеющих отношение к рассматриваемой системе;

**Что такое UML-диаграммы?**

Unified Modeling Language (UML) — унифицированный язык моделирования. Расшифруем: *modeling* подразумевает создание модели, описывающей объект. *Unified* (универсальный, единый) — подходит для широкого класса проектируемых программных систем, различных областей приложений, типов организаций, уровней компетентности, размеров проектов. UML описывает объект в едином заданном синтаксисе, поэтому где бы вы не нарисовали диаграмму, ее правила будут понятны для всех, кто знаком с этим графическим языком — даже в другой стране.

**Для чего используется UML?**

Одна из задач UML — служить средством коммуникации внутри команды и при общении с заказчиком. Давайте рассмотрим возможные варианты использования диаграмм.

* Проектирование. UML-диаграммы помогут при [моделировании архитектуры больших проектов](https://evergreens.com.ua/ru/development-services/software-architecture-design.html), в которой можно собрать как крупные, так и более мелкие детали и нарисовать каркас (схему) приложения. По нему впоследствии будет строиться код.
* Реверс-инжиниринг — создание UML-модели из существующего кода приложения, обратное построение. Может применяться, например, на проектах поддержки, где есть написанный код, но [документация](https://evergreens.com.ua/ru/development-services/srs-development.html) неполная или отсутствует.
* Из моделей можно извлекать текстовую информацию и генерировать относительно удобочитаемые тексты — документировать. Текст и графика будут дополнять друг друга.

**Нотация UML для описания логики проекта**

Как и любой другой язык, UML имеет собственные правила оформления моделей и синтаксис. С помощью графической нотации UML можно визуализировать систему, объединить все компоненты в единую структуру, уточнять и улучшать модель в процессе работы. На общем уровне графическая нотация UML содержит 4 основных типа элементов:

* фигуры;
* линии;
* значки;
* надписи.

UML-нотация является де-факто отраслевым стандартом в области разработки программного обеспечения, ИТ-инфраструктуры и бизнес-систем.

**Виды UML-диаграмм**

В языке UML есть 12 типов диаграмм:

* 4 типа диаграмм представляют **статическую структуру** приложения;
* 5 типов представляют **поведенческие аспекты** системы;
* 3 представляют**физические аспекты** функционирования системы (диаграммы реализации).

Некоторые из видов диаграмм специфичны для определенной системы и приложения. Самыми доступными из них являются:

* Диаграмма прецедентов (Use-case diagram);
* Диаграмма классов (Class diagram);
* Диаграмма активностей (Activity diagram);
* Диаграмма последовательности (Sequence diagram);
* Диаграмма развёртывания (Deployment diagram);
* Диаграмма сотрудничества (Collaboration diagram);
* Диаграмма объектов (Object diagram);
* Диаграмма состояний (Statechart diagram).

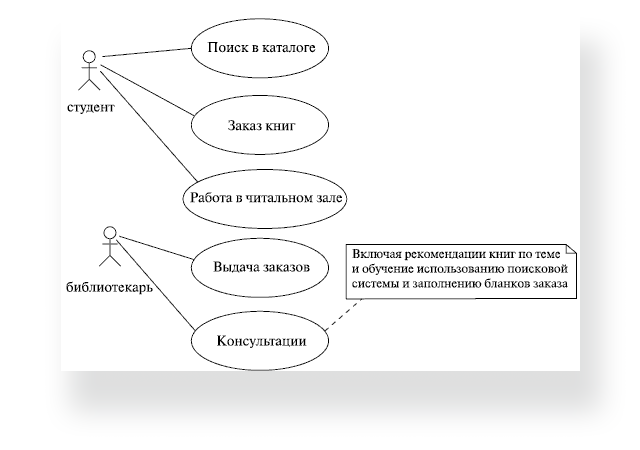
## Диаграмма прецедентов — Use-case diagram

Диаграмма прецедентов использует 2 основных элемента:

1) Actor (участник) — множество логически связанных ролей, исполняемых при взаимодействии с прецедентами или сущностями (система, подсистема или класс). Участником может быть человек, роль человека в системе или другая система, подсистема или класс, которые представляют нечто вне сущности.

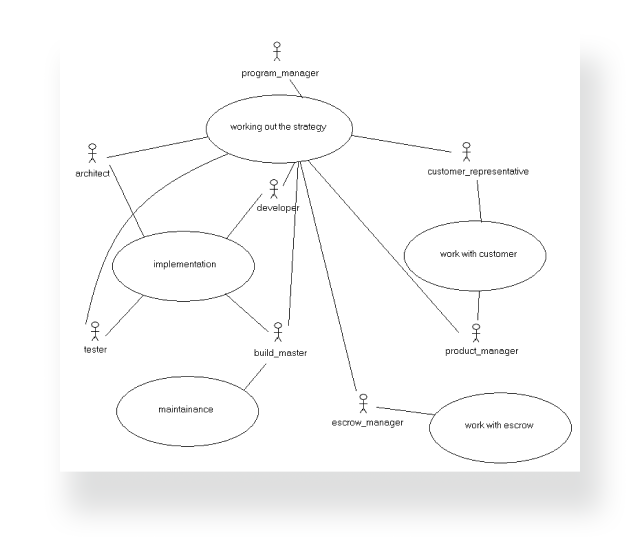
2) Use case (прецедент) — описание отдельного аспекта поведения системы с точки зрения пользователя. Прецедент не показывает, "как" достигается некоторый результат, а только "что" именно выполняется.

Рассмотрим классический студенческий пример, в котором есть 2 участника: студент и библиотекарь. Прецеденты для студента: ищет в каталоге, заказывает, работает в читальном зале. Роль библиотекаря: выдача заказа, консультации (рекомендации книг по теме, обучение использованию поисковой системы и заполнению бланков заказа).



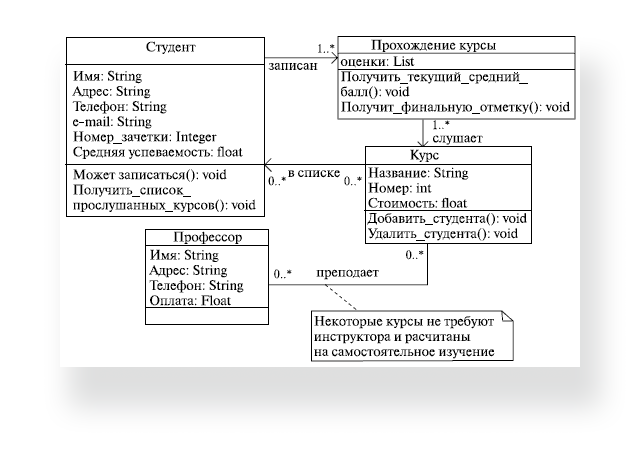
Источник: slide-share.ru

Второй пример немного сложнее. Видим, что одно и то же лицо может выступать в нескольких ролях. Например, product manager у нас работает над стратегией и больше ничем не занимается, архитектор работает над стратегией и занимается внедрением, build master занимается тремя вещами одновременно, и так далее. По такой схеме мы можем проследить, какая из ролей связана с какими прецедентами.



Источник: slide-share.ru

## Диаграмма классов — Class diagram

Класс (class) — категория вещей, которые имеют общие атрибуты и операции. Сама диаграмма классов являет собой набор статических, декларативных элементов модели. Она дает нам наиболее полное и развернутое представление о связях в программном коде, функциональности и информации об отдельных классах. Приложения генерируются зачастую именно с диаграммы классов. Рассмотрим на простом примере ниже: 

Источник: slide-share.ru

Для класса "студент" есть таблица, содержащая атрибуты: имя, адрес, телефон, e-mail, номер зачетки, средняя успеваемость. И также показаны связи данной сущности с другими: прохождением курса, какой курс слушает, кто профессор. В этом примере также добавляются функции, которые могут быть применены к сущности "студент".

## Диаграмма активностей — Activity diagram

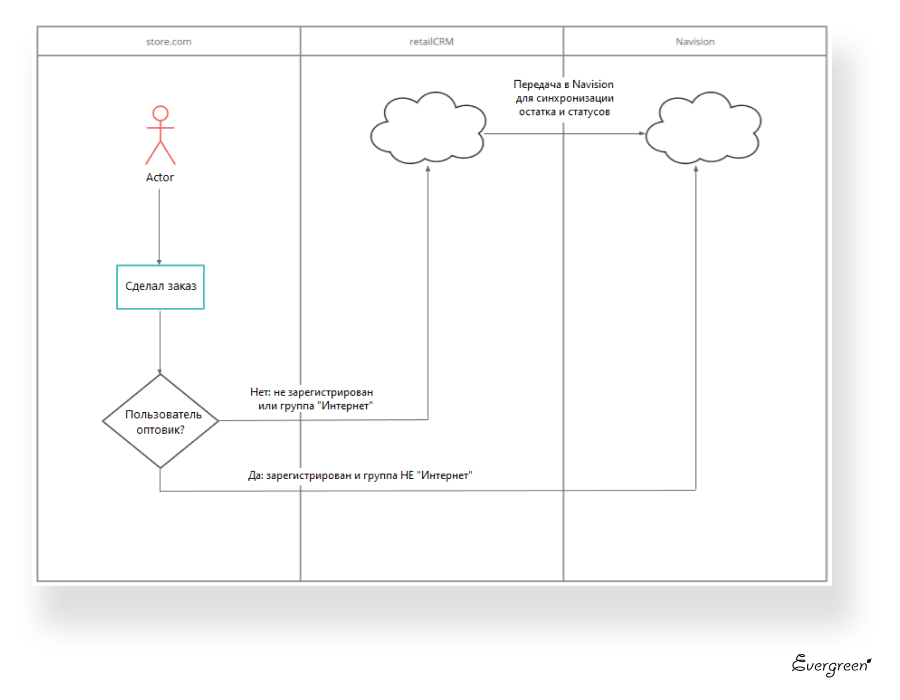
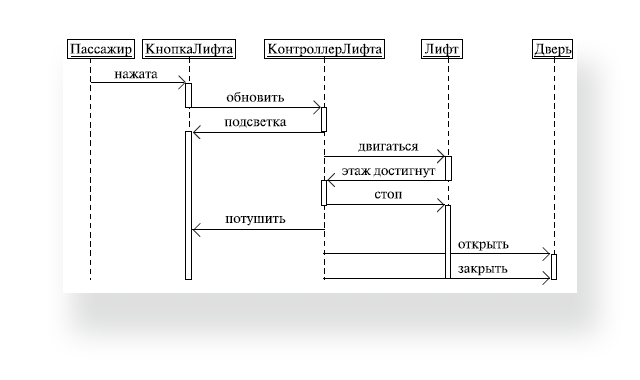
Тоже крутая штука, которая очень часто используется на практике. Диаграмма активностей описывает динамические аспекты поведения системы в виде блок-схемы, которая отражает бизнес-процессы, логику процедур и потоки работ — переходы от одной деятельности к другой. По сути, мы рисуем алгоритм действий (логику поведения) системы или взаимодействия нескольких систем. Ниже — пример подобной диаграммы для интернет-магазина.

Диаграмма активностей для сайта магазина максимально доступно объясняет, какие есть интеграции в системе. Актер (в нашем случае — покупатель), зашедший на сайт, делает заказ. Далее у нас происходит разветвление: проверяем, является ли пользователь оптовиком (Да/Нет). Если он не зарегистрирован в системе и не оптовик, заказ отправляется в retailCRM. Если пользователь зарегистрирован, его заказ попадает в Navision. При этом между retailCRM и Navision происходит синхронизация остатка и статусов.

Эту базовую диаграмму мы можем дополнить, расширить, она может выступить частью документации и дает общее представление о работе системы.

## Диаграмма последовательности — Sequence Diagram

Используется для уточнения диаграмм прецедентов — описывает поведенческие аспекты системы. Диаграмма последовательности отражает взаимодействие объектов в динамике, во времени. При этом информация принимает вид сообщений, а взаимодействие объектов подразумевает обмен этими сообщениями в рамках сценария.



 Источник: slide-share.ru

## Диаграмма развертывания — Deployment Diagram

Диаграмма развертывания отображает графическое представление инфраструктуры, на которую будет развернуто приложение: топологию системы и распределение компонентов по ее узлам, а также соединения — маршруты передачи данных между узлами. Диаграмма помогает более рационально организовать компоненты, от чего зависит в числе прочего и производительность системы, а также решить вспомогательные задачи, например, связанные с безопасностью.

