Технология программирования

Основы алгоритмических языков программирования. Классификация языков программирования

Вопрос 1. Основы алгоритмических языков. Понятие алгоритма. Признаки алгоритма. Виды записи алгоритмов. Виды языков программирования

Алгоритмизация и программирование составляют основополагающие элементы теоретической информатики. Рассмотрим на примере термина «алгоритмический язык программирования» качества и способы записи алгоритмов.

Алгоритмический язык программирования — это формальный язык, предназначенный для описания, реализации и анализа алгоритмов.

В отличие от многих других языков программирования, этот язык не привязан к определенной архитектуре компьютера и не учитывает особенности компьютерного оборудования. Это специально разработанный язык, который содержит ограниченный набор ключевых слов и фраз. Его объем значительно меньше, чем у естественного языка. Он отличается лаконичностью, отсутствием синонимов и двусмысленности, а все выражения четко определяют конкретные действия. В целом, это «человеческий» язык, так как он использует слова и фразы, заимствованные из разговорной речи. Большинство распространенных алгоритмических языков базируется на английском языке, потому что они разрабатывались американскими компаниями. Основная цель алгоритмического языка — описывать алгоритмы вычислений. Такое описание называется программой.

Современные алгоритмические языки представляют собой универсальные высокоуровневые языки. Чем выше уровень языка, тем обобщеннее команды. Они записываются на алгоритмическом языке и представляют собой макрокоманды, в то время как компьютер выполняет операции на уровне микрокоманд, то есть элементарных действий. Использование макрокоманд при написании программы значительно уменьшает ее объем и облегчает восприятие. Программа на алгоритмическом языке понятна

человеку, но «не понятна» компьютеру. Компьютер обрабатывает информацию на языке машинных двоичных кодов. Поэтому, чтобы компьютер смог интерпретировать информацию, заложенную в программе, надо «перевести» ее на машинный язык. Для этой цели используются трансляторы.

Существуют разные типы таких программ, но наиболее популярными считаются компиляторы. Они выполняют преобразование алгоритмического языка в машинный и проводят синтаксический анализ кода. Этот анализ включает проверку корректности написанных в программе выражений и конструкций в соответствии с установленными языковыми стандартами. Если компилятор находит несоответствия, он выводит сообщения об ошибках на экран. До тех пор, пока все ошибки не устранят, программа не выполняется. Компилятор — это неотъемлемая частью языка программирования.

Чтобы программы работали без сбоев и всегда предоставляли верные результаты независимо от входных данных, нужно учесть характеристики алгоритма.

Основные свойства алгоритмов

1. Понятность — исполнитель алгоритма должен понимать и исполнять команды.
2. Дискретность (прерывность) — алгоритм должен представлять процесс решения задачи как последовательное выполнение простых команд.
3. Определенность — каждое правило алгоритма должно быть четким и определенным.
4. Результативность — алгоритм должен приводить к решению задачи за конечное число шагов.
5. Массовость — алгоритм решения задачи разрабатывается в общем виде, то есть он должен подходить для некоторого класса задач, которые различаются лишь исходными данными.

Примеры алгоритмов включают:

- сортировку документов;
- следование рецепту;
- сигналы светофора;
- расписание автобусов;
- распознавание лиц;
- завязывание шнурков на ботинках;
- классификация объектов;
- поиск книги в библиотеке;
- путешествие к чему-либо или из чего-либо.

Любой алгоритм записывают несколькими способами. Наиболее распространены следующие формы записи алгоритмов:

- словесная — записи на естественном языке;
- табличная, когда алгоритм представлен в виде таблицы, где каждая строка соответствует отдельному шагу или действию, а столбцы содержат информацию о параметрах этого шага или действия;
- блок-схемы графически изображают последовательность действий, используют стандартизированные символы, чтобы отобразить этапы процесса;
- псевдокод — описания алгоритмов на условном алгоритмическом языке, которые включают элементы языка программирования и фразы естественного языка, общепринятые математические обозначения;
- программная — тексты на языках программирования.

Виды языков программирования

Разберем, какие виды языков программирования есть сейчас и какие популярнее других.

Таблица 1. Виды языков программирования

Типы	Основные различия	Виды	Примеры
Низкоуровневые	Взаимодействуют непосредственно с аппаратным обеспечением, предоставляют ему четкие инструкции. Это все компиляторы, которые работают с.NET, трансформируют языки высокого уровня из этого фреймворка в язык CIL, а он позволяет управлять аппаратным функционалом устройства		Машинный код содержит только «1» и «0»
			Язык ассемблера создает новый синтаксис для управления процессорами в различных архитектурах
			Forth — с его помощью разрабатывали ядра и ОС
			CIL используется в качестве промежуточного языка в виртуальной машине.NET

<table>
<tr><td rowspan="4">Высокоуровневые</td><td rowspan="4">Используют более абстрактные концепции, позволяют программисту не вникать в тонкости работы каждого компонента системы, а сосредоточиться на выполнении общих задач и функций</td><td>Процедурные</td><td colspan="2">Fortran
Pascal
Perl</td></tr>
<tr><td rowspan="2">Декларативные</td><td>Функциональные</td><td>Lisp
Pascal (некоторые диалекты, например, ML и Haskell)
Ruby (некоторые аспекты)</td></tr>
<tr><td>Логические</td><td>Prolog</td></tr>
<tr><td>Объективно-ориентированные</td><td colspan="2">C++
Java
C#
Delphi
Python</td></tr>
</table>

Вопрос 2. Парадигмы программирования

Программисты придумали разные способы решения задач и собрали их в четкие правила, которые назвали парадигмами. На основе этих правил создаются языки программирования. У каждого языка есть свой набор таких правил, которыми пользуются при написании программ.

Парадигмы также можно воспринимать как ограничения на выполнение определенных действий в коде программы. Этот подход был предложен Робертом Мартином, американским инженером и программистом, консультантом и автором в области разработки программного обеспечения, более известным среди программистов как дядя Боб. По его мнению, парадигмы представляют собой ограничения на некоторые языковые конструкции, которые вынуждают программистов придерживаться определенного стиля. Например, процедурное программирование запрещает «скачки» по коду, а функциональное программирование — прямое изменение оперативной памяти.

Парадигмы программирования устанавливают принципы написания кода. Это значительно облегчает переход разработчика к другому языку, который

использует знакомую парадигму, и помогает преодолеть так называемый «языковой барьер».

Основные парадигмы

1. Императивная.
2. Декларативная.

Каждая из них включает разные подгруппы. Например, к императивной парадигме относятся процедурное и объектно-ориентированное программирование, далее — ООП, а декларативная парадигма охватывает функциональное и логическое программирование.

Императивная парадигма считается одной из самых простых и распространенных в программировании. Ее суть — в выполнении операций последовательно, шаг за шагом. Многие языки программирования придерживаются императивного подхода и существуют в разных вариантах.

Ранее большинство разработчиков использовали процедурные языки, но с течением времени эта форма программирования стала менее актуальной. Сегодня процедурное программирование предпочитают использовать только в тех случаях, когда нельзя применить объектно-ориентированный подход. Он сейчас доминирует в мире императивных языков и относится к современным парадигмам разработки.

Императивное программирование развивалось как процедурное, в центре которого находилось понятие функции.

> Функция, или процедура, представляет собой набор действий, которые были зафиксированы и получили имя.

Например, инструкция по сборке мебели или рецепт приготовления тыквенного супа можно рассматривать как функции.

Процедура и функция обозначают одно и то же, однако процедурное и функциональное программирование имеют свои различия. Первое относится к императивной парадигме, второе — к декларативной. В свое время процедурное программирование стало настоящей революцией, заменило такие языки, как Assembler. Но в наши дни оно отошло на второй план, уступило место объектно-ориентированному программированию.

В объектно-ориентированном программировании объекты обладают способностью выполнять функции самостоятельно. Например, если суп готовит некий объект, который создала программа. Хотя все команды в компьютере выполняет процессор, в объектно-ориентированном программировании

объекты — это как бы отдельные персонажи со своими способностями. Они могут делать разные вещи, например, считать, рисовать или хранить информацию.

Этот подход позволяет устанавливать четкие связи между разными частями программы, каждая из которых отвечает за свою область деятельности. Благодаря этому процесс разработки становится удобнее — программу проще распределить между разработчиками, ее легче поддерживать и проверять автоматически с помощью тестов.

Объектно-ориентированная парадигма считается одной из самых простых и универсальных среди современных парадигм программирования, что и объясняет ее популярность. В следующих разделах мы подробнее рассмотрим основные аспекты ООП.

Вопрос 3. Концепции ООП

Объектно-ориентированное программирование — одно из ключевых парадигм разработки программного обеспечения.

Парадигма — это система правил и стандартов, которыми руководствуются разработчики при создании кода.

Например, в кулинарной книге рецепты написаны в определенном формате: сначала список ингредиентов, потом пошаговая инструкция по приготовлению блюда. Этот формат и есть парадигма.

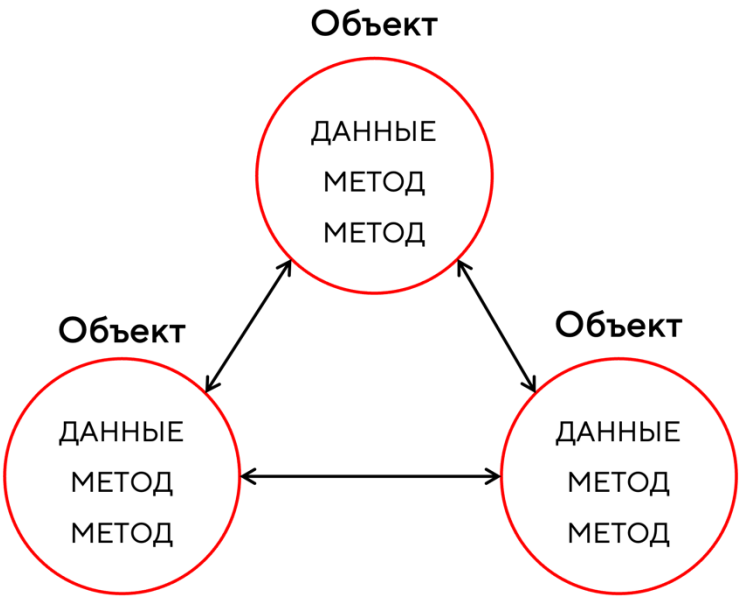
В программировании парадигма — это тоже своего рода стандарт. Она определяет, как нужно писать код, чтобы его понимали другие программисты и чтобы его было легко читать и менять. Парадигма помогает избежать ошибок, ускоряет разработку и стандартизирует процесс написания кода.

Основная идея объектно-ориентированного программирования заключается в том, что программы, которые создают с его использованием, состоят из объектов. Каждый объект представляет собой определенную сущность, которая обладает собственными данными и набором возможных операций.

Рассмотрим пример: при создании каталога товаров для интернет-магазина в первую очередь нужно создать объекты, а именно карточки товаров. Затем следует заполнить эти карточки информацией: названием товара, его характеристиками и ценой. После этого можно определить доступные действия для каждого объекта, такие как обновление, изменение и взаимодействие.

Структура объектно-ориентированного программирования

Схема 1. Структура кода по принципу ООП



В коде, который создали в соответствии с принципами ООП, выделяют четыре ключевых компонента:

Объект — это фрагмент кода, представляющий собой элемент с определенными характеристиками и функциями. Например, карточка товара в интернет-магазине — это объект, так же, как и кнопка «заказать».

Класс — это своего рода шаблон, на основе которого можно создавать объекты. На примере интернет-магазина можно представить класс «Карточка товара», который описывает общую структуру всех карточек. Из этого класса впоследствии формируются конкретные объекты — индивидуальные карточки товаров. Классы могут наследоваться друг от друга. Так, у нас может быть основной класс «Карточка товара», а также производные классы или подклассы, такие как «Карточка бытовой техники», «Карточка ноутбука» и «Карточка смартфона». Подклассы наследуют атрибуты родительского класса, такие как цена товара, количество единиц на складе или производитель, но при этом могут иметь собственные уникальные характеристики. Например, для «Карточки ноутбука» можно добавить свойство «диагональ экрана», а для «Карточки смартфона» — количество поддерживаемых сим-карт.

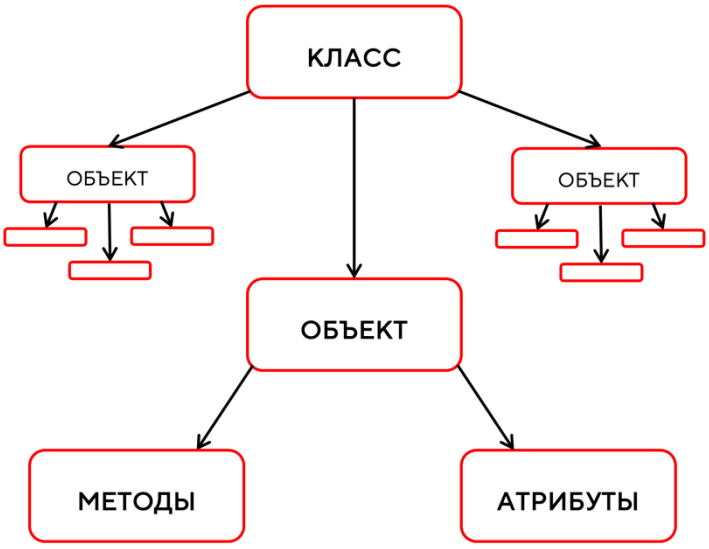
Метод — это функция, которая находится внутри объекта или класса и позволяет ему взаимодействовать с другими компонентами кода. В контексте карточек товара метод может выполнять следующие задачи:

- заполнять информацию о конкретном товаре;

- актуализировать количество доступных товаров, проверяя данные в базе данных;
- сравнивать характеристики двух различных товаров;
- предлагать покупателю аналоги или похожие товары.

Атрибут — это характеристики объекта в программировании, такие как цена, производитель или объем оперативной памяти. В классе указывают наличие этих атрибутов, а в отдельных объектах они заполняются данными с помощью методов.

Схема 2. Компоненты ООП



Основные принципы объектно-ориентированного программирования

Объектно-ориентированное программирование базируется на трех основных принципах, которые обеспечивают удобство использования этой парадигмы: инкапсуляция, наследование и полиморфизм. Рассмотрим каждый из них подробнее.

Инкапсуляция. Вся информация, которая нужна, чтобы объект функционировал, должна находиться внутри него самого. Изменения вносятся также через методы, которые реализуются непосредственно в этом объекте — внешние классы и объекты не имеют на это права. Для внешнего мира доступны только публичные методы и атрибуты.

Например, метод для внесения данных в карточку товара обязательно должен находиться в классе «Карточка товара», а не в классах «Корзина» или «Каталог товаров». Эта концепция обеспечивает безопасность и предотвращает случайное повреждение данных, которые находятся в одном классе, со стороны других классов. Кроме того, инкапсуляция помогает избежать нежелательных зависимостей, из-за которых изменение одного объекта может нарушить работу другого.

Наследование. В этом принципе — вся суть объектно-ориентированного программирования.

Разработчик создает:

- класс с определенными свойствами;
- подкласс на его основе, который берет свойства класса и добавляет свои;
- объект подкласса, который также копирует его свойства и добавляет свои.

Каждый дочерний элемент наследует методы и атрибуты, прописанные в родительском. Он может использовать их все, отбросить часть или добавить новые. При этом заново прописывать эти атрибуты и методы не нужно.

Например, в каталоге товаров у класса «Карточка товара» есть атрибуты тип товара, название, цена, производитель, а также методы «Вывести карточку» и «Обновить цену».

Подкласс «Смартфон» берет все атрибуты и методы, записывает в атрибут «тип товара» слово «смартфон» добавляет свои атрибуты — «Количество сим-карт» и «Емкость аккумулятора».

Объект «Смартфон Xiaomi» заполняет все атрибуты своими значениями и может использовать методы класса «Карточка товара».

Наследование хорошо видно в примере кода выше, когда сначала создавали класс, потом подкласс, а затем объект с общими свойствами.

Полиморфизм. Работает как волшебная кнопка, которая может делать разные вещи в зависимости от того, где она находится и какие цифры вы на ней нажимаете.

Примерно так же работает метод в объектно-ориентированном программировании. Метод — это команда, которую можно дать объекту. Но один и тот же метод может работать по-разному в зависимости от того, какому объекту вы его приказываете и какие данные при этом передаете.

Например, метод «Удалить» будет работать по-разному. Если применить его к корзине для покупок, он уберет товар только из нее, а если к карточке товара, то удалит саму карточку из каталога. Этот механизм также относится и к объектам. Мы можем применять их публичные методы и атрибуты в других функциях и быть уверенными в корректной работе этого взаимодействия.

Полиморфизм, как и другие принципы объектно-ориентированного программирования, способствует снижению количества ошибок при работе с объектами.

Преимущества и недостатки ООП

Основная функция объектно-ориентированного программирования — облегчить написание больших, сложных программ, над которыми трудятся группы разработчиков.

Таблица 2. Преимущества и недостатки ООП

Преимущества	Недостатки
Код проще создать. Вы можете один раз определить класс или метод и далее просто их использовать. Не нужно многократно переписывать одни и те же строки. Кроме того, существуют полезные рекомендации для написания ООП-кода, такие как принципы SOLID, которые помогают в структурировании и улучшении кода	Сложность в освоении. ООП сложнее, чем функциональное программирование. Для написания кода в этой парадигме нужно знать гораздо больше. Поэтому перед созданием первой рабочей программы придется освоить много информации: разобраться в классах и наследовании, научиться писать публичные и внутренние функции, изучить способы взаимодействия объектов между собой
Код легче читать. Даже в чужом коде обычно сразу видны конкретные объекты и методы, их удобно искать, чтобы посмотреть, что именно они делают	Громоздкость. Там, где в функциональном программировании хватит одной функции, в ООП нужно создать класс, объект, методы и атрибуты. Для больших программ это плюс, так как структура будет понятной, а для маленьких может оказаться лишней тратой времени
Код проще обновлять. Класс или метод достаточно изменить в одном месте, чтобы он изменился во всех наследуемых классах и объектах. Не нужно переписывать каждый объект отдельно, выискивать, где именно в коде он расположен	Низкая производительность. Объекты потребляют больше памяти, чем простые функции и переменные. Скорость компиляции от этого тоже страдает
Программистам удобнее работать в команде. Разные люди отвечают за разные объекты и при этом пользуются плодами трудов коллег	
Код можно переиспользовать. Один раз написанный класс или объект можно затем переносить в другие	

проекты. Достаточно однажды написать объект «Кнопка заказа» и потом можно вставлять его в почти неизменном виде в разные каталоги товаров и мобильные приложения	
Можно создать шаблоны проектирования. Именно на базе ООП построены готовые решения для взаимодействия классов друг с другом. Можно не писать этот код с нуля, а взять шаблон	

Получается, что основная функция объектно-ориентированного программирования — облегчить написание больших, сложных программ, над которыми трудятся группы разработчиков.

Есть несколько популярных языков объектно-ориентированного программирования. Сами по себе они не могут быть объектно-ориентированными. ООП — это парадигма, которую можно применять для написания кода на любом языке. Определенные языки подходят для использования объектно-ориентированного программирования больше, так как предоставляют удобные инструменты для работы с классами и объектами:

- Java;
- Go;
- Python;
- C++;
- JavaScript;
- C#;
- PHP;
- Ruby;
- Scala;
- Kotlin;
- Swift;
- Dart.

Типы моделей в объектно-ориентированном моделировании и проектировании

Смысл объектно-ориентированного подхода в том, чтобы использовать его идеи и методы на протяжении всей работы над программой — от создания общего плана до финальных правок.

Этот подход помогает решать задачи при помощи моделей, которые основываются на идеях, похожих на те, что мы видим в реальной жизни. Ключевой элемент этой схемы — объект, который объединяет в себе как данные, так и функции, что позволяет более эффективно описывать и управлять сложными системами.

Назначение моделей:

- тестировать физический объект перед тем, как его создать;
- общаться с заказчиками;
- визуализировать;
- снижать сложность.

Типы моделей:

- тестировать физический объект перед тем, как его создать;
- общаться с заказчиками;
- визуализировать;
- снижать сложность.

Таблица 3. Типы моделей в ООП

Тип	Определение
Модель класса	Показывает все классы, которые присутствуют в системе. Модель класса показывает атрибуты и поведение, связанные с объектами. **Диаграмма классов** используется для отображения модели класса. На диаграмме классов показано имя класса, за которым следуют атрибуты, за которыми следуют функции или методы, связанные с объектом класса. Цель построения модели класса — охватить те концепции из реального мира, которые важны для приложения.
Модель состояния	Описывает те аспекты объектов, которые связаны со временем и последовательностью операций — события, которые отмечают изменения, состояния, которые определяют контекст для событий, организацию событий и состояний. Действия и события на диаграмме состояний становятся операциями над объектами в модели класса. **Диаграмма состояний** описывает модель состояния.
Модель взаимодействия	Используется для демонстрации разных взаимодействий между объектами, того, как объекты взаимодействуют для достижения поведения системы в целом. **Следующие**

	диаграммы используются для демонстрации модели взаимодействия: • схема вариантов использования; • схема последовательности; • схема действий.

Вопрос 4. Диаграммы последовательности, кооперации и деятельности

UML — унифицированный язык моделирования. Из этих трех слов главное слово **«язык»**. Что же такое язык?

Язык — система знаков, которая служит средством человеческого общения и мыслительной деятельности, способом выражения самосознания личности, средством хранения и передачи информации.

Язык представляет собой систему символов (словарный запас) и набор правил их использования и интерпретации (грамматика).

Это определение достаточно полное, однако стоит упомянуть, что языки могут быть как естественными, так и искусственными, а также формальными и неформальными. UML относится к формальным и искусственным языкам, хотя, как мы увидим в дальнейшем, это определение не совсем точно отражает его суть. Искусственный UML потому, что был создан определенными авторами. При этом развитие UML продолжается, что сближает его с естественными языками.

Формальным его можно считать, поскольку существуют четкие правила его использования, хотя в описании UML присутствуют и некоторые неформальные элементы. Важно, что UML — это графический язык.

Когда описывают формальный искусственный язык, упоминаются такие его элементы, как:

- синтаксис, то есть правила, как построить конструкции языка;
- семантика, то есть правила, в соответствии с которыми конструкции языка приобретают смысловое значение;
- прагматика, то есть правила, по которым конструкции языка используются, чтобы можно было достичь нужных целей.

Естественно, UML включает все эти элементы, хотя, как мы опять-таки увидим далее, в их описании тоже наблюдаются отличия от правил, принятых в языках программирования.

Второе слово в фразе, которой расшифровывается аббревиатура UML — слово **«моделирование»**. UML — это язык моделирования. Причем объектно-

ориентированного моделирования. В английском языке есть целых два слова — modeling и simulation, которые оба переводятся как «моделирование», хотя означают разное. Modeling подразумевает создание модели, которая лишь описывает объект, а simulation предполагает, что можно получать с помощью созданной модели некоторую дополнительную информацию об объекте. UML в первую очередь — язык моделирования именно в первом смысле, то есть с его помощью строятся описательные модели. Как средство симулирования его тоже можно использовать, хотя для этой роли он подходит не так хорошо.

Третье слово в названии UML — слово **«унифицированный»**. Его тоже можно понять неоднозначно. В литературе встречается описание эры «до UML» как «войны методов» моделирования, ни один из которых «не дотягивал» до уровня индустриального стандарта. UML как раз и стал таким единым универсальным стандартом для объектно-ориентированного моделирования, которое во времена его создания как раз «вошло в моду».

«Единым» языком моделирования UML можно назвать еще и потому, что в его создании, как мы увидим далее, объединились усилия авторов трех наиболее популярных методов моделирования.

Подводя итоги, кратко можно сказать, что UML — искусственный язык, который имеет некоторые черты естественного языка, и формальный язык, который имеет черты неформального.

UML-диаграмма — это схема, нарисованная с применением символов UML. Она может содержать множество элементов и соединений между ними. Полное описание масштабного проекта может состоять из нескольких (многих) UML-диаграмм, связанных или не связанных между собой.

Элементами диаграммы UML могут быть классы программного кода, страницы сайта, части механизма, зоны торгового зала — в зависимости от того, какой процесс или какую сущность описывает ее создатель.

UML представляет собой относительно открытый стандарт, который находится под управлением группы OMG (Object Management Group — группа управления объектами), открытого консорциума компаний. Группу OMG сформировали, чтобы создать стандарты, которые поддерживают межсистемное взаимодействие, в частности взаимодействие объектно-ориентированных систем.

Типы диаграмм UML

1. Структурные.

2. Поведенческие.

Структурные диаграммы описывают статическую структуру системы, ее компоненты и связи между ними.

Поведенческие диаграммы описывают, как система ведет себя в ответ на различные действия или события.

Внутри этих категорий много других. Рассмотрим виды поведенческих диаграмм: диаграмму последовательности, диаграмму кооперации и диаграмму деятельности.

Диаграмма последовательности

Диаграмма последовательности — или **sequence diagram** — это способ наглядно показать, как разные элементы в модели системы взаимодействуют друг с другом и в каком порядке это происходит.

Схема 3. Клиент снимает деньги с банковской карты

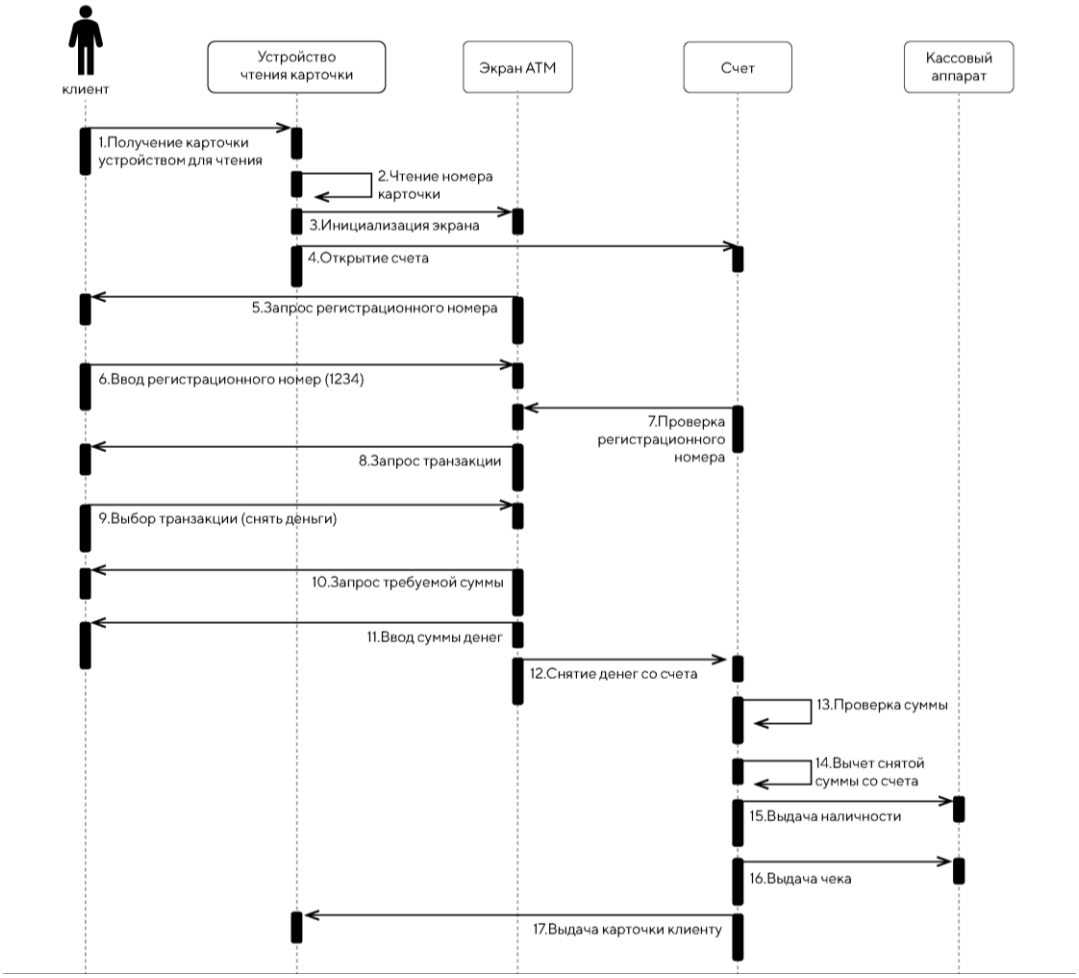
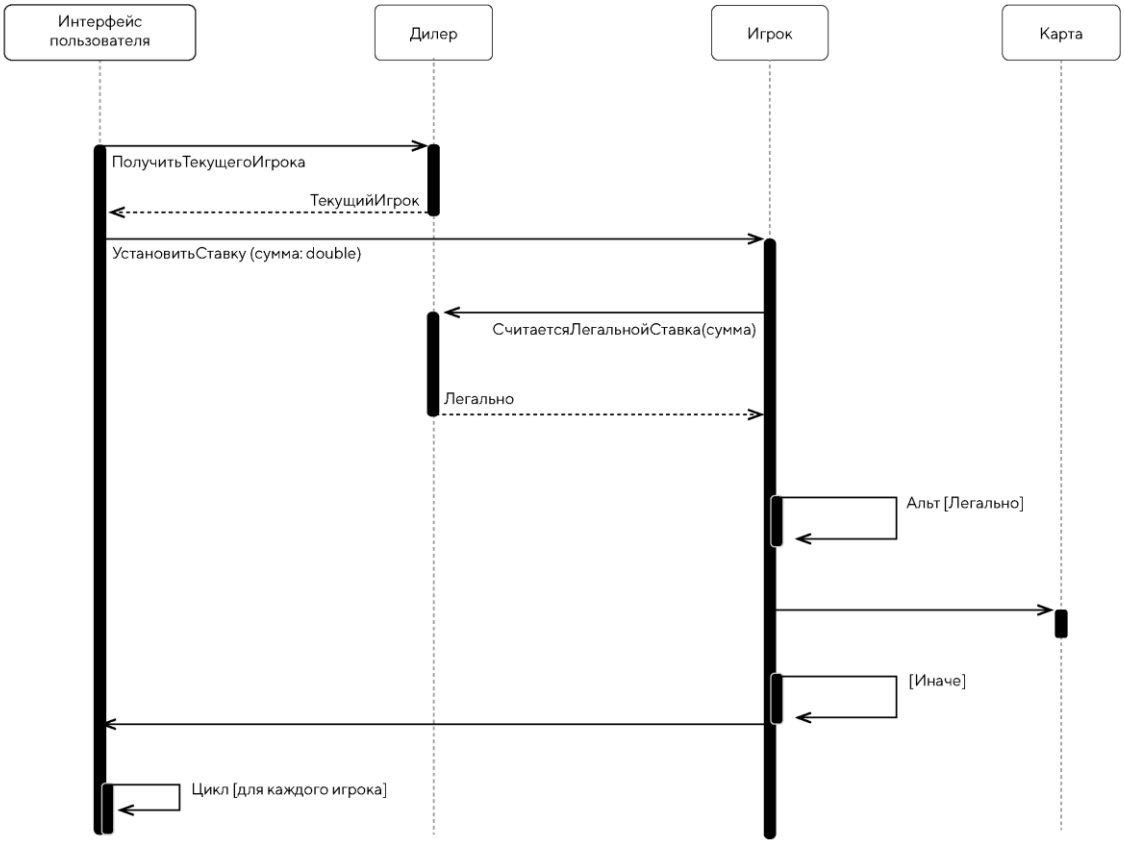


Диаграмма на иллюстрации выше показывает, как снимают деньги с банковской карты: человек вставляет ее в устройство для чтения, оно считывает номер, информация отображается на экране, устройство запрашивает код. Когда код проверен, последовательно выполняется транзакция, деньги снимаются со счета и выдаются клиенту.

Элементы диаграммы последовательности

Схема 4. Элементы диаграммы последовательности



1. Объект. Объекты системы в таких диаграммах всегда представлены прямоугольниками с именами внутри. В прямоугольнике помимо названия объекта указывают его класс. Обе эти части разделяются двоеточием.

Часто название класса или объекта опускают. Если указан только класс, перед ним стоит двоеточие. По принципу построения для более четкой и понятной диаграммы лучше помещать объекты с более частым взаимодействием

ближе друг к другу, а тот объект, который инициализирует активность, — Actor — в левый угол.

2. Линия жизни (Life Line). Временная шкала в диаграмме последовательности идет сверху вниз. У каждого объекта своя пунктирная линия, их еще называют линиями жизни, которые обозначают последовательности действий объектов и не пересекаются.

3. Фрагмент выполнения или активация (Activation Bar). Узкий прямоугольник на линии жизни, который показывает начало и завершение действия с участием объекта, его активизации во времени. В начале примера с банковским сценарием клиент предоставляет карточку для чтения, а устройство считывает ее номер, причем наглядно можно понять, что эти действия начинаются одновременно, но одно из них заканчивается раньше.

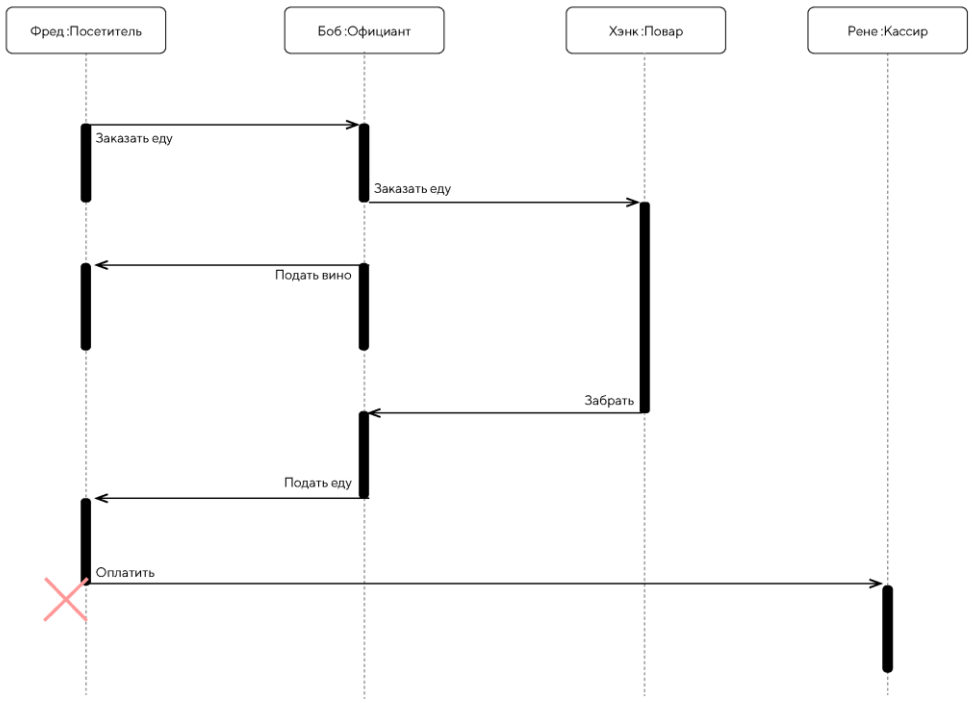
4. Сообщение (Message). Стрелка, которая показывает взаимодействие между разными объектами в момент, когда они посылают сообщения друг другу. Стрелки помогают понять смысл взаимодействия и то, кто его начал и по отношению к кому. Сообщения могут говорить также о создании или уничтожении участников последовательности.

Если стрелка закрашена, то активация происходит синхронно. Асинхронное сообщение говорит о том, что отправитель сигнала передает ход получателю, чтобы тот произвел некое действие.

Пунктирная стрелка — это возвратное или ответное сообщение. Она говорит о том, что получатель обработал предыдущее сообщение и передает ход обратно отправителю. Этот элемент не всегда появляется в диаграммах последовательности, так как автоматически подразумевается при синхронном сообщении. Если минимизировать возвратные сообщения, диаграмма получится более лаконичной.

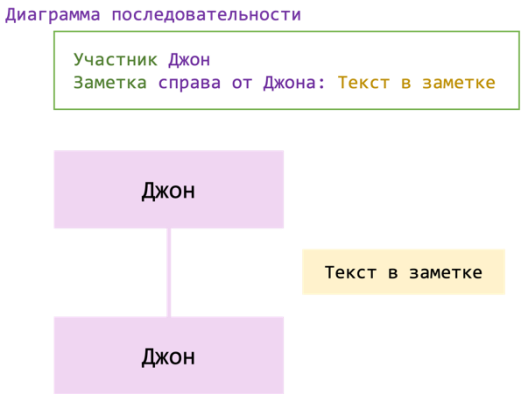
5. Уничтожение объекта. Знак уничтожения объекта показывает конец его линии жизни в этом взаимодействии — момент, когда объект перестает иметь значение в последовательности. Он обозначается диагональным крестом, как, например, на следующей диаграмме.

Схема 5. Клиента обслуживают в ресторане



6. Примечание (Note). Это комментарий с информацией для разработчика или аналитика, который можно прикрепить к элементу.

Схема 6. Текст в заметке



Как построить диаграмму последовательности

Распишем последовательность действий. Например, чтобы создать диаграмму последовательности для процесса оформления онлайн-заказа, надо выполнить следующие шаги:

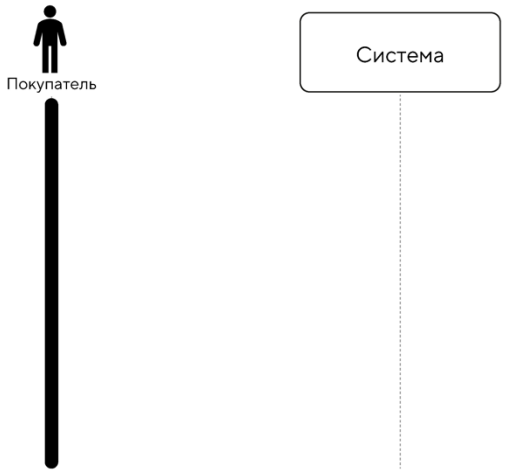
- покупатель вводит в систему данные заказа;
- система получает их, обрабатывает и создает заказ;

- получателю приходит уведомление о создании заказа.

В последовательности участвуют два объекта: покупатель и система. На каждом этапе взаимодействия заранее пропишем, какими сообщениями они будут обмениваться.

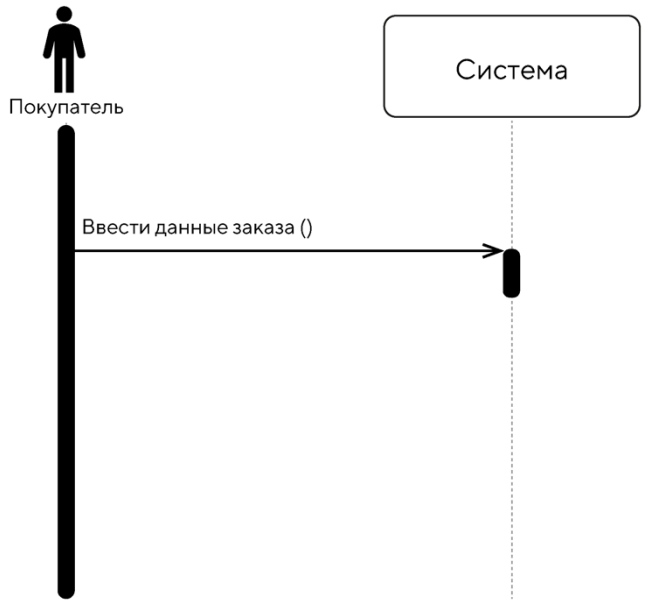
1. Создаем объекты «покупатель» и «система» — это можно сделать в бесплатном сервисе по построению диаграмм.

Схема 7. Объекты в диаграмме последовательности



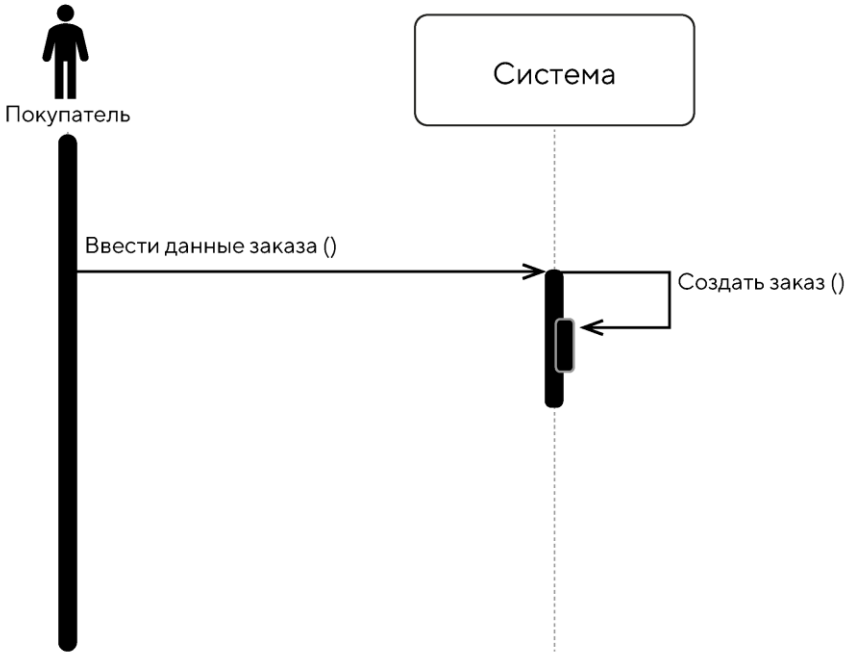
2. Создаем сообщение о вводе данных заказа.

Схема 8. Ввод заказа



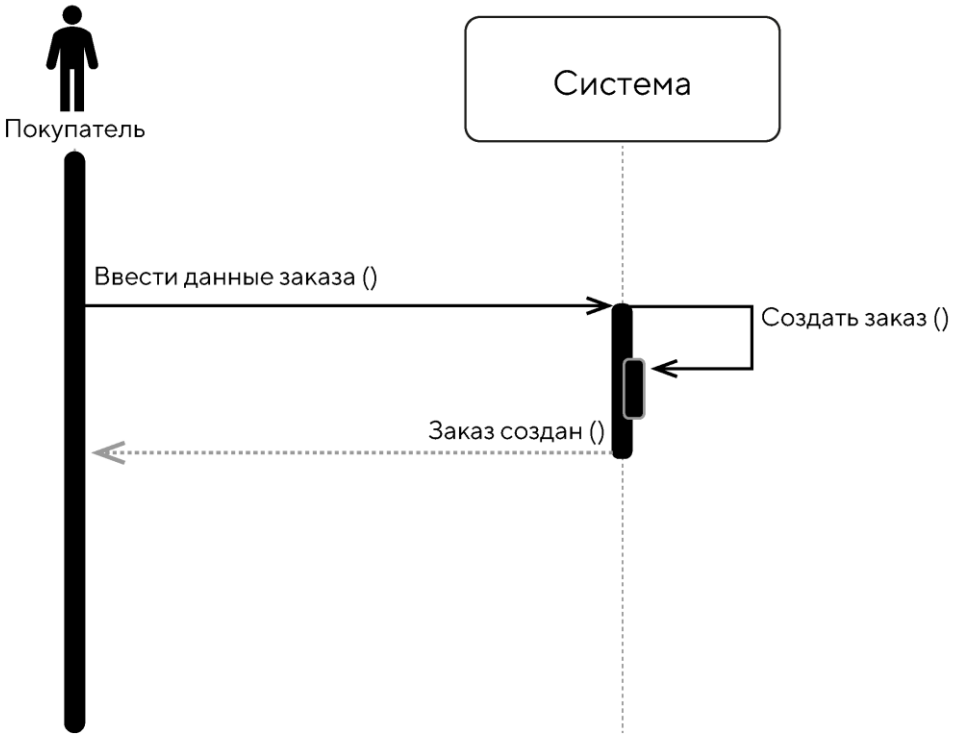
3. Система создает заказ.

Схема 9. Система создает заказ



4. Система отправляет покупателю возвратное сообщение о том, что все сделано.

Схема 10. Возвратное сообщение покупателю



По такому же принципу создаются и более сложные диаграммы последовательности.

Самые частые ошибки при построении таких диаграмм

1. Большое количество деталей — лучше не загромождать диаграмму лишними мелкими действиями, это затруднит ее чтение.
2. Элементы диаграммы меняются вместе с развитием инструмента, и некоторые из них раньше выглядели по-другому. Для того, чтобы диаграмма была всем понятна, нужно, чтобы данные на ней были актуальными.
3. Диаграммы не соответствуют интерфейсу или архитектуре проекта, поэтому надо проверять их, чтобы вовремя заменить или усовершенствовать.
4. Следует все проверять, когда создаете диаграммы последовательности. Любые, даже самые мелкие, изменения означают, что придется двигать все элементы. Это трата времени.

Диаграммы кооперации (взаимодействия)

Диаграммы взаимодействия иллюстрируют связи между объектами, которые действуют в определенной ситуации.

Они во многом перекликаются с диаграммами последовательностей, однако акцент здесь делается не на временной динамике взаимодействий, а на взаимосвязях между объектами и их структурной конфигурацией.

На диаграммах кооперации сообщения, которые отправляют от одного объекта к другому, изображаются стрелками с указанием имени, параметров и порядка отправки.

Эти диаграммы отлично подходят, чтобы визуализировать рабочий процесс программы или проанализировать конкретную ситуацию.

Также они позволяют быстро продемонстрировать или объяснить логические процессы в рамках программы.

Пример 1. Диаграмма кооперации (взаимодействия)

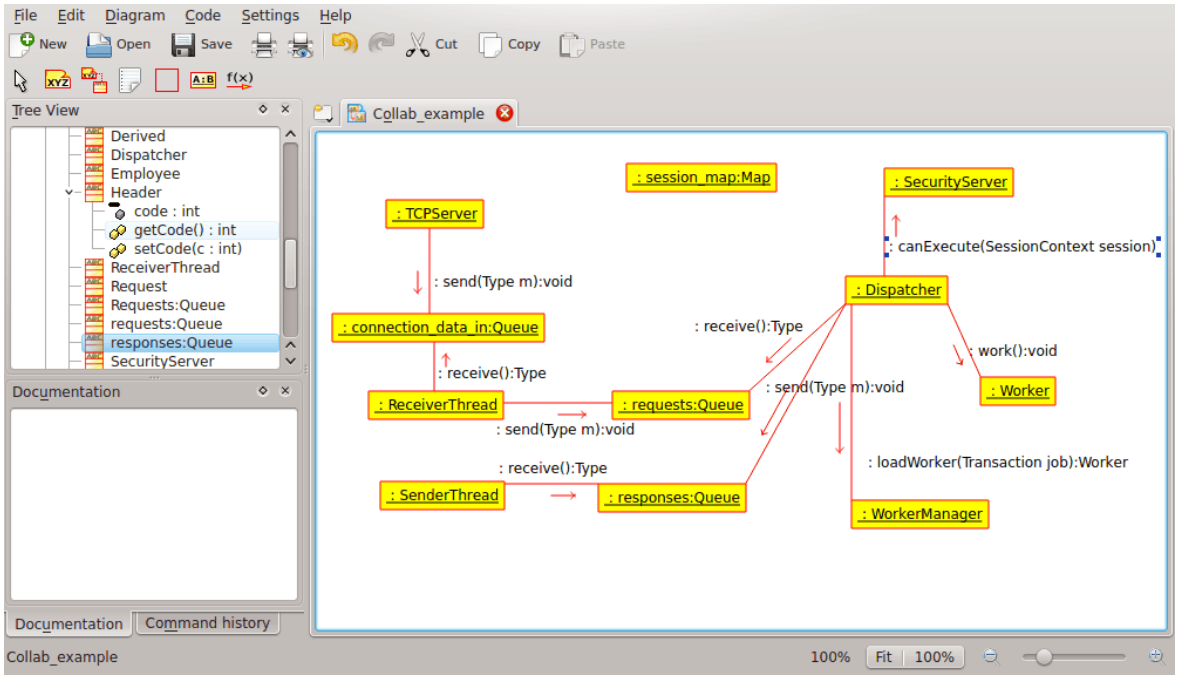


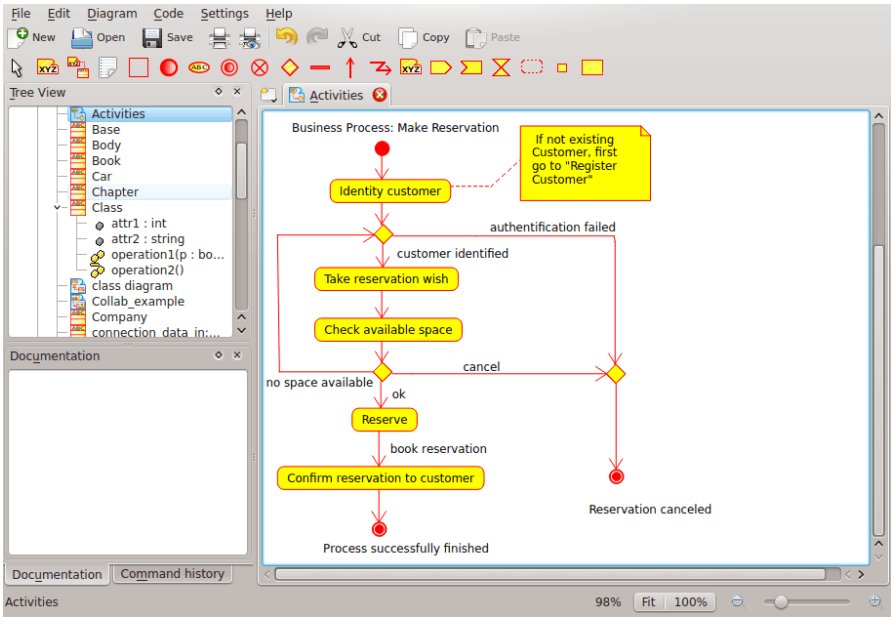
Диаграмма деятельности (активности)

Описывают последовательность действий в системе с помощью видов деятельности.

Диаграммы деятельности, или активности, — это специальные формы диаграмм состояний, в основном содержат виды деятельности.

На изображении представлена диаграмма активности, которая иллюстрирует процесс бронирования. Она описывает последовательность шагов, которые необходимо выполнить для успешного завершения бронирования.

Пример 2. Диаграмма деятельности (активности)



Описание процесса

1. Начало процесса — процесс начинается с идентификации клиента. Если клиент не существует, необходимо перейти к этапу регистрации (заметка в желтой рамке справа вверху).
2. Идентификация клиента (Identify customer) — проверяется, существует ли клиент в системе. Если аутентификация не удалась, процесс завершится на этапе отмены.
3. Принятие желания бронирования (Take reservation wish) — после идентификации клиента принимается его желание сделать бронирование.
4. Проверка доступного пространства (Check available space) — проверяется наличие свободных мест. Если мест нет, процесс завершится отменой бронирования. Если места есть, процесс продолжается.
5. Резервирование (Reserve) — бронирование места для клиента.
6. Подтверждение бронирования клиенту (Confirm reservation to customer) — после успешного бронирования отправляется подтверждение клиенту.
7. Завершение процесса — если бронирование успешно, процесс заканчивается сообщением о том, что бронирование завершено (Process successfully finished).