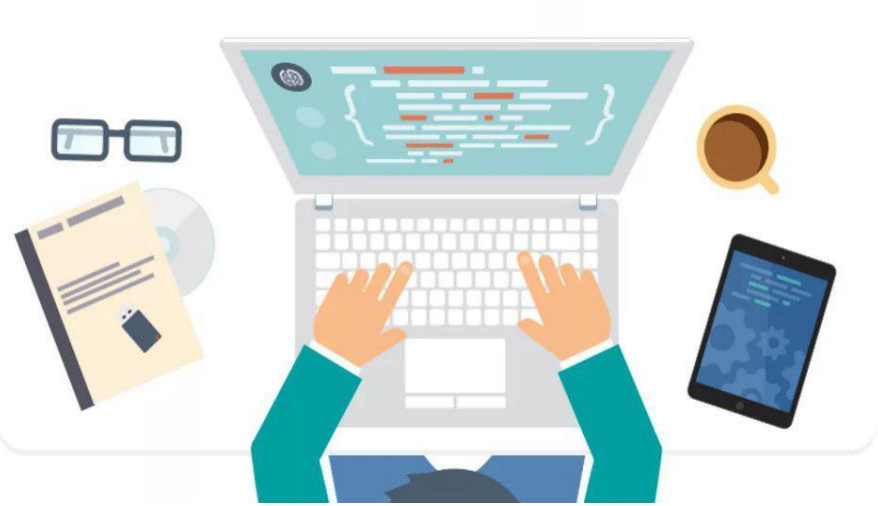
GIT, UML, GCC

Инструментарий разработчика.

Для чего нужна система контроля версий.

- Пока проект состоит из пары-тройки файлов, его разработка не создаёт никаких сложностей. Программист пишет код, запускает его и радуется жизни. Клиент доволен, заказчик тоже. С ростом кодовой базы появляются определённые неудобства, которые затем превращаются в реальные проблемы:
  - Как не потерять файлы с исходным кодом?
  - Как защититься от случайных исправлений и удалений?
  - Как отменить изменения, если они оказались некорректными?
  - Как одновременно поддерживать рабочую версию и разработку новой?
- Совместная разработка — это отдельная головная боль. Если два программиста работают над задачами, требующими исправления кода в одних и тех же файлах, то как они выполнят эту работу так, чтобы не повредить или перезаписать изменения другого разработчика?

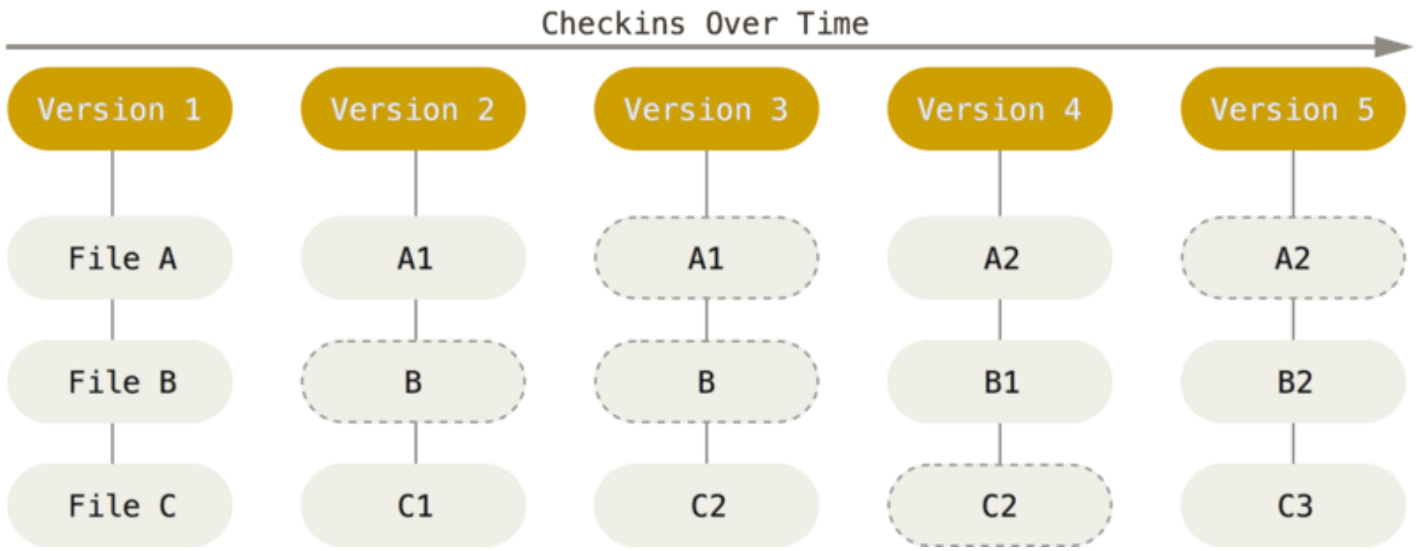


Система контроля версий GIT

- Обязана своему появлению в 2005г. Линусу Торвальду
- Некоторыми целями, которые преследовала новая система, были:
  - ✓ Скорость
  - ✓ Простая архитектура
  - ✓ Хорошая поддержка нелинейной разработки (тысячи параллельных веток)
  - ✓ Полная децентрализация
  - ✓ Возможность эффективного управления большими проектами, такими как ядро Linux (скорость работы и разумное использование дискового пространства)
- Git хранит и использует информацию совсем иначе по сравнению с другими системами, даже несмотря на то, что интерфейс пользователя достаточно похож.

Как работает GIT?

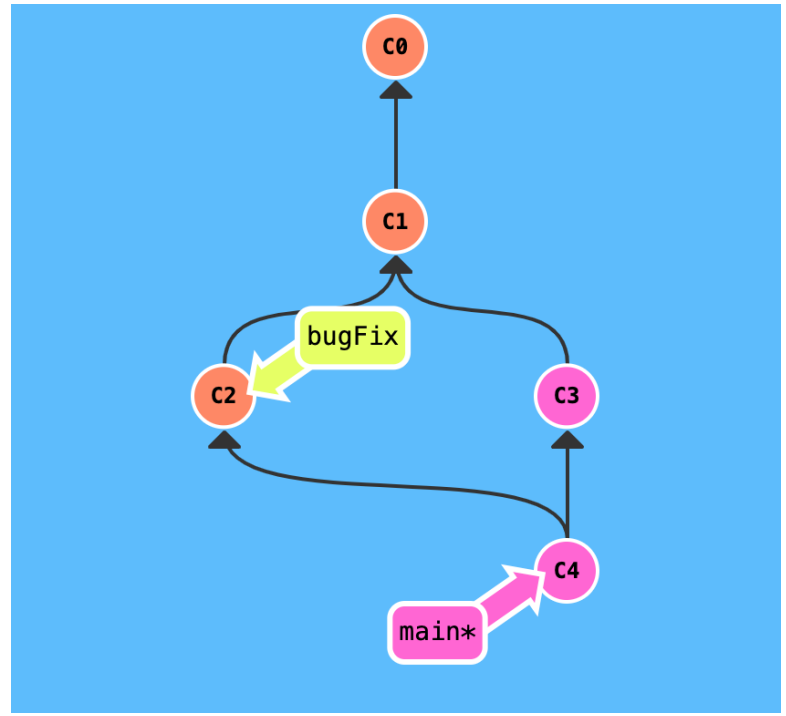
- Подход **Git** к хранению данных похож на набор снимков миниатюрной файловой системы. Каждый раз, когда вы делаете коммит, то есть сохраняете состояние своего проекта в **Git**, система запоминает, как выглядит каждый файл в этот момент, и сохраняет ссылку на этот снимок. Для увеличения эффективности, если файлы не были изменены, **Git** не запоминает эти файлы вновь, а только создаёт ссылку на предыдущую версию идентичного файла, который уже сохранён. Git представляет свои данные как **поток снимков**.



Варианты работы с GIT

Git может быть:

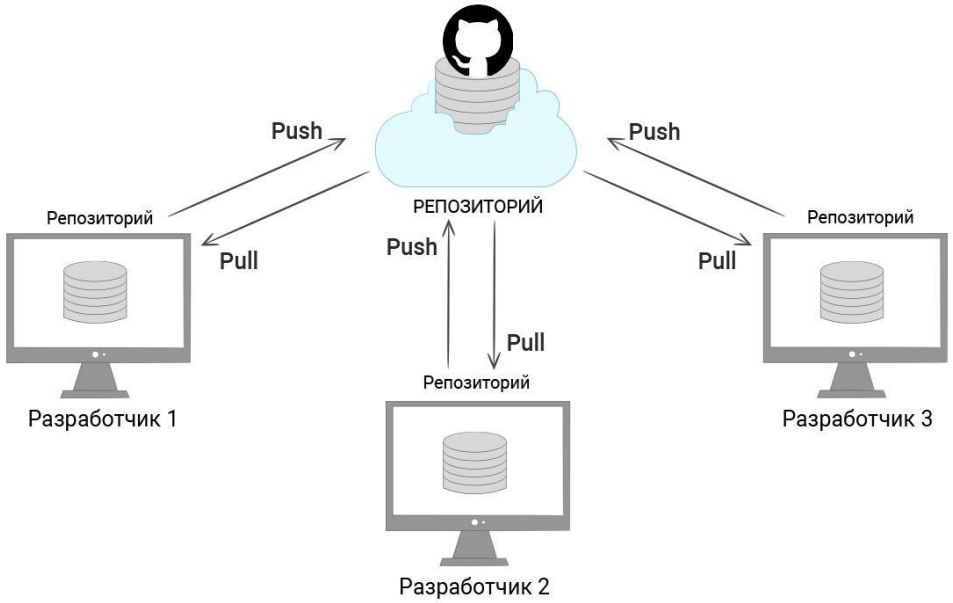
- **Локальный** установлен на одном компьютере и хранит файлы только в одном экземпляре в рамках настроенного окружения — подходит, если программист пишет код в одиночку.
- **Централизованный** находится на общем севере и хранит все файлы на нем.
- **Распределённый** хранит данные и в общем облачном хранилище, и в устройствах участников команды.



Распределённая система лучше всего подходит для командной работы. Даже если с центральным хранилищем что-то случится, проект можно восстановить из копий участников команды.

Удобство и гибкость сделали **Git** стандартом для большинства современных IT-компаний. Умение работать с ним критично для любого программиста.

Что такое репозиторий Git



- **Репозиторий** — это место, в котором хранится весь код и вся история его изменений. По сути это просто папка, однако она связана с **Git** напрямую и содержит файлы в понятном для **Git** формате. Кроме того, для папки, заявленной как репозиторий, **Git** формирует историю изменений.

- Репозиторий может быть **локальным** — храниться на компьютере пользователя. А может быть **удалённым** — лежать на сервере или в облачном хранилище. В таком случае пользователи со своих устройств подключаются к этому репозиторию через интернет.

Зачем нужен GitHub

- **Git** — это просто программа, которую нужно установить и подключить к своему проекту. Можно установить её на сервер и настроить удалённую работу самостоятельно. А можно воспользоваться уже готовыми сервисами. Самый популярный из них — **GitHub**.
- По сути **GitHub** — это сайт-хранилище. Нужно сначала установить **Git**, потом зарегистрироваться на **GitHub**, создать там онлайн-репозиторий — и перенести туда файлы из своего репозитория. Можно настроить автоматический перенос и многие другие функции, которые позволят работать с кодом совместно.
- На **GitHub** можно создавать публичные или открытые проекты — это позволяет знакомить со своим кодом других людей. А можно приватные или закрытые, доступные только тем, кто работает над кодом.

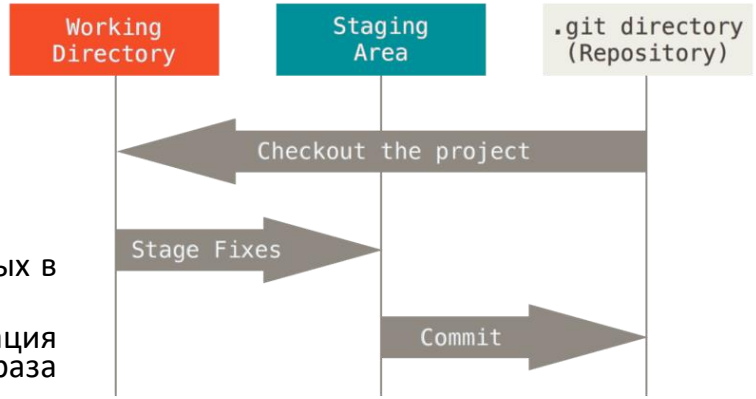
Прежде чем начнем работать с GIT !

- У **Git** есть три основных состояния, в которых могут находиться ваши файлы:
  ***изменён* (modified), *индексирован* (staged)** и ***зафиксирован* (committed):**

К *изменённым* относятся файлы, которые поменялись, но ещё не были зафиксированы.

Индексированный — это изменённый файл в его текущей версии, отмеченный для включения в следующий коммит.

Зафиксированный значит, что файл уже сохранён в вашей локальной базе.



- Существует 3 основные секции проекта **Git**:
  - ***рабочая копия* (working tree)**,
  - ***область индексирования* (staging area)**
  - ***каталог Git* (Git directory).**

  Рабочая копия является снимком одной версии проекта. Эти файлы извлекаются из сжатой базы данных в каталоге Git и помещаются на диск, для того чтобы их можно было использовать или редактировать.

  Область индексирования — это файл, обычно находящийся в каталоге Git, в нём содержится информация о том, что попадёт в следующий коммит. Её техническое название на языке Git — «индекс», но фраза «область индексирования» также работает.

  Каталог Git — это то место, где Git хранит метаданные и базу объектов вашего проекта. Это самая важная часть Git и это та часть, которая копируется при *клонировании* репозитория с другого компьютера.

- Базовый подход в работе с Git выглядит так:
  - ✓ Изменяете файлы вашей рабочей копии.
  - ✓ Выборочно добавляете в область индексирования только те изменения, которые должны попасть в следующий коммит, добавляя тем самым снимки *только* этих изменений в область индекс-я.
  - ✓ Когда вы делаете коммит, используются файлы из области индекса как есть, и этот снимок сохраняется в ваш каталог Git.

- Если определённая версия файла есть в каталоге Git, эта версия считается *зафиксированной* (committed). Если файл был изменён и добавлен в индекс, значит, он *индексирован* (staged). И если файл был изменён с момента последнего распаковывания из репозитория, но не был добавлен в индекс, он считается *изменённым* (modified).

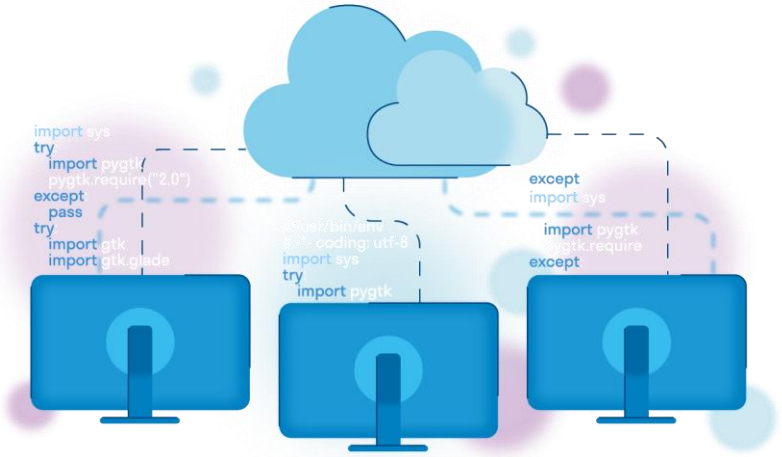
Принципы работы с Git

При работе с Git в среде разработчиков принято руководствоваться тремя принципами:

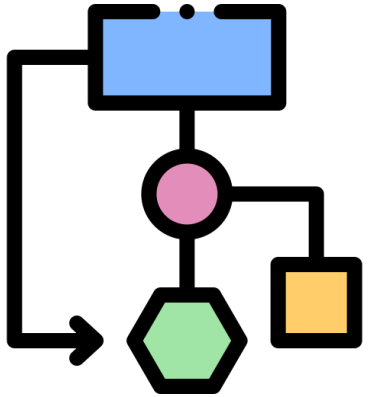
1. Регулярно коммитить — сохранять изменения в Git. Такой подход позволит сохранять более подробную историю версий и быстро замечать ошибки в коде.

2. Создавать новые ветки. Они позволяют легко управлять изменениями, особенно параллельными. Лучше создать ещё одну ветку, чем что-то испортить в старой.

3. Чётко и лаконично описывать коммиты. Изменения кода, которые отправляются в Git, обязательно должны содержать пояснения и комментарии по добавленным правкам, доработкам и изменениям. Это значительно облегчает совместную работу и помогает быстрее разбираться в своем старом коде.



Диаграммы UML



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. А так же:

- **Проектирование**. UML-диаграммы помогут при моделировании архитектуры больших проектов, 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 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.

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 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.
- Например, 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).

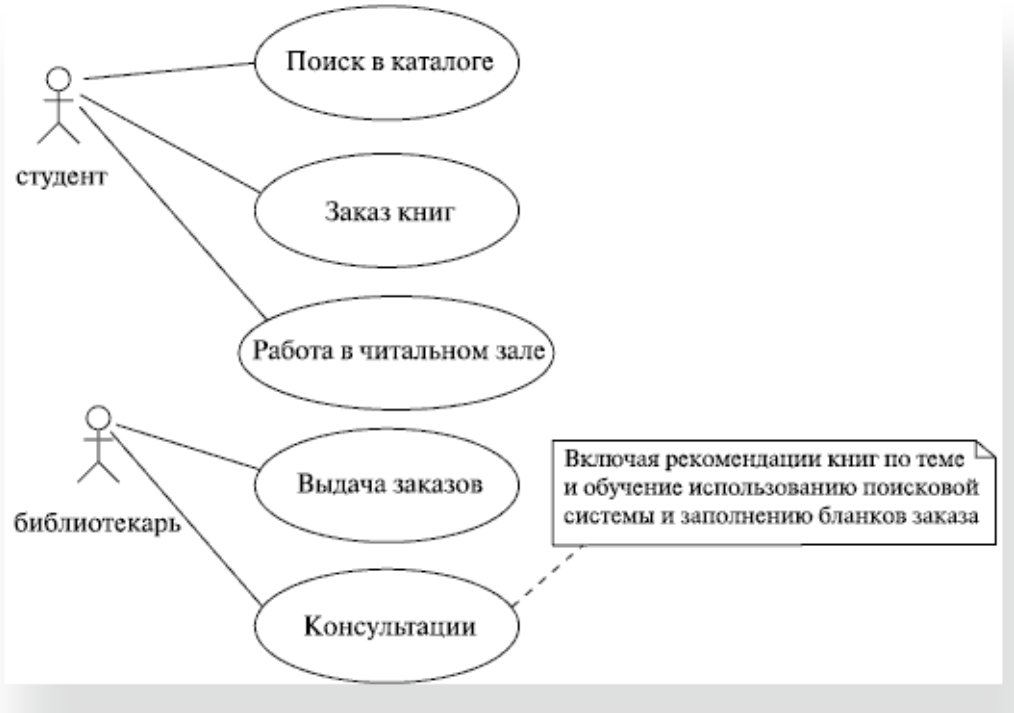


Диаграмма классов — Class diagram

- **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.
- 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".

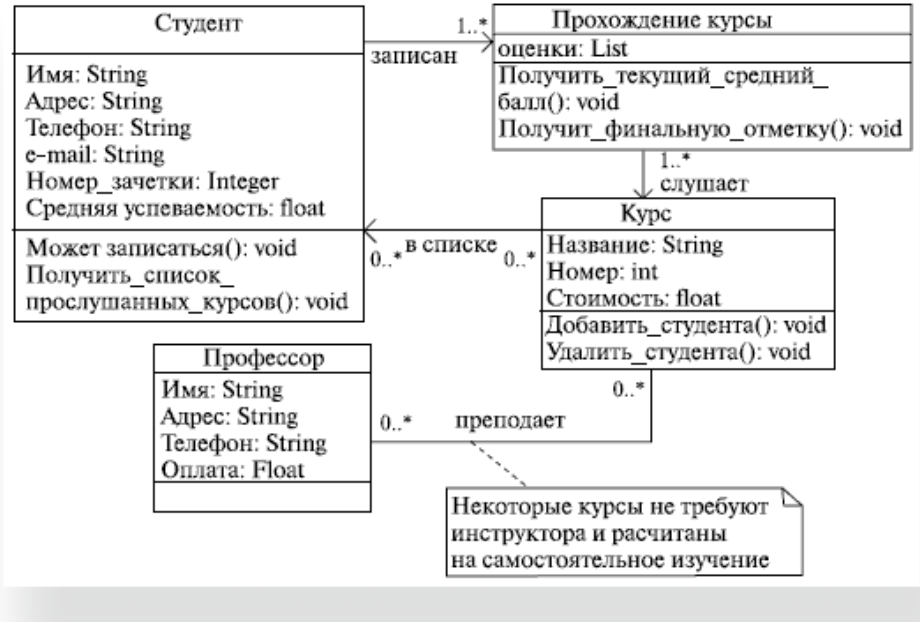
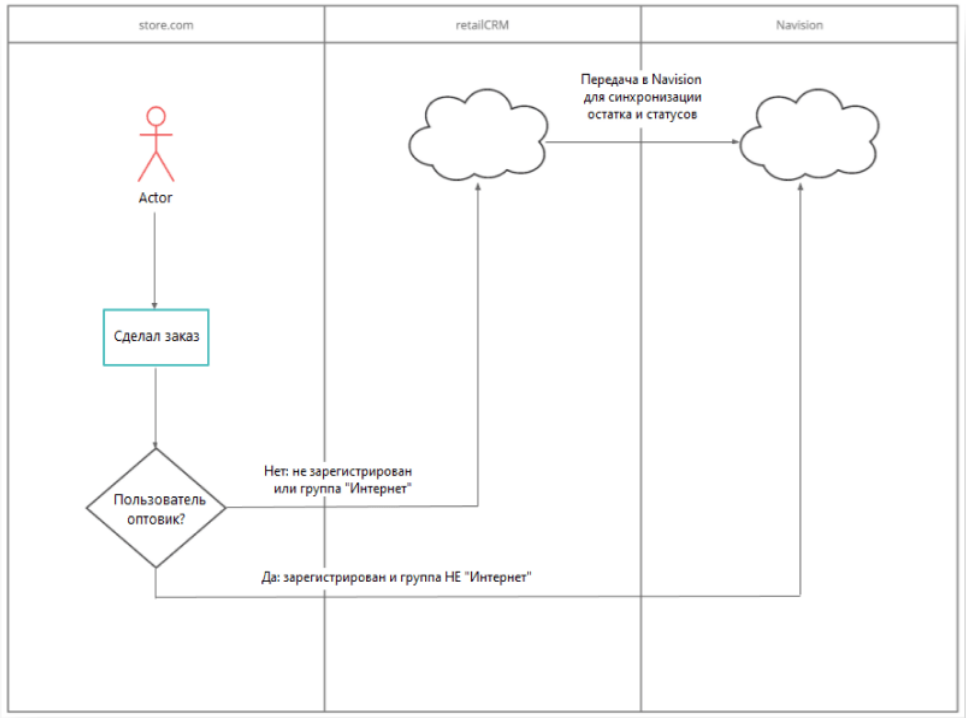


Диаграмма активностей — Activity diagram

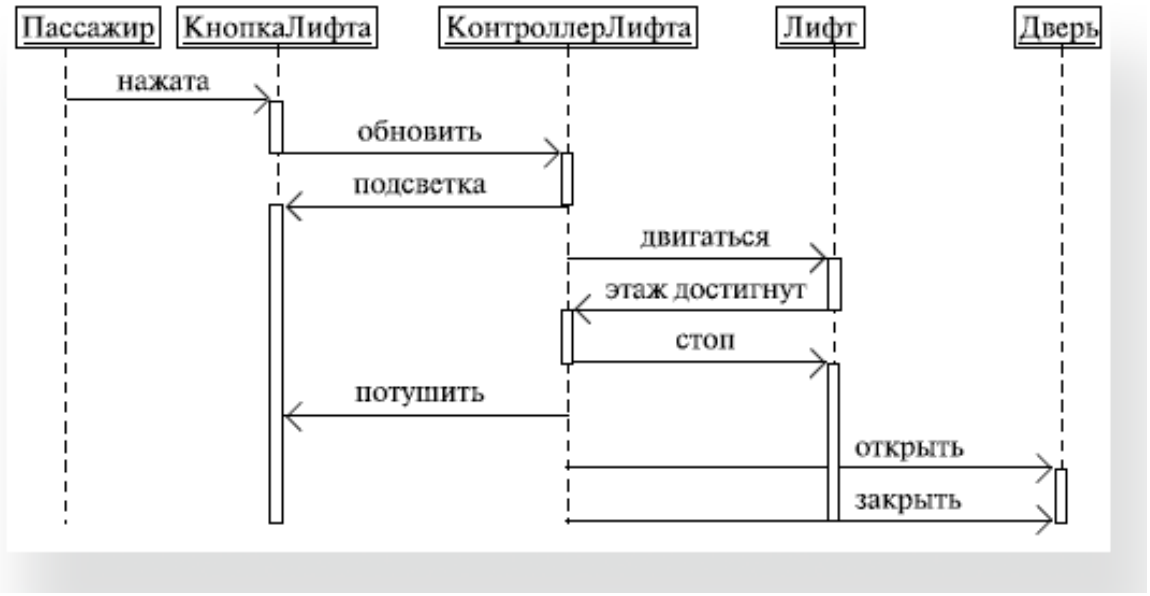
- Диаграмма активностей описывает динамические аспекты поведения системы в виде блок-схемы, которая отражает бизнес-процессы, логику процедур и потоки работ — переходы от одной деятельности к другой. По сути, мы рисуем алгоритм действий (логику поведения) системы или взаимодействия нескольких систем. Ниже — пример подобной диаграммы для интернет-магазина.
- Эту базовую диаграмму мы можем дополнить, расширить, она может выступить частью документации и дает общее представление о работе системы.



Evergreen

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.



Ресурсы для работы с диаграммами UML

- https://plantuml.com/ru/
- https://www.planttext.com/
- https://miro.com/ru/diagramming/uml-diagram/
- MS Visio.

Компилятор GCC

- **GNU Compiler Collection** (обычно используется сокращение **GCC**) — набор компиляторов для различных языков программирования, разработанный в рамках проекта GNU.
- GCC является свободным программным обеспечением, распространяется на условиях GNU GPL и GNU LGPL и является ключевым компонентом GNU toolchain.
- Изначально названный **GNU C Compiler**, поддерживал только язык Си. Позднее GCC был расширен для компиляции исходных кодов на таких языках программирования, как C++, Objective-C, Java, Фортран, Ada, Go, GAS и D.

Работа с компилятором GCC

- Программа **gcc**, запускаемая из командной строки. В зависимости от расширений имен файлов, передаваемых в качестве параметров, и дополнительных опций, **gcc** запускает необходимые препроцессоры, компиляторы, линкеры.
- Файлы с расширением **.cc** или **.C** рассматриваются, как файлы на языке C++, файлы с расширением **.c** как программы на языке C, а файлы с расширением **.o** считаются объектными.
- Чтобы откомпилировать исходный код C++, находящийся в файле **F.cc**, и создать объектный файл **F.o**, необходимо выполнить команду:

$ gcc -c F.cc

Опция **–c** означает «только компиляция».

- Чтобы скомпоновать один или несколько объектных файлов, полученных из исходного кода - **F1.o**, **F2.o**, ... - в единый исполняемый файл **F**, необходимо ввести команду:

$ gcc -o F F1.o F2.o

Опция -o задает имя исполняемого файла.

- Можно совместить два этапа обработки - компиляцию и компоновку - в один общий этап с помощью команды:

$ gcc -o F <compile-and-link-options> F1.cc ... -lg++ <other-libraries>

<compile-and-link –options> - возможные дополнительные опции компиляции и компоновки. Опция –lg++ указывает на необходимость подключить стандартную библиотеку языка C++, <other-libraries> - возможные дополнительные библиотеки.

- После компоновки будет создан исполняемый файл F, который можно запустить с помощью команды

$ gcc ./F <arguments>

<arguments> - список аргументов командной строки Вашей программы.

- В процессе компоновки очень часто приходится использовать библиотеки.

Библиотеки могут быть подключены с помощью опции вида -lname. В этом случае в стандартных каталогах, таких как **/lib , /usr/lib, /usr/local/lib** будет проведен поиск библиотеки в файле с именем **libname.a**. Библиотеки должны быть перечислены после исходных или объектных файлов, содержащих вызовы к соответствующим функциям.

Опция	Назначение
-c	Эта опция означает, что необходима только компиляция. Из исходных файлов программы создаются объектные файлы в виде name.o. Компоновка не производится.
-Dname=value	Определить имя name в компилируемой программе, как значение value. Эффект такой же, как наличие строки #define name value в начале программы. Часть =value может быть опущена, в этом случае значение по умолчанию равно 1.
-o file-name	Использовать file-name в качестве имени для создаваемого файла.
-lname	Использовать при компоновке библиотеку libname.so
-Llib-path -Iinclude-path	Добавить к стандартным каталогам поиска библиотек и заголовочных файлов пути lib-path и include-path соответственно.
-g	Поместить в объектный или исполняемый файл отладочную информацию для отладчика gdb. Опция должна быть указана и для компиляции, и для компоновки. В сочетании –g рекомендуется использовать опцию отключения оптимизации –O0 (см.ниже)
-MM	Вывести зависимости от заголовочных файлов , используемых в Си или C++ программе, в формате, подходящем для утилиты make. Объектные или исполняемые файлы не создаются.
-pg	Поместить в объектный или исполняемый файл инструкции профилирования для генерации информации, используемой утилитой gprof. Опция должна быть указана и для компиляции, и для компоновки. Собранная с опцией -pg программа при запуске генерирует файл статистики. Программа gprof на основе этого файла создает расшифровку, указывающую время, потраченное на выполнение каждой функции.
-Wall	Вывод сообщений о всех предупреждениях или ошибках, возникающих во время компиляции программы.
-O1 -O2 -O3	Различные уровни оптимизации.
-O0	Не оптимизировать. Если вы используете многочисленные -O опции с номерами или без номеров уровня, действительной является последняя такая опция.
-I	Используется для добавления ваших собственных каталогов для поиска заголовочных файлов в процессе сборки
-L	Передается компоновщику. Используется для добавления ваших собственных каталогов для поиска библиотек в процессе сборки.
-l	Передается компоновщику. Используется для добавления ваших собственных библиотек для поиска в процессе сборки.

Требует ознакомления

- https://git-scm.com/book/ru/v2/%D0%92%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D0%B8%D0%B5-%D0%9E-%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B5-%D0%BA%D0%BE%D0%BD%D1%82%D1%80%D0%BE%D0%BB%D1%8F-%D0%B2%D0%B5%D1%80%D1%81%D0%B8%D0%B9
- http://www-cs-students.stanford.edu/~blynn/gitmagic/intl/ru/index.html

На сегодня Все!