1

Подготовка стенда

1.1 Программная инженерия

1.1.1 Семантическое версионирование

Семантический подход в версионированию программного обеспечения.

1.1.1.1 Краткое описание семантического версионирования

- Семантическое версионирование описывается в манифесте семантического версионирования.
- Кратко его можно описать следующим образом:
  - Версия задаётся в виде кортежа `МАЖОРНАЯ_ВЕРСИЯ.МИНОРНАЯ_ВЕРСИЯ.ПАТЧ`.
  - Номер версии следует увеличивать:
    - МАЖОРНУЮ версию, когда сделаны обратно несовместимые изменения API.
    - МИНОРНУЮ версию, когда вы добавляете новую функциональность, не нарушая обратной совместимости.
    - ПАТЧ-версию, когда вы делаете обратно совместимые исправления.
  - Дополнительные обозначения для предрелизных и билд-метаданных возможны как дополнения к МАЖОРНАЯ.МИНОРНАЯ.ПАТЧ формату.

1.1.1.2 Принципы

— Коммиты должны иметь стандартизованный вид.
— В семантическое версионирование применяется вместе с *общепринятыми коммитами*.

1.1.2 Общепринятые коммиты

Использование спецификации *Conventional Commits*.

1.1.2.1 Описание

Спецификация Conventional Commits:

— Соглашение о том, как нужно писать сообщения commit'ов.
— Совместимо с SemVer.
— Регламентирует структуру и основные типы коммитов.

1.1.2.1.1 Структура коммита

```
<type>(<scope>): <subject>
<BLANK LINE>
<body>
<BLANK LINE>
<footer>
```

Или, по-русски:

```
<тип>(<область>): <описание изменения>
<пустая линия>
[необязательное тело]
<пустая линия>
[необязательный нижний колонтитул]
```

— Заголовок является обязательным.
— Любая строка сообщения о фиксации не может быть длиннее 100 символов.

- Тема (subject) содержит краткое описание изменения.
  - Используйте повелительное наклонение в настоящем времени: «изменить» («change» not «changed» nor «changes»).
  - Не используйте заглавную первую букву.
  - Не ставьте точку в конце.
- Тело (body) должно включать мотивацию к изменению и противопоставлять это предыдущему поведению.
  - Как и в теме, используйте повелительное наклонение в настоящем времени.
- Нижний колонтитул (footer) должен содержать любую информацию о критических изменениях.
  - Следует использовать для указания внешних ссылок, контекста коммита или другой мета информации.
  - Также содержит ссылку на issue (например, на github), который закрывает эта фиксация.
  - Критические изменения должны начинаться со слова `BREAKING CHANGE:` с пробела или двух символов новой строки. Затем для этого используется остальная часть сообщения фиксации.

1.1.2.1.2 Базовые типы коммитов

- `fix:` — коммит типа fix исправляет ошибку (bug) в вашем коде (он соответствует PATCH в SemVer).
- `feat:` — коммит типа feat добавляет новую функцию (feature) в ваш код (он соответствует MINOR в SemVer).
- `BREAKING CHANGE:` — коммит, который содержит текст `BREAKING CHANGE:` в начале своего не обязательного тела сообщения (body) или в подвале (footer), добавляет изменения, нарушающие обратную совместимость вашего API (он соответствует MAJOR в SemVer). BREAKING CHANGE может быть частью коммита любого типа.
- `revert:` — если фиксация отменяет предыдущую фиксацию. Начинается с `revert:`, за которым следует заголовок отменённой фиксации. В теле должно быть написано: Это отменяет фиксацию `<hash>` (это SHA-хэш отменяемой фиксации).
- Другое: коммиты с типами, которые отличаются от `fix:` и `feat:`, также разрешены. Например, `@commitlint/config-conventional` (основанный на The Angular convention) рекомендует: chore:, docs:, style:, refactor:, perf:, test:, и другие.

1.1.2.1.3 Соглашения The Angular convention

Одно из популярных соглашений о поддержке исходных кодов — конвенция Angular (The Angular convention).

1.1.2.1.3.1 Типы коммитов The Angular convention

Конвенция Angular (The Angular convention) требует следующие типы коммитов:

- `build:` — изменения, влияющие на систему сборки или внешние зависимости (примеры областей (scope): gulp, broccoli, npm).
- `ci:` — изменения в файлах конфигурации и скриптах CI (примеры областей: Travis, Circle, BrowserStack, SauceLabs).
- `docs:` — изменения только в документации.
- `feat:` — новая функция.
- `fix:` — исправление ошибок.
- `perf:` — изменение кода, улучшающее производительность.
- `refactor:` — Изменение кода, которое не исправляет ошибку и не добавляет функции (рефакторинг кода).
- `style:` — изменения, не влияющие на смысл кода (пробелы, форматирование, отсутствие точек с запятой и т. д.).
- `test:` — добавление недостающих тестов или исправление существующих тестов.

1.1.2.1.3.2 Области действия (scope)

Областью действия должно быть имя затронутого пакета npm (как его воспринимает человек, читающий журнал изменений, созданный из сообщений фиксации).

Есть несколько исключений из правила «использовать имя пакета»:

- `packaging` — используется для изменений, которые изменяют структуру пакета, например, изменения общедоступного пути.
- `changelog` — используется для обновления примечаний к выпуску в `CHANGELOG.md`.
- отсутствует область действия — полезно для изменений стиля, тестирования и рефакторинга, которые выполняются во всех пакетах (например, style: добавить отсутствующие точки с запятой).

1.2 Использование git

1.2.1 Первоначальна настройка git

1.2.1.1 Системы контроля версий. Общие понятия

— *Системы контроля версий (Version Control System, VCS)* применяются при работе нескольких человек над одним проектом. Обычно основное дерево проекта хранится в локальном или удалённом репозитории, к которому настроен доступ для участников проекта. При внесении изменений в содержание проекта система контроля версий позволяет их фиксировать, совмещать изменения, произведённые разными участниками проекта, производить откат к любой более ранней версии проекта, если это требуется.
— В классических системах контроля версий используется централизованная модель, предполагающая наличие единого репозитория для хранения файлов. Выполнение большинства функций по управлению версиями осуществляется специальным сервером. Участник проекта (пользователь) перед началом работы посредством определённых команд получает нужную ему версию файлов. После внесения изменений, пользователь размещает новую версию в хранилище. При этом предыдущие версии не удаляются из центрального хранилища и к ним можно вернуться в любой момент. Сервер может сохранять не полную версию изменённых файлов, а производить так называемую дельта-компрессию — сохранять только изменения между последовательными версиями, что позволяет уменьшить объём хранимых данных.
— Системы контроля версий поддерживают возможность отслеживания и разрешения конфликтов, которые могут возникнуть при работе нескольких человек над одним файлом. Можно объединить (слить) изменения, сделанные разными участниками (автоматически или вручную), вручную выбрать нужную версию, отменить изменения вовсе или заблокировать файлы для изменения. В зависимости от настроек блокировка не позволяет другим пользователям получить рабочую копию или препятствует изменению рабочей копии файла средствами файловой системы ОС, обеспечивая таким образом, привилегированный доступ только одному пользователю, работающему с файлом.
— Системы контроля версий также могут обеспечивать дополнительные, более гибкие функциональные возможности. Например, они могут поддерживать работу с несколькими версиями одного файла, сохраняя общую историю изменений до точки ветвления версий и собственные истории изменений каждой ветви. Кроме того, обычно доступна информация о том, кто из участников, когда и какие изменения вносил. Обычно такого рода информация хранится в журнале изменений, доступ к которому можно ограничить.
— В отличие от классических, в распределённых системах контроля версий цен-

тральный репозиторий не является обязательным.
- Среди классических VCS наиболее известны CVS, Subversion, а среди распределённых — Git, Bazaar, Mercurial. Принципы их работы схожи, отличаются они в основном синтаксисом используемых в работе команд.

1.2.1.2 Примеры использования git

- Система контроля версий Git представляет собой набор программ командной строки. Доступ к ним можно получить из терминала посредством ввода команды `git` с различными опциями.
- Благодаря тому, что Git является распределённой системой контроля версий, резервную копию локального хранилища можно сделать простым копированием или архивацией.

1. Основные команды git
   - Перечислим наиболее часто используемые команды `git`.
   - Создание основного дерева репозитория:

     ```
     git init
     ```

   - Получение обновлений (изменений) текущего дерева из центрального репозитория:

     ```
     git pull
     ```

   - Отправка всех произведённых изменений локального дерева в центральный репозиторий:

     ```
     git push
     ```

   - Просмотр списка изменённых файлов в текущей директории:

     ```
     git status
     ```

   - Просмотр текущих изменений:

     ```
     git diff
     ```

   - Сохранение текущих изменений:
     - Добавить все изменённые и/или созданные файлы и/или каталоги:

       ```
       git add .
       ```

     - Добавить конкретные изменённые и/или созданные файлы и/или каталоги:

```
git add имена_файлов
```

  — Удалить файл и/или каталог из индекса репозитория (при этом файл и/или каталог остаётся в локальной директории):

```
git rm имена_файлов
```

— Сохранение добавленных изменений:
  — Сохранить все добавленные изменения и все изменённые файлы:

```
git commit -am 'Описание коммита'
```

  — Сохранить добавленные изменения с внесением комментария через встроенный редактор:

```
git commit
```

  — Создание новой ветки, базирующейся на текущей:

```
git checkout -b имя_ветки
```

  — Переключение на некоторую ветку:

```
git checkout имя_ветки
```

    — При переключении на ветку, которой ещё нет в локальном репозитории, она будет создана и связана с удалённой.
  — Отправка изменений конкретной ветки в центральный репозиторий:

```
git push origin имя_ветки
```

  — Слияние ветки с текущим деревом:

```
git merge --no-ff имя_ветки
```

— Удаление ветки:
  — удаление локальной уже слитой с основным деревом ветки:

```
git branch -d имя_ветки
```

  — принудительное удаление локальной ветки:

```
git branch -D имя_ветки
```

  — удаление ветки с центрального репозитория:

```
git push origin :имя_ветки
```

2. Стандартные процедуры работы при наличии центрального репозитория
   — Работа пользователя со своей веткой начинается с проверки и получения

изменений из центрального репозитория (при этом в локальное дерево до начала этой процедуры не должно было вноситься изменений):

```
git checkout master
git pull
git checkout -b имя_ветки
```

— Затем можно вносить изменения в локальном дереве и/или ветке.
— После завершения внесения какого-то изменения в файлы и/или каталоги проекта необходимо разместить их в центральном репозитории. Для этого необходимо проверить, какие файлы изменились к текущему моменту:

```
git status
```

— При необходимости удаляем лишние файлы, которые не хотим отправлять в центральный репозиторий.
— Затем полезно просмотреть текст изменений на предмет соответствия правилам ведения чистых коммитов:

```
git diff
```

— Если какие-либо файлы не должны попасть в коммит, то помечаем только те файлы, изменения которых нужно сохранить. Для этого используем команды добавления и/или удаления с нужными опциями:

```
git add …
git rm …
```

— Если нужно сохранить все изменения в текущем каталоге, то используем:

```
git add .
```

— Затем сохраняем изменения, поясняя, что было сделано:

```
git commit -am "Some commit message"
```

— Отправляем изменения в центральный репозиторий:

```
git push origin имя_ветки
```

или

```
git push
```

3. Работа с локальным репозиторием
   — Создадим локальный репозиторий.

— Сначала сделаем предварительную конфигурацию, указав имя и email владельца репозитория:

```
git config --global user.name "Имя Фамилия"
git config --global  user.email "work@mail"
```

— Настроим utf-8 в выводе сообщений `git`:

```
git config --global quotepath false
```

— Для инициализации локального репозитория, расположенного, например, в каталоге `~/tutorial`, необходимо ввести в командной строке:

```
cd
mkdir tutorial
cd tutorial
git init
```

— После это в каталоге `tutorial` появится каталог `.git`, в котором будет храниться история изменений.
— Создадим тестовый текстовый файл `hello.txt` и добавим его в локальный репозиторий:

```
echo 'hello world' > hello.txt
git add hello.txt
git commit -am 'Новый файл'
```

— Воспользуемся командой `status` для просмотра изменений в рабочем каталоге, сделанных с момента последней ревизии:

```
git status
```

— Во время работы над проектом так или иначе могут создаваться файлы, которые не требуется добавлять в последствии в репозиторий. Например, временные файлы, создаваемые редакторами, или объектные файлы, создаваемые компиляторами. Можно прописать шаблоны игнорируемых при добавлении в репозиторий типов файлов в файл `.gitignore` с помощью сервисов. Для этого сначала нужно получить список имеющихся шаблонов:

```
curl -L -s https://www.gitignore.io/api/list
```

— Затем скачать шаблон, например, для C и C++

```
curl -L -s https://www.gitignore.io/api/c >> .gitignore
curl -L -s https://www.gitignore.io/api/c++ >> .gitignore
```

4. Работа с сервером репозиториев

— Для последующей идентификации пользователя на сервере репозиториев необходимо сгенерировать пару ключей (приватный и открытый):

```
ssh-keygen -C "Имя Фамилия <work@mail>"
```

— Ключи сохраняться в каталоге `~/.ssh/`.
— Существует несколько доступных серверов репозиториев с возможностью бесплатного размещения данных. Например, https://github.com/.
— Для работы с ним необходимо сначала завести на сайте https://github.com / учётную запись. Затем необходимо загрузить сгенерённый нами ранее открытый ключ.
— Для этого зайти на сайт https://github.com/ под своей учётной записью и перейти в меню *GitHub setting*.
— После этого выбрать в боковом меню *GitHub setting>SSH-ключи* и нажать кнопку *Добавить ключ*. Скопировав из локальной консоли ключ в буфер обмена:

```
cat ~/.ssh/id_rsa.pub | xclip -sel clip
```

— Вставляем ключ в появившееся на сайте поле.
— После этого можно создать на сайте репозиторий, выбрав в меню , дать ему название и сделать общедоступным (публичным).
— Для загрузки репозитория из локального каталога на сервер выполняем следующие команды:

```
git remote add origin
  ssh://git@github.com/<username>/<reponame>.git
git push -u origin master
```

— Далее на локальном компьютере можно выполнять стандартные процедуры для работы с `git` при наличии центрального репозитория.

1.2.1.3 Базовая настройка git

1. Первичная настройка параметров git
   — Зададим имя и email владельца репозитория:

```
git config --global user.name "Name Surname"
git config --global user.email "work@mail"
```

   — Настроим utf-8 в выводе сообщений git:

```
git config --global core.quotepath false
```

   — Настройте верификацию и подписание коммитов git.

- Зададим имя начальной ветки (будем называть её `master`):

```
git config --global init.defaultBranch master
```

2. Учёт переносов строк
   - В разных операционных системах приняты разные символы для перевода строк:
     - Windows: `\r\n` (`CR` и `LF`);
     - Unix: `\n` (`LF`);
     - Mac: `\r` (`CR`).
   - Посмотреть значения переносов строк в репозитории можно командой:

```
git ls-files --eol
```

   1. Параметр `autocrlf`
      - Настройка `core.autocrlf` предназначена для того, чтобы в главном репозитории все переводы строк текстовых файлах были одинаковы.
      - Настройка `core.autocrlf` с параметрами `true` и `input` делает все переводы строк текстовых файлов в главном репозитории одинаковыми.
        - `core.autocrlf true`: конвертация `CRLF`→`LF` при коммите и обратно `LF`→`CRLF` при выгрузке кода из репозитория на файловую систему (обычно используется в Windows).
        - `core.autocrlf input`: конвертация `CRLF`→`LF` только при коммитах (используются в MacOS/Linux).
      - Варианты конвертации

core.autocrlf	false	input	true
git commit	LF → LF	LF → LF	LF → CRLF
	CR → CR	CR → CR	CR → CR
	CRLF → CRLF	CRLF → LF	CRLF → CRLF
git checkout	LF → LF	LF → LF	LF → CRLF
	CR → CR	CR → CR	CR → CR
	CRLF → CRLF	CRLF → CRLF	CRLF → CRLF

      Варианты конвертации для разных значений параметра core.autocrlf
      - Установка параметра:
        - Для Windows

```
git config --global core.autocrlf true
```

        - Для Linux

```
git config --global core.autocrlf input
```

2. Параметр `safecrlf`
   - Настройка `core.safecrlf` предназначена для проверки, является ли окончаний строк обратимым для текущей настройки `core.autocrlf`.
     - `core.safecrlf true`: запрещается необратимое преобразование `lf<->crlf`. Полезно, когда существуют бинарные файлы, похожие на текстовые файлы.
     - `core.safecrlf warn`: печать предупреждения, но коммиты с необратимым переходом принимаются.
   - Установка параметра:

```
git config --global core.safecrlf warn
```

1.2.1.4 Создание ключа ssh

1. Общая информация
   1. Алгоритмы шифрования ssh
      1. Аутентификация
         В SSH поддерживается четыре алгоритма аутентификации по открытым ключам:
         - DSA:
           - размер ключей DSA не может превышать 1024, его следует отключить;
         - RSA:
           - следует создавать ключ большого размера: 4096 бит;
         - ECDSA:
           - ECDSA завязан на технологиях NIST, его следует отключить;
         - Ed25519:
           - используется пока не везде.
      2. Симметричные шифры
         - Из 15 поддерживаемых в SSH алгоритмов симметричного шифрования, безопасными можно считать:
           - `chacha20-poly1305`;
           - `aes*-ctr`;
           - `aes*-gcm`.
         - Шифры `3des-cbc` и `arcfour` потенциально уязвимы в силу использования DES и RC4.
         - Шифр `cast128-cbc` применяет слишком короткий размер блока (64 бит).
      3. Обмен ключами
         - Применяемые в SSH методы обмена ключей DH (Diffie-Hellman) и ECDH (Elliptic Curve Diffie-Hellman) можно считать безопасными.

    - Из 8 поддерживаемых в SSH протоколов обмена ключами вызывают подозрения три, основанные на рекомендациях NIST:
      - `ecdh-sha2-nistp256`;
      - `ecdh-sha2-nistp384`;
      - `ecdh-sha2-nistp521`.
    - Не стоит использовать протоколы, основанные на *SHA1*.
2. Файлы ssh-ключей
    - По умолчанию пользовательские ssh-ключи сохраняются в каталоге `~/.ssh` в домашнем каталоге пользователя.
    - Убедитесь, что у вас ещё нет ключа.
    - Файлы закрытых ключей имеют названия типа `id_<алгоритм>` (например, `id_dsa`, `id_rsa`).
      - По умолчанию закрытые ключи имеют имена:

        ```
        id_dsa
        id_ecdsa
        id_ed25519
        id_rsa
        ```

    - Открытые ключи имеют дополнительные расширения `.pub`.
      - По умолчанию публичные ключи имеют имена:

        ```
        id_dsa.pub
        id_ecdsa.pub
        id_ed25519.pub
        id_rsa.pub
        ```

    - При создании ключа команда попросит ввести любую ключевую фразу для более надёжной защиты вашего пароля. Можно пропустить этот этап, нажав `Enter`.
    - Сменить пароль на ключ можно с помощью команды:

      ```
      ssh-keygen -p
      ```

2. Создание ключа ssh
    - Ключ ssh создаётся командой:

      ```
      ssh-keygen -t <алгоритм>
      ```

    - Создайте ключи:
      - по алгоритму *rsa* с ключём размером 4096 бит:

        ```
        ssh-keygen -t rsa -b 4096
        ```

      - по алгоритму *ed25519*:

        ```
        ssh-keygen -t ed25519
        ```

- При создании ключа команда попросит ввести любую ключевую фразу для более надёжной защиты вашего пароля. Можно пропустить этот этап, нажав `Enter`.
- Сменить пароль на ключ можно с помощью команды:

```
ssh-keygen -p
```

3. Добавление SSH-ключа в учётную запись GitHub
   - Скопируйте созданный SSH-ключ в буфер обмена командой (для X11):

```
xclip -i < ~/.ssh/id_ed25519.pub
```

   - Скопируйте созданный SSH-ключ в буфер обмена командой (для Wayland):

```
wl-copy < ~/.ssh/id_ed25519.pub
```

   - Откройте настройки своего аккаунта на GitHub и перейдем в раздел `SSH and GPC keys`.
   - Нажмите кнопку `ew SSH key`.
   - Добавьте в поле `Title` название этого ключа, например, `ed25519@hostname`.
   - Вставьте из буфера обмена в поле `Key` ключ.
   - Нажмите кнопку `Add SSH key`.

1.2.1.5 Верификация коммитов

1. Проверка коммитов в Git
   - GitHub и GitLab будут показывать значок *Verified* рядом с вашими новыми коммитами.

   1. Режим бдительности (vigilant mode)
      - На GitHub есть настройка vigilant mode.
      - Все неподписанные коммиты будут явно помечены как *Unverified*.
      - Включается это в настройках в разделе *SSH and GPG keys*. Установите метку на *Flag unsigned commits as unverified*.

2. Верификация коммитов с помощью *PGP*
   - Как настроить PGP-подпись коммитов с помощью `gpg`.

   1. Общая информация
      - Коммиты имеют следующие свойства:
        - author (автор) — контрибьютор, выполнивший работу (указывается для справки);

    - committer (коммитер) — пользователь, который закоммитил изменения.
  - Эти свойства можно переопределить при совершении коммита.
  - Авторство коммита можно подделать.
  - В git есть функция подписи коммитов.
  - Для подписывания коммитов используется технология PGP.
  - Подпись коммита позволяет удостовериться в том, кто является коммитером. Авторство не проверяется.
2. Создание ключа
  - Генерируем ключ

```
gpg --full-generate-key
```

  - Из предложенных опций выбираем:
    - тип *RSA and RSA*;
    - размер 4096;
    - выберите срок действия; значение по умолчанию — 0 (срок действия не истекает никогда).
  - GPG запросит личную информацию, которая сохранится в ключе:
    - Имя (не менее 5 символов).
    - Адрес электронной почты.
      - При вводе email убедитесь, что он соответствует адресу, используемому на GitHub.
    - Комментарий. Можно ввести что угодно или нажать клавишу ввода, чтобы оставить это поле пустым.
3. Экспорт ключа
  - Выводим список ключей и копируем отпечаток приватного ключа:

```
gpg --list-secret-keys --keyid-format LONG
```

  - Отпечаток ключа — это последовательность байтов, используемая для идентификации более длинного, по сравнению с самим отпечатком ключа.
  - Формат строки:

```
sec   Алгоритм/Отпечаток_ключа Дата_создания [Флаги] [Годен_до]
      ID_ключа
```

  - Экспортируем ключ в формате ASCII по его отпечатку:

```
gpg --armor --export <PGP Fingerprint>
```

4. Добавление PGP ключа в GitHub
  - Копируем ключ и добавляем его в настройках профиля на GitHub (или GitLab).
  - Скопируйте ваш сгенерированный PGP ключ в буфер обмена:

```
gpg --armor --export <PGP Fingerprint> | xclip -sel clip
```

- Перейдите в настройки GitHub (https://github.com/settings/keys), нажмите на кнопку *New GPG key* и вставьте полученный ключ в поле ввода.

5. Подписывание коммитов git
   - Подпись коммитов при работе через терминал:

```
git commit -a -S -m 'your commit message'
```

   - Флаг `-S` означает создание подписанного коммита. При этом может потребоваться ввод кодовой фразы, заданной при генерации GPG-ключа.
6. Настройка автоматических подписей коммитов git
   - Используя введёный email, укажите Git применять его при подписи коммитов:

```
git config --global user.signingkey <PGP Fingerprint>
git config --global commit.gpgsign true
git config --global gpg.program $(which gpg)
```

1.2.2 Рабочий процесс Gitflow

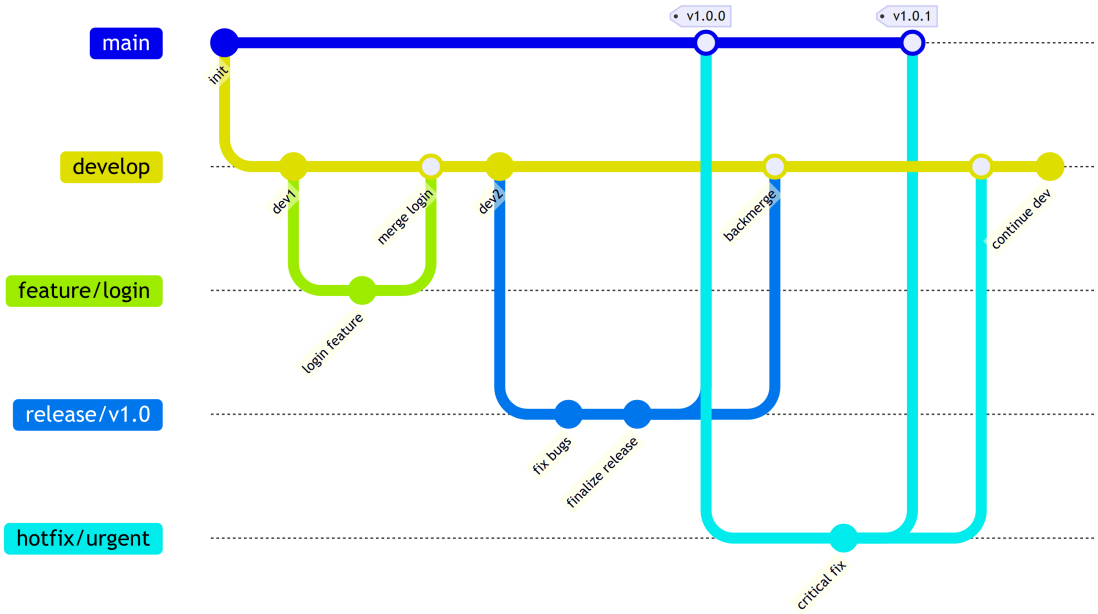
Рабочий процесс *Gitflow Workflow*. Будем описывать его с использованием пакета `git-flow`.

1.2.2.1 Общая информация

- Репозиторий: https://github.com/petervanderdoes/gitflow-avh/
- Описание модели ветвления: https://nvie.com/posts/a-successful-git-branching-model/
- Gitflow Workflow опубликована и популяризована Винсентом Дриссеном из компании vie.
- Gitflow Workflow предполагает выстраивание строгой модели ветвления с учётом выпуска проекта.
- Данная модель отлично подходит для организации рабочего процесса на основе релизов.
- Работа по модели Gitflow включает создание отдельной ветки для исправлений ошибок в рабочей среде.
- Последовательность действий при работе по модели Gitflow:
  - Из ветки `master` создаётся ветка `develop`.
  - Из ветки `develop` создаётся ветка `release`.
  - Из ветки `develop` создаются ветки `feature`.

— Когда работа над веткой `feature` завершена, она сливается с веткой `develop`.
— Когда работа над веткой релиза `release` завершена, она сливается в ветки `develop` и `master`.
— Если в `master` обнаружена проблема, из `master` создаётся ветка `hotfix`.
— Когда работа над веткой исправления `hotfix` завершена, она сливается в ветки `develop` и `master`.

1.2.2.2 Пример



Ключевые моменты:

— Сначала создаются основные ветки (main/master и develop)
— Feature ветки создаются от develop и туда же мержатся
— Release ветки создаются от develop и мержатся в main (с тегом) и обратно в develop
— Hotfix ветки создаются от main и мержатся в main (с тегом) и в develop

1.2.2.3 Установка программного обеспечения

— Для Windows используется пакетный менеджер Chocolatey.
— Git-flow входит в состав пакета git.

```
choco install git
```

- Для MacOS используется пакетный менеджер Homebrew.

```
brew install git-flow
```

- Linux
  - Gentoo

    ```
    emerge dev-vcs/git-flow
    ```

  - Ubuntu

    ```
    apt-get install git-flow
    ```

  - Centos
    - Первоначально нужно установить репозиторий *epel* (https://fedoraproject.org/wiki/EPEL):

      ```
      dnf install epel-release
      ```

    - Затем, собственно, установить git-flow:

      ```
      dnf install gitflow
      ```

  - Fedora
    - Это программное обеспечение удалено из основного репозитория.
    - Можно установить вручную или из коллекции репозиториев *Copr*.
    - Установка из коллекции репозиториев *Copr* (https://copr.fedorainfracloud.org/coprs/elegos/gitflow/):

      ```
      # Enable the copr repository
      dnf copr enable elegos/gitflow
      # Install gitflow
      dnf install gitflow
      ```

    - Установка вручную:

      ```
      cd /tmp
      wget --no-check-certificate -q https://raw.github.com/peterv
       ↪ anderdoes/gitflow/develop/contrib/gitflow-installer.sh
      chmod +x gitflow-installer.sh
      sudo ./gitflow-installer.sh install stable
      ```

1.2.2.4 Поддержка завершения команды

1.2.2.4.1 git-flow-completion

— Репозиторий: https://github.com/bobthecow/git-flow-completion
— Поддержка bash, zsh, fish.

1.2.2.4.2 oh-my-fish/plugin-git-flow

— Репозиторий: https://github.com/oh-my-fish/plugin-git-flow
— Завершение для fish в рамках фреймворка *Oh My Fish*.

1.2.2.5 Процесс работы с Gitflow

1.2.2.5.1 Основные ветки (master) и ветки разработки (develop)

Для фиксации истории проекта в рамках этого процесса вместо одной ветки `master` используются две ветки. В ветке `master` хранится официальная история релиза, а ветка `develop` предназначена для объединения всех функций. Кроме того, для удобства рекомендуется присваивать всем коммитам в ветке `master` номер версии.

При использовании библиотеки расширений `git-flow` нужно инициализировать структуру в существующем репозитории:

```
git flow init
```

Для github параметр `Version tag prefix` следует установить в `v`.

После этого проверьте, на какой ветке Вы находитесь:

```
git branch
```

1.2.2.5.2 Функциональные ветки (feature)

- Под каждую новую функцию должна быть отведена собственная ветка, которую можно отправлять в центральный репозиторий для создания резервной копии или совместной работы команды. Ветки `feature` создаются не на основе `master`, а на основе `develop`. Когда работа над функцией завершается, соответствующая ветка сливается обратно с веткой `develop`. Функции не следует отправлять напрямую в ветку `master`.
- Как правило, ветки `feature` создаются на основе последней ветки `develop`.

1. Создание функциональной ветки
   Создадим новую функциональную ветку:

```
git flow feature start feature_branch
```

   - Далее работаем как обычно.
2. Окончание работы с функциональной веткой
   - По завершении работы над функцией следует объединить ветку `feature_branch` с `develop`:

```
git flow feature finish feature_branch
```

1.2.2.5.3 Ветки выпуска (release)

- Когда в ветке `develop` оказывается достаточно функций для выпуска, из ветки develop создаётся ветка `release`. Создание этой ветки запускает следующий цикл выпуска, и с этого момента новые функции добавить больше нельзя — допускается лишь отладка, создание документации и решение других задач. Когда подготовка релиза завершается, ветка `release` сливается с `master` и ей присваивается номер версии. После нужно выполнить слияние с веткой `develop`, в которой с момента создания ветки релиза могли возникнуть изменения.
- Благодаря тому, что для подготовки выпусков используется специальная ветка, одна команда может дорабатывать текущий выпуск, в то время как другая команда продолжает работу над функциями для следующего.
- Создать новую ветку release можно с помощью следующей команды:

```
git flow release start 1.0.0
```

- Для завершения работы на ветке `release` используются следующие команды:

```
git flow release finish 1.0.0
```

1.2.2.5.4 Ветки исправления (hotfix)

- Ветки поддержки или ветки `hotfix` используются для быстрого внесения исправлений в рабочие релизы. Они создаются от ветки `master`. Это единственная ветка, которая должна быть создана непосредственно от master. Как только исправление завершено, ветку следует объединить с `master` и `develop`. Ветка `master` должна быть помечена обновленным номером версии.
- Наличие специальной ветки для исправления ошибок позволяет команде решать проблемы, не прерывая остальную часть рабочего процесса и не ожидая следующего цикла релиза.
- Ветку `hotfix` можно создать с помощью следующих команд:

```
git flow hotfix start hotfix_branch
```

- По завершении работы ветка `hotfix` объединяется с `master` и `develop`:

```
git flow hotfix finish hotfix_branch
```

1.3 Соглашение об именовании Denote

- Приведены принципы именования Denote.
- Для справки приведена базовая схема именования Denote.
- Далее приведена схема именования, применяемая при создании каталогов и имён файлов.

1.3.1 Базовая схема именования

- Каталог с файлами содержит просто список файлов, желательно избегать подкаталоги.
- Шаблон имени файла:

  `DATE==SIGNATURE--TITLE__KEYWORDS.EXTENSION`

- Поле `DATE` : дата в формате год-месяц-день, за которой следует с большой буквы `T` (для «времени») текущее время в обозначение час-минута-секунда.
  - Примерно так: `20220531T091625`.
  - `DATE` служит уникальным идентификатором каждого файла.
- Поле `SIGNATURE` : строка буквенно-цифровых символов.
  - Не имеют чётко определённой цели.
  - Определяется пользователем.
  - Варианты заполнения:
    - Разделение видео на части : `part1` и `part2`.
    - Приоритет : `a`, `b`, `c`.
    - Для установления последовательные отношения между файлами (например, 1, 1a, 1b, 1b1, 1b2, ...).
- Поле `TITLE` : заголовок заметки, предоставленный пользователем.
  - По умолчанию он автоматически записывается строчными буквами, через дефис.
- Поле `KEYWORDS` состоит из одной или нескольких записей, разделённых значком подчёркивание.
  - Каждое ключевое слово строка, предоставленная пользователем в соответствующем приглашении, которая в целом описывает содержание записи.
  - Каждое из ключевых слов представляет собой одно слово.
- `EXTENSION` : тип файла.
- Примеры:

```
20220610T043241--initial-thoughts-on-the-zettelkasten-
method__notetaking.org
20220610T062201--define-custom-org-hyperlink-type__denote_emacs_package.md
20220610T162327--on-hierarchy-and-taxis__notetaking_philosophy.txt
```

- Различные разделители полей, а именно `--` и `__` дают эффективный способ поиска.
- Можно использовать следующие перестановки частей имён файлов:

```
DATE.EXT
DATE--TITLE.EXT
DATE__KEYWORDS.EXT
DATE==SIGNATURE.EXT
DATE==SIGNATURE--TITLE.EXT
DATE==SIGNATURE--TITLE__KEYWORDS.EXT
DATE==SIGNATURE__KEYWORDS.EXT
```

- По умолчанию имена файлов имеют три поля и два набора разделителей между ними:

```
DATE--TITLE__KEYWORDS.EXTENSION
```

— Когда подпись присутствует:

  `DATE==SIGNATURE--TITLE__KEYWORDS.EXTENSION`

1.3.2 Концептуальные вариации применения

— Идеология Denote совместима с разными методологиями.

Концепция	Как применяется в Denote
Плоское хранение (Flat-file)	Все файлы можно хранить в одной папке, а для навигации полагаться на уникальные имена и ключевые слова. Это основной и рекомендуемый подход.
Zettelkasten и Folgezettel	Поле `SIGNATURE` подходит для создания сложных цепочек заметок (например, `20231001==1a`, `20231002==1a1`, `20231003==1a2`).
Проектное управление	Ключевые слова можно использовать как теги проектов (например, `__project_alpha`), а в заголовке указывать конкретную задачу.
Мультимедийный архив	Схему можно применять не только к текстовым заметкам, но и к любым другим файлам. Имя файла становится универсальным описательным якорем.

— При создании вариаций стоит придерживаться двух ключевых принципов Denote, которые обеспечивают её эффективность:
  — Уникальность: идентификатор на основе timestamp (`DATE`) должен оставаться неизменным якорем файла.
  — Машиночитаемость: чёткие разделители (`--`, `__`, `==`) нужно сохранять, чтобы система и скрипты могли легко парсить имена.
— Denote есть каркас, а не жёсткий стандарт.
— Вы можете убрать необязательные поля, изменить порядок или использовать подпись для своих целей, сохраняя при этом главные преимущества системы — понятность, уникальность и независимость от инструментов.

1.3.3 Применяемая схема

1.3.3.1 Основная идея

— Использовать каталоги для грубой семантической и хронологической группировки.

- Полное имя файла Denote — универсальный, уникальный идентификатор и описание.
- Идея категорий взята из Johnny Decimal.
- Плоская и гибкая категоризация: файл может принадлежать к нескольким равнозначным категориям без жёсткой иерархии.
- Машиночитаемость: знак `=` есть уникальный разделитель, который легко выделить скриптом или настроить поиск.
- Визуальная ясность: категории чётко сгруппированы в начале имени между `==` и `--`

1.3.3.2 Варианты меток даты для поля `DATE`

- Вы можете выбирать формат даты в зависимости от смысла содержимого, сохраняя сортировку.

Формат даты в имени файла	Когда использовать	Пример имени файла
`2026` (год)	Для документов, тематика которых актуальна целый год: годовые отчёты, планы, резюме.	`2026--годовой-отчет__финансы.pdf`
`2026-1` (полугодие)	Для целей, планов или результатов, сгруппированных по полугодиям.	`2026-1--планы-на-семестр__стратегия.md`
`2026-02-06` (конкретная дата)	Для заметок, событий и документов, привязанных к конкретному дню (встречи, дневниковые записи).	`2026-02-06--встреча-с-командой__проект_митинг.org`

1.3.3.3 Поле SIGNATURE

- В поле `SIGNATURE` (после `==`) указывается набор равноправных категорий, разделённых знаком `=`.
- Это создаёт плоский, но структурированный список категорий прямо в имени файла.

1.3.3.4 Общий формат

```
ДАТА==КАТЕГОРИЯ1=КАТЕГОРИЯ2=КАТЕГОРИЯ3--ЗАГОЛОВОК__КЛЮЧЕВЫЕ_СЛОВА
```

- Примеры:
  - `2026-02-10==проект=вебсайт=дизайн--обзор-макетов__ui_клиент_встреча.fig`
  - `2026-1==сфера=здоровье=питание--полезные-рецепты__еда_идеи.md`
  - `2025-12-24==архив=путешествия=япония=фото--поездка-в-токио__отпуск.jpg`

1.4 Рабочее пространство лабораторной работы

При выполнении лабораторной работы следует придерживаться структуры рабочего пространства.

1.4.1 Основные идеи

- Стандартные соглашения об именах
- Стандартное соглашение для путей к файлам
- Стандартная настройка курса внутри шаблона курса

1.4.2 Используемые стандарты и программные продукты

- Стандарт Git Flow (раздел 1.2.2).
- Стандарт Семантическое версионирование (раздел 1.1.1).
- Стандарт Общепринятые коммиты (раздел 1.1.2).

1.5 Литературное программирование

Литературное (грамотное) программирование — это подход, приоритизирующий понятность программы для человека, а не её исполнение компьютером. В экосистеме Julia он реализуется через несколько инструментов. Предлагается использовать `Literate.jl`.

1.5.1 Суть литературного программирования

Идея, предложенная Дональдом Кнутом [1—3], меняет традиционный взгляд: программа создаётся не как последовательность инструкций для машины, а как ли-

тературное эссе, объясняющее логику решения задачи.

Литературное программирование подходит не всем и не для всех задач, но может стать мощным инструментом для создания сложных, ясных и поддерживаемых программ.

- Идеи, а не синтаксис.
  - Автор представляет программу как сеть взаимосвязанных идей, располагая фрагменты кода в логическом для человеческого понимания порядке, а не в диктуемом языком программирования.
- Два процесса. Для работы с таким исходником нужны две утилиты:
  - `weave` — «плетёт» (weave) документ для чтения (PDF, HTML).
  - `tangle` — «запутывает» (tangle) из фрагментов исполняемый исходный код.
- Не просто комментарии.
  - Это не просто подробные комментарии (вроде JavaDoc). Это методология проектирования, где документация первична, а код в неё встраивается.

1.5.2 Реализация в Julia: основные инструменты

В Julia есть несколько пакетов для литературного программирования:

Инструмент	Ключевая задача	Основные форматы вывода	Особенность
Literate.jl	Преобразование скриптов `.jl` в документы.	Markdown, Jupyter Notebook (`.ipynb`), скрипт `.jl`.	Простая интеграция с Quatro для документации пакетов.
Weave.jl	Генерация научных отчётов с исполняемым кодом.	LaTeX/PDF, HTML, Jupyter Notebook, MS Word.	Богатые возможности оформления научных публикаций.
Jupyter / IJulia	Интерактивные вычисления и прототипирование.	`.ipynb` (Jupyter Notebook).	Интерактивность, но менее пригоден для автоматизации.

- `Literate.jl` отлично подходит для автоматизации создания документации, а также для написания статей и учебников прямо в `.jl`-файлах.
- `Weave.jl` больше заточен под академические отчёты и публикации, требующие строгого форматирования (LaTeX/PDF).
- `Jupyter` лучше подходит для интерактивного исследования данных, но управление версиями `.ipynb`-файлов и их автоматизация сложнее.

`Literate.jl` — наиболее универсальный и простой для старта инструмент. Его базовый синтаксис интуитивен: - Строки, начинающиеся с #, интерпретируются как Markdown. - Все остальные строки — это код на Julia. - Пакет умеет выполнять код и вставлять результаты (текст, графики) в итоговый документ.

1.5.3 Пример с Literate.jl

Допустим, у вас есть файл `linear_regression.jl`:

```
# # Простая линейная регрессия
# Сгенерируем синтетические данные и подберём модель.
using LinearAlgebra, Plots

# ## Генерация данных
n = 100
x = rand(n)
# Целевая переменная с шумом:
y = 2.5 * x .+ 1.2 .+ 0.3 * randn(n)

# scatter(x, y, label="данные", legend=:topleft)

# ## Решение методом наименьших квадратов
# Создаём матрицу плана с колонкой единиц:
X = [ones(n) x]
β = X \ y  # Находим коэффициенты
println("Найденные коэффициенты: β₀ = $(round(β[1], digits=3)), β₁ =
 ↪  $(round(β[2], digits=3))")
```

Чтобы преобразовать это в документ, выполните в Julia:

```
using Literate
# Создать Markdown-файл в папке `docs`
Literate.markdown("linear_regression.jl", "docs")
# Или сразу создать Jupyter Notebook
Literate.notebook("linear_regression.jl", "docs")
```

1.6 Julia. Пакет DrWatson

1.6.1 Общая информация

— DrWatson.jl — фреймворк для организации научных проектов в Julia.
— Репозиторий: https://github.com/JuliaDynamics/DrWatson.jl
— Документация: https://juliadynamics.github.io/DrWatson.jl

1.6.2 Установка и инициализация

— Установка в REPL:

```julia
# Установка
using Pkg
Pkg.add("DrWatson")

# Создание проекта
using DrWatson
initialize_project("MyProject"; authors=["Ваше Имя"], git=true)
```

— Создаваемая структура:

```
MyProject/
├── Project.toml
├── Manifest.toml
├── data/
│   ├── sims/       # Результаты симуляций
│   ├── exp_raw/    # Сырые экспериментальные данные
│   └── exp_pro/    Обработанные данные
├── scripts/        # Скрипты для анализа
├── src/            # Исходный код проекта
└── _research/      # Черновики и временные файлы
```

— Активирует окружение проекта, загружает зависимости:

```julia
@quickactivate :MyProject  # Автоматически находит Project.toml
```

1.6.3 Пути

— Стандартные пути:

```
datadir()        # → "data"
plotsdir()       # → "plots"
projectdir()     # → Корень проекта
scriptdir("analysis.jl") → "scripts/analysis.jl"
```

1.6.4 Сохранение результатов

```
using DrWatson: savename, @dict, safesave

params = @dict(α=0.1, β=2.0)  # Автоматическое именование
results = Dict(:data ⇒ rand(10))

## Сохранение с уникальным именем
filename = savename(params, "jld2")  # → "α=0.1_β=2.0.jld2"
safesave(datadir("sims", filename), results)
```

1.6.5 Загрузка данных

```
using DrWatson: load, @strdict

## Поиск файлов по параметрам
filter = @strdict(α=0.1)
file = getlatest(filter, datadir("sims"))  # Последний файл с α=0.1
data = load(file)["data"]
```

1.6.6 Воспроизводимость

```
gitdescribe()  # Возвращает текущий коммит Git → "v0.1.0-5-gf7a2d"
tag!(filename) # Добавляет хеш коммита к имени файла
```

1.6.7 Пакетная обработка

```julia
using BSON

## Генерация параметров
all_params = Dict(:α => [0.1, 0.5], :β => 1:5)

## Запуск симуляций
for params in dict_list(all_params)
    result = simulate(params)
    safesave(datadir("sims", savename(params, "bson")), result)
end
```

1.6.8 Анализ результатов

```julia
using DataFrames

## Сбор всех данных в DataFrame
df = collect_results(datadir("sims"))
select!(df, :α, :β, :result)
```

1.6.9 Интеграция с Literate.jl

```julia
## scripts/generate_report.jl
using Literate

Literate.markdown("scripts/analysis.jl", "docs"; flavor =
 ↪  Literate.QuartoFlavor())
```

1.6.10 Makefile

— Сделаем Makefile:

```makefile
.PHONY: report clean
```

```
report:
    julia --project=. scripts/run_simulations.jl
    julia --project=. scripts/generate_report.jl
    quarto render docs/report.qmd

clean:
    rm -rf data/sims/*
    rm -rf plots/*
```

1.6.11 Практики

- Структура кода
  - Храните многократно используемые функции в `src/` как модуль:

```julia
# src/MyProject.jl
module MyProject
export simulate

function simulate(params)
  # ...
end
end
```

- Версионирование данных
  - Используйте Git LFS для больших файлов в `data/`
- Шаблоны проектов
  - Создайте шаблон через PkgTemplates.jl:

```julia
using PkgTemplates
t = Template(;
  dir="~/Projects",
  julia=v"1.9",
  plugins=[DrWatsonPlugin()]
)
t("MyNewProject")
```

- Отладка
  - Используйте `produce_or_load` для кэширования результатов:

```julia
produce_or_load(params, datadir("cache")) do p
  compute_expensive_operation(p)
```

```
end
```

1.6.12 Пример рабочего процесса

- `initialize_project()` → Создает структуру
- `quickactivate()` → Активирует окружение
- Редактируем скрипты в `scripts/`
- `savename()` + `safesave()` → Сохраняем результаты
- `collect_results()` → Анализируем данные
- Literate.jl → Генерируем отчёт
- Quarto → Рендерим финальный документ

1.7 Язык Markdown

1.7.1 Синтаксис языка Markdown

Общая информация по языку Markdown.

1.7.1.1 Базовый синтаксис Markdown

- Чтобы создать заголовок, используйте знак (#), например:

```
### This is heading 1
#### This is heading 2
##### This is heading 3
###### This is heading 4
```

- Чтобы задать для текста полужирное начертание, заключите его в двойные звездочки:

```
This text is **bold**.
```

This text is **bold**.

— Чтобы задать для текста курсивное начертание, заключите его в одинарные звездочки:

```
This text is *italic*.
```

This text is *italic*.

— Чтобы задать для текста полужирное и курсивное начертание, заключите его в тройные звездочки:

```
This is text is both ***bold and italic***.
```

This is text is both ***bold and italic***.

Блоки цитирования создаются с помощью символа >:

```
> The drought had lasted now for ten million years, and the reign of
 ↪ the terrible lizards had long since ended. Here on the Equator,
 ↪ in the continent which would one day be known as Africa, the
 ↪ battle for existence had reached a new climax of ferocity, and
 ↪ the victor was not yet in sight. In this barren and desiccated
 ↪ land, only the small or the swift or the fierce could flourish,
 ↪ or even hope to survive.
```

Неупорядоченный (маркированный) список можно отформатировать с помощью звездочек или тире:

```
- List item 1
- List item 2
- List item 3
```

Чтобы вложить один список в другой, добавьте отступ для элементов дочернего списка:

```
- List item 1
  - List item A
```

```
  - List item B
- List item 2
```

Упорядоченный список можно отформатировать с помощью соответствующих цифр:

```
1. First instruction
1. Second instruction
1. Third instruction
```

Чтобы вложить один список в другой, добавьте отступ для элементов дочернего списка:

```
1. First instruction
   1. Sub-instruction
   1. Sub-instruction
1. Second instruction
```

Синтаксис Markdown для встроенной ссылки состоит из части `[link text]`, представляющей текст гиперссылки, и части `(file-name.md)` – URL-адреса или имени файла, на который дается ссылка:

```
[link text](file-name.md)
```

Markdown поддерживает как встраивание фрагментов кода в предложение, так и их размещение между предложениями в виде отдельных огражденных блоков. Огражденные блоки кода — это простой способ выделить синтаксис для фрагментов кода. Общий формат огражденных блоков кода:

```
''' language
your code goes in here
'''
```

Верхние и нижние индексы:

H_2O

записывается как

```
H~2~O
```

2^{10}

записывается как

```
2^10^
```

1.7.2 Математика в Markdown

Markdown сам по себе не обрабатывает формулы, но он легко интегрируется с системой вёрстки математических выражений LaTeX.

1.7.2.1 Основные принципы

Формулы записываются с помощью синтаксиса LaTeX внутри специальных разделителей:

- `$ ... $` — для формул внутри текста (инлайновых).
  - Пример: `Скорость вычисляется по формуле $v = \frac{s}{t}$.`
  - Результат: Скорость вычисляется по формуле $v = \frac{s}{t}$.
- `$$ ... $$` — для выключенных формул (блочных), которые отображаются на отдельной строке, по центру.
  - Пример:

    ```
    Второй закон Ньютона:
    $$ \vec{F} = m \vec{a} $$
    ```

  - Результат: Второй закон Ньютона:

$$\vec{F} = m\vec{a}$$

Внутритекстовые формулы делаются аналогично формулам LaTeX. Например, формула $\sin^2(x) + \cos^2(x) = 1$ запишется как

```
$\sin^2 (x) + \cos^2 (x) = 1$
```

Выключные формулы:

$$\sin^2(x) + \cos^2(x) = 1 \tag{1.1}$$

со ссылкой в тексте «Смотри формулу (1.1).» записывается как

```
$$
\sin^2 (x) + \cos^2 (x) = 1
$$ {#eq-sin2-cos2}

Смотри формулу (-@eq-sin2-cos2).
```

1.7.2.2 Основные элементы синтаксиса

Элемент	Синтаксис LaTeX	Пример (внутри $ или $$)	Результат
Индексы	`x^2`, `a_n`	`$E = mc^2$`, `$x_{i+1}$`	$E = mc^2$, x_{i+1}
Дроби	`\frac{a}{b}`	`$\frac{\Delta y}{\Delta x}$`	$\frac{\Delta y}{\Delta x}$
Корни	`\sqrt{x}`, `\sqrt[n]{x}`	`$\sqrt{2}`, `$\sqrt[3]{8}$`	$\sqrt{2}$, $\sqrt[3]{8}$
Греческие буквы	`\alpha`, `\beta`, `\Omega`, `\Gamma`	`$\alpha, \beta, \gamma, \pi$`	$\alpha, \beta, \gamma, \pi$
Скобки (авторазмер)	`\left( ... \right)`	`$\left( \frac{1}{x} \right)$`	$\left(\frac{1}{x}\right)$
Интегралы, суммы	`\int`, `\sum`, `\prod`	`$\int_a^b f(x)dx$`, `$\sum_{i=1}^n i$`	$\int_a^b f(x)dx$, $\sum_{i=1}^n i$
Пределы	`\lim_{x \to a}`	`$\lim_{x \to 0} \frac{\sin x}{x}$`	$\lim_{x\to 0} \frac{\sin x}{x}$
Векторы	`\vec{a}`, `\overline{AB}`	`$\vec{F}`, `$\overline{v}$`	$\vec{F}$, $\overline{v}$
Матрицы	`\begin{matrix} ... \end{matrix}` (см. ниже)		

1.7.2.3 3. Примеры сложных конструкций

— Матрица (в блочном режиме $$):

```
$$
A = \begin{pmatrix}
a_{11} & a_{12} \\
a_{21} & a_{22}
\end{pmatrix}
$$
```

$$
A = \begin{pmatrix}
a_{11} & a_{12} \\
a_{21} & a_{22}
\end{pmatrix}
$$

— Система уравнений:

```
$$
\begin{cases}
x + 2y = 5 \\
3x - y = 1
\end{cases}
$$
```

$$
\begin{cases}
x + 2y = 5 \\
3x - y = 1
\end{cases}
$$

— Определённый интеграл с пределом:

```
$$
I = \int_0^{\pi} \sin^2 x \, dx = \frac{\pi}{2}
$$
```

$$
I = \int_0^{\pi} \sin^2 x \, dx = \frac{\pi}{2}
$$

Формула с производной:

```
$$
f'(x) = \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x}
$$
```

$$
f'(x) = \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x}
$$

1.7.2.4 Практические советы

- Стиль и читаемость: используйте блочные формулы ($$) для важных, ключевых уравнений. Инлайновые ($) — для упоминаний в тексте.
- Комментарии в коде: В сыром Markdown-файле рядом со сложной формулой можно ставить комментарий:

```
$$ \Delta = b^2 - 4ac $$ <!-- Дискриминант квадратного уравнения -->
```

- Пробелы и выравнивание: Внутри формул можно использовать `\,` (тонкую шпацию), `\:` (среднюю шпацию), `\;` (большую шпацию), `\quad` (квадратную шпацию), `\qquad` (двойную квадратную шпацию) для ручной расстановки отступов, чтобы формулы выглядели аккуратнее.

1.7.3 Язык markdown. Вариант Quarto

1.7.3.1 Общая информация

- Quarto основан на Pandoc и использует его вариацию markdown в качестве базового синтаксиса документа.
- Pandoc markdown — это расширенная и слегка переработанная версия синтаксиса Markdown Джона Грубера.

1.7.3.2 Форматирование текста

- Markdown:

```
*курсив*, **полужирный**, ***полужирный курсив***
```

— *курсив* , **полужирный** , ***полужирный курсив***
— Markdown:

```
верхний индекс^2^ нижний индекс~2~
```

— верхний индекс2 / нижний индекс2
— Markdown:

```
~~зачеркивание~~
```

— ~~зачеркивание~~
— Markdown:

```
`verbatim code`
```

— `verbatim code`

1.7.3.3 Заголовки

— Markdown:

```
# Заголовок 1

## Заголовок 2

### Заголовок 3

#### Заголовок 4

##### Заголовок 5

###### Заголовок 6
```

1.7.3.4 Ссылки и изображения

— Markdown:

```
<https://quarto.org>

[Quarto](https://quarto.org)

![Caption](elephant.png)

[![Caption](elephant.png)](https://quarto.org)

[![Caption](elephant.png "An elephant")](https://quarto.org)

[![](elephant.png){fig-alt="Alt text"}](https://quarto.org)
```

1.7.3.5 Списки

- Списки в Quarto требуют целой пустой строки над списком.
- В противном случае список не будет отображаться в виде списка, а будет отображаться как обычный текст вдоль одной строки.
- Markdown:

```
* unordered list
    + sub-item 1
    + sub-item 2
        - sub-sub-item 1
```

- Вывод:
  - unordered list
    - sub-item 1
    - sub-item 2
      - sub-sub-item 1
- Markdown:

```
*   item 2

    Continued (indent 4 spaces)
```

- Вывод:
  - item 2
    Continued (indent 4 spaces)
- Markdown:

```
1. ordered list
2. item 2
    i) sub-item 1
         A.  sub-sub-item 1
```

— Вывод:

1. ordered list
2. item 2
    1. sub-item 1
        1. sub-sub-item 1

— Markdown:

```
- [ ] Task 1
- [x] Task 2
```

— Вывод:

☐ Task 1
☒ Task 2

1.7.3.6 Сноски

— Поддерживается нумерация и форматирование сносок[1]:

```
Here is a footnote reference,[^1] and another.[^longnote]

[^1]: Here is the footnote.

[^longnote]: Here's one with multiple blocks.

    Subsequent paragraphs are indented to show that they
belong to the previous footnote.

        { some.code }

    The whole paragraph can be indented, or just the first
    line.  In this way, multi-paragraph footnotes work like
    multi-paragraph list items.
```

[1] Вот такая вот сноска.

```
This paragraph won't be part of the note, because it
isn't indented.
```

— Можно писать сноски в виде отдельных абзацев:

```
Here is an inline note.^[Inlines notes are easier to write, since
↪ you don't have to pick an identifier and move down to type the
↪ note.]
```

— Идентификаторы сносок должны быть уникальными в пределах документа.
— В книгах Quarto главы объединяются в один документ для определённых форматов (включая PDF, DOCX и EPUB), поэтому идентификаторы сносок должны быть уникальными для всех глав.

1.7.3.7 Таблицы

— Markdown:

```
| Right | Left | Default | Center |
|------:|:-----|---------|:------:|
|   12  |  12  |    12   |    12  |
|  123  |  123 |   123   |   123  |
|    1  |    1 |     1   |     1  |
```

— Вывод:

Right	Left	Default	Center
12	12	12	12
123	123	123	123
1	1	1	1

1.7.3.8 Исходный код

— Следует использовать ““ ““ для разграничения блоков исходного кода:

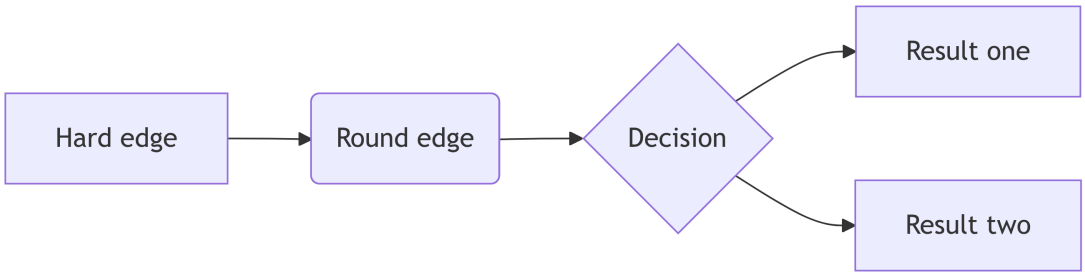
```
```
code
```
```

— Можно указать язык для подсветки синтаксиса блоков кода:

```
```python
1 + 1
```
```

— Если ваш язык не поддерживается, вы можете использовать язык `default`, чтобы получить оформление по умолчанию:

```
```default
code
```
```

— Эквивалентом краткой формы является длинная форма, которая использует язык как класс (например, `.python`) внутри фигурных скобок:

```
```{.python}
1 + 1
```
```

— Длинная форма позволяет добавлять атрибуты к блоку:

```
```{.python filename="run.py"}
code
```
```

1.7.3.9 Аннотации кода

— Аннотации ячеек кода включают в себя:
  — Завершение каждой аннотированной строки комментарием и номером аннотации в угловых скобках (например, `# <2>`). Используйте одно и то же число, чтобы охватить аннотацию на нескольких строках.
  — Добавление упорядоченного списка сразу после ячейки кода, где каждый элемент соответствует номеру аннотации в коде.

```
```r
x <- 1:10  # <1>
y <- x^2   # <2>
z <- x^3   # <2>
```
```

```
1. Create a vector `x`, and then,
2. compute `y` and `z`
```

— Возможные позиции аннотаций для вывода HTML:
  — `below` (по умолчанию): По умолчанию аннотации отображаются под ячейкой кода.
  — `hover` : Аннотации отображаются при наведении мыши на маркер строки.
  — `select` : Аннотации появляются при нажатии на маркер и могут быть удалены повторным нажатием.
— Чтобы изменить положение аннотации по умолчанию, используйте поле метаданных `code-annotations` в заголовке документа:

```
---
code-annotations: hover
---
```

1.7.3.10 Необработанный контент

— Необработанный контент может быть включён напрямую без разбора Quarto, используя атрибут `raw`:

```
```{=html}
<iframe src="https://quarto.org/" width="500"
 ↪  height="400"></iframe>
```
```

— Для вывода PDF используйте необработанный блок LaTeX:

```
```{=latex}
\renewcommand*{\labelitemi}{\textgreater}
```
```

— Можно включить необработанный контент в строку:

```
Here's some raw inline HTML: `<a>html</a>`{=html}
```

1.7.3.11 Уравнения

— Используйте разделители $ для строчной математики и разделители $$ выделенной математики.
— Markdown:

```
inline math: $E = mc^{2}$
```

— Вывод: inline math: $E = mc^2$
— Markdown:

```
display math:

$$E = mc^{2}$$
```

— Вывод: display math:

$$E = mc^2$$

— Если вы хотите определить пользовательские макросы TeX, включите их в разделители $$, заключенные в блок `.hidden`:

```
::: {.hidden}
$$
 \def\RR{{\bf R}}
 \def\bold#1{{\bf #1}}
$$
:::
```

Для MathJax (применяется по умолчанию) можно использовать команды `\def`, `\newcommand`, `\renewcommand`, `\newenvironment`, `\renewenvironment`, `\let`.

1.7.3.12 Диаграммы

— Quarto имеет встроенную поддержку для встраивания диаграммы Mermaid и Graphviz:

```
```{mermaid}
```
```

```
flowchart LR
    A[Hard edge] --> B(Round edge)
    B --> C{Decision}
    C --> D[Result one]
    C --> E[Result two]
```



1.7.3.13 Разрывы страниц

— Шорткод `pagebreak` позволяет вставлять в документ собственный разрыв страницы (например, в LaTeX это будет `\newpage`, в MS Word — собственный разрыв страницы docx, в HTML — директива CSS `page-break-after: always`).
— Собственные разрывы страниц поддерживаются для HTML, LaTeX, Context, MS Word, Open Document, ePub (для других форматов используется символ перевода страницы).

```
page 1

{{< pagebreak >}}

page 2
```

1.7.3.14 Порядок атрибутов

— `div` и `span` в pandoc могут иметь любую комбинацию идентификаторов, классов и атрибутов ключ-значение.
— Чтобы pandoc распознал их, они должны быть предоставлены в определённом порядке: идентификаторы, классы, а затем атрибуты ключ-значение.
— Любой из них может быть опущен, но должен следовать этому порядку, если они предоставлены.

— Например, следующее является допустимым:

```
[This is good]{#id .class key1="val1" key2="val2"}
```

— Следующее pandoc не распознает:

```
[This does *not* work!]{.class key="val" #id}
```

1.7.3.15 Сочетания клавиш

— Шорткод `kbd` можно использовать для описания сочетаний клавиш в документации. В форматах Javascript он попытается определить операционную систему формата и отобразит правильное сочетание клавиш. В форматах печати он выведет информацию о сочетаниях клавиш для всех операционных систем.

```
To print, press {{< kbd Shift-Ctrl-P >}}.
To open an existing new project, press {{< kbd mac=Shift-Command-O
↪  win=Shift-Control-O linux=Shift-Ctrl-L >}}.
```

To print, press `Shift-Ctrl-P`. To open an existing new project, press `Shift-Ctrl-L` (linux), `Shift-Command-O` (mac), `Shift-Control-O` (windows).

1.8 Лабораторная работа

1.8.1 Настройка git

1.8.1.1 Установка git

— Установим *git*:

```
sudo dnf -y install git
```

1.8.1.2 Базовая настройка git

— Зададим имя и email владельца репозитория:

```
git config --global user.name "Name Surname"
git config --global user.email "work@mail"
```

— Настроим utf-8 в выводе сообщений git:

```
git config --global core.quotepath false
```

— Настройте верификацию и подписание коммитов git.
— Зададим имя начальной ветки (будем называть её `master`):

```
git config --global init.defaultBranch master
```

— Параметр `autocrlf`:

```
git config --global core.autocrlf input
```

— Параметр `safecrlf`:

```
git config --global core.safecrlf warn
```

1.8.1.3 Настройка gitverse

— Создайте учётную запись на https://gitverse.ru.
— Заполните основные данные на https://gitverse.ru.

1.8.1.4 Настройка sourcecraft

— SourceCraft есть платформа для разработки от Yandex B2B Tech.
— Для регистрации требуется учётная запись Yandex ID.
— Рекомендуется использовать не корпоративную, а собственную учётную запись.
— Создайте учётную запись на https://sourcecraft.dev/.
— Заполните основные данные на https://sourcecraft.dev/.

1.8.1.5 Создайте ключ SSH

— Создадим ключ для доступа к серверу git:

```
ssh-keygen -t ed25519 -f ~/.ssh/id_ed25519-git -C
↪ "your_email@example.com"
```

— Добавляем ключ в агент:

```
ssh-add ~/.ssh/id_ed25519-git
```

— Добавьте ключ в учётную запись gitverse через web-интерфейс (https://gitverse.ru/settings/keys).
— Добавьте ключ в учётную запись sourcecraft через web-интерфейс (https://sourcecraft.dev/me/ssh).

1.8.1.6 Создайте ключи *pgp*

— Генерируем ключ

```
gpg --full-generate-key
```

— Из предложенных опций выбираем:
  — тип *RSA and RSA*;
  — размер 4096;
  — выберите срок действия; значение по умолчанию — 0 (срок действия не истекает никогда).
— GPG запросит личную информацию, которая сохранится в ключе:
  — Имя (не менее 5 символов).
  — Адрес электронной почты.
    — При вводе email убедитесь, что он соответствует адресу, используемому на GtVerse и Sourcecraft.
  — Комментарий. Можно ввести что угодно или нажать клавишу ввода, чтобы оставить это поле пустым.

1.8.1.7 Добавление PGP ключа на хостинг

— Выводим список ключей и копируем отпечаток приватного ключа:

```
gpg --list-secret-keys --keyid-format LONG
```

— Отпечаток ключа — это последовательность байтов, используемая для идентификации более длинного, по сравнению с самим отпечатком ключа.
— Формат строки:

```
sec   Алгоритм/Отпечаток_ключа Дата_создания [Флаги] [Годен_до]
      ID_ключа
```

— Скопируйте ваш сгенерированный PGP ключ в буфер обмена:

```
gpg --armor --export <PGP Fingerprint> | wl-copy
```

— Перейдите в настройки Gitverse (https://gitverse.ru/settings/keys), нажмите на кнопку *Добавить GPG* и вставьте полученный ключ в поле ввода.
— Перейдите в настройки Sourcecraft (https://sourcecraft.dev/me/gpg), нажмите на кнопку *Новый GPG ключ* и вставьте полученный ключ в поле ввода.

1.8.1.8 Настройка автоматических подписей коммитов git

— Используя введёный email, укажите Git применять его при подписи коммитов:

```
git config --global user.signingkey <PGP Fingerprint>
git config --global commit.gpgsign true
git config --global gpg.program $(which gpg)
```

1.8.2 Рабочее пространство лабораторной работы

1.8.2.1 Дополнительное программное обеспечение

1.8.2.1.1 Средства разработки

1. Fedora
   — Установите средства разработки:

```
sudo dnf -y group install development-tools
```

1.8.2.1.2 Quarto

1. Установка
   1. Windows
      - Chocolatey:

        ```
        choco install quarto
        ```

   2. Linux
      2. Gentoo
         - Gentoo, репозиторий karma:

           ```
           emerge quarto
           ```

      3. Arch
         - Arch linux:

           ```
           pacman -S quarto-cli-bin
           ```

         - Manjaro linux:

           ```
           pamac install quarto-cli-bin
           ```

      4. Fedora
         - Установка из CORP:

           ```
           sudo dnf -y copr enable iucar/rstudio
           sudo dnf -y install quarto
           sudo dnf -y install libxcrypt-compat
           ```

1.8.2.1.3 Общепринятые коммиты

1. Установка Node.js
   - На Node.js базируется программное обеспечение для семантического версионирования и общепринятых коммитов.
   - Для управления пакетами лучше использовать `pnpm`, но можно и `yarn`.
   - Gentoo
     - Node.js:

       ```
       emerge nodejs
       emerge yarn
       ```

     - pnpm ставим из оверлея `guru`:

```
eselect repository enable guru
emerge --sync guru
emerge pnpm-bin
```

- Ubuntu

```
apt-get install nodejs
apt-get install yarn
apt-get install pnpm
```

- Fedora

```
sudo dnf -y install nodejs
sudo dnf -y install yarn pnpm
```

- Windows
  - Chocolatey:

```
choco install nodejs
choco install yarn
choco install pnpm
```

- MacOS

```
brew install node
```

2. Настройка Node.js
   Для работы с Node.js добавим каталог с исполняемыми файлами, устанавливаемыми пакетным менеджером, в переменную `PATH`.
   - Linux
     - `pnpm`
       - Запустите:

```
pnpm setup
```

       - Перелогиньтесь, или выполните:

```
source ~/.bashrc
```

     - `yarn`
       - В файле `~/.bashrc` добавьте к переменной `PATH`:

```
PATH=~/.yarn/bin:$PATH
```

3. Установка git-flow
   - Linux
     - Gentoo

```
emerge dev-vcs/git-flow
```

- Ubuntu

```
apt-get install git-flow
```

- Fedora
  - Устанавливается из COPR:

```
sudo dnf -y copr enable elegos/gitflow
sudo dnf install gitflow
```

- Windows Git-flow входит в состав пакета git.

```
choco install git
```

- MacOS

```
brew install git-flow
```

4. Общепринятые коммиты
   1. commitizen
      - Данная программа используется для помощи в форматировании коммитов.
        - pnpm:

```
pnpm add -g commitizen
```

        - yarn:

```
yarn global add commitizen
```

      - При этом устанавливается скрипт `git-cz`, который мы и будем использовать для коммитов.
   2. cz-customizable
      - Данная программа позволяет более глубоко настроить форматирование коммитов.
        - pnpm:

```
pnpm add -g cz-customizable
```

   3. standard-version
      - Данная программа автоматизирует изменение номера версии.
        - pnpm:

```
pnpm add -g standard-version
```

— yarn:

```
yarn global add standard-version
```

1.8.2.2 Общие правила

— Для именования каталогов и файлов будем использовать соглашение Denote (см. 1.3).
— Рабочее пространство по предмету располагается в следующей иерархии:

```
~/work/study/
└── <учебное полугодие>/
    └── <учебное полугодие>==study--<код предмета>/
```

— Например, для 2026-2027 учебного года (второй семестр) и предмета «Компьютерный практикум по статистическому анализу данных» (код предмета `computer-practice`) структура каталогов примет следующий вид:

```
~/work/study/
└── 2026-2/
    └── 2026-2==study--computer-practice/
```

— Название проекта на хостинге git имеет вид (из-за ограничений на символы):

```
<учебный год>-study-<код предмета>
```

— Например, для 2026-2027 учебного года и предмета «Компьютерный практикум по статистическому анализу данных» (код предмета `computer-practice`) название проекта примет следующий вид:

```
2026-2-study-computer-practice
```

— Каталог для лабораторных работ имеет вид `labs`.
— Каталоги для лабораторных работ имеют вид `lab<номер>`, например: `lab01`, `lab02` и т.д.
— Каталог для групповых проектов имеет вид `group-project`.
— Каталог для персональных проектов имеет вид `personal-project`.
— Каталог для внешнего курса имеет вид `external-course`.
— Если проектов несколько, то они нумеруются подобно лабораторным работам.
— Этапы проекта обозначаются как `stage<номер>`.

1.8.2.3 Шаблон для рабочего пространства

- Репозиторий:
  - https://gitverse.ru/dharma/course-directory-student-template

1.8.2.3.1 Сознание репозитория курса на основе шаблона

1.8.2.3.1.1 Создание вручную

- Авторизуйтесь на https://gitverse.ru.
- Перейдите в репозиторий https://gitverse.ru/dharma/course-directory-student-template.
- Найдите кнопку (ссылку) «Использовать как шаблон» https://gitverse.ru/new?template_id=195300.
- После нажатия откроется форма для создания нового репозитория.
- Укажите название для вашего нового проекта: `2026-2-study-computer-practice`
- Укажите описание для вашего проекта.
- Выберите уровень доступа (публичный).
- Нажмите кнопку для подтверждения создания.
- Клонируйте репозиторий:

```
mkdir -p ~/work/study/2026-2/2026-2==study--computer-practice
cd ~/work/study/2026-2/2026-2==study--computer-practice
git clone ssh://git@gitverse.ru:2222/<owner>/2026-2-study-computer-p
↪ ractice.git
```

1.8.2.3.2 Структура шаблона

- Посмотреть доступные цели `make`:

  ```
  make help
  ```

- Посмотреть список доступных курсов:

  ```
  make list
  ```

- При создании структуры название курса берётся из следующих мест:
  - название курса находится в файле `COURSE`;
  - каталог курса называется как аббревиатура курса.

1.8.2.3.3 Настройка каталога курса

- Перейдите в каталог курса:

```
cd ~/work/study/2026-2/2026-2==study--computer-practice/2026-2-study-computer-practice
```

- Инициализируйте курс:

```
echo computer-practice > COURSE
make prepare
```

- Отправьте файлы на сервер:

```
git add .
git commit -am 'feat(main): make course structure'
git push
```

1.8.2.3.4 Скопируйте на sourcecraft

- Создайте на sourcecraft репозиторий `2026-2-study-computer-practice`
- Запуште репозиторий на sourcecraft:

```
git remote add src ssh://ssh.sourcecraft.dev/<owner>/2026-2-study-computer-practice.git
git push src master
```

1.8.2.3.5 Использование git flow

- Будем использовать для работы git flow (см. раздел 1.2.2).

1. Конфигурация git-flow

— Инициализируем git-flow

```
git flow init
```

Префикс для ярлыков установим в `v`.

— Проверьте, что Вы на ветке `develop`:

```
git branch
```

— Загрузите весь репозиторий в хранилище:

```
git push -u --all
```

— Создадим релиз с версией 1.0.0

```
git flow release start 1.0.0
```

— Создадим журнал изменений

```
standard-changelog --first-release
```

— Добавим журнал изменений в индекс

```
git add CHANGELOG.md
git commit -am 'chore(site): add changelog'
```

— Зальём релизную ветку в основную ветку

```
git flow release finish 1.0.0
```

— Отправим данные на github

```
git push --all
git push --tags
```

— Скопируем CHANGELOG.md в каталог `release`:

```
mkdir -p ../release
cp CHANGELOG.md ../release
```

— Создадим релиз.
— На gitverse сделайте это вручную.
— Для github будем использовать утилиты работы с github:

```
gh release create v1.0.0 -F ../release/CHANGELOG.md
```

1.8.2.4 Rutube

Рабочее пространство для лабораторной работы

Смотреть видео на Rutube

1.8.2.5 VKVideo

Рабочее пространство для лабораторной работы

Смотреть видео на VK

1.8.3 Создание проекта DrWatson

— Каталог DrWatson делается в рамках лабораторной работы:

```
cd labs/lab01
```

1.8.3.1 Создание каталога проекта DrWatson

1.8.3.1.1 Вариант A: Через REPL Julia

1. Откройте терминал и запустите Julia:

   ```
   julia
   ```

2. В REPL выполните:

```
using Pkg
Pkg.add("DrWatson")
using DrWatson
initialize_project("project"; authors="Ваше Имя", git=false)
```

3. Перейдите в созданный каталог:

```
cd("project")
```

1.8.3.1.2 Вариант Б: Через скрипт

Создайте файл `setup_project.jl`:

```
###!/usr/bin/env julia
### setup_project.jl

using Pkg
Pkg.add("DrWatson")
using DrWatson

### Создание проекта
project_name = "project"
initialize_project(project_name; authors="Ваше Имя", git=false)

println("✅ Проект создан: ", project_name)
println("📁 Перейдите в директорию: cd ", project_name)
```

Запустите:

```
julia setup_project.jl
cd project
```

1.8.3.2 Добавление необходимых пакетов

1.8.3.2.1 Вариант А: Установка скриптом

Создайте файл `add_packages.jl` в корне проекта:

```julia
###!/usr/bin/env julia
### add_packages.jl

using Pkg
Pkg.activate(".")  # Активируем текущий проект

### ОСНОВНЫЕ ПАКЕТЫ ДЛЯ РАБОТЫ
packages = [
    "DrWatson",           # Организация проекта
    "DifferentialEquations",  # Решение ОДУ
    "Plots",              # Визуализация
    "DataFrames",         # Таблицы данных
    "CSV",                # Работа с CSV
    "JLD2",               # Сохранение данных
    "Literate",           # Literate programming
    "IJulia",             # Jupyter notebook
    "BenchmarkTools",     # Бенчмаркинг
    "Quarto"              # Создание отчетов
]

println("Установка базовых пакетов...")
Pkg.add(packages)

println("\n✅ Все пакеты установлены!")
println("Для проверки: using DrWatson, DifferentialEquations, Plots")
```

Запустите:

```
julia add_packages.jl
```

1.8.3.2.2 Вариант Б: Поэтапная установка в REPL

1. Активируйте проект:

   ```julia
   using Pkg
   Pkg.activate(".")
   ```

2. Установите основные пакеты:

   ```julia
   Pkg.add("DrWatson")
   ```

```
Pkg.add("DifferentialEquations")
Pkg.add("Plots")
Pkg.add("DataFrames")
Pkg.add("Literate")
Pkg.add("CSV")
Pkg.add("JLD2")
Pkg.add("IJulia")
Pkg.add("BenchmarkTools")
Pkg.add("Quarto")
```

1.8.3.3 Проверка установки

Создайте тестовый скрипт `scripts/test_setup.jl`:

```
###!/usr/bin/env julia
### test_setup.jl

using DrWatson
@quickactivate "project"

println("✅ Проект активирован: ", projectdir())

### Проверка пакетов
packages = [
    "DrWatson",           # Организация проекта
    "DifferentialEquations",  # Решение ОДУ
    "Plots",              # Визуализация
    "DataFrames",         # Таблицы данных
    "CSV",                # Работа с CSV
    "JLD2",               # Сохранение данных
    "Literate",           # Literate programming
    "IJulia",             # Jupyter notebook
    "BenchmarkTools",     # Бенчмаркинг
    "Quarto"              # Создание отчетов
]

println("\nПроверка пакетов:")
for pkg in packages
    try
        eval(Meta.parse("using $pkg"))
        println("  ✓ $pkg")
```

```
    catch e
        println("  × $pkg: Ошибка загрузки")
    end
end

### Проверка путей
println("\nСтруктура проекта:")
println("  Корень:        ", projectdir())
println("  Данные:        ", datadir())
println("  Скрипты:       ", srcdir())
println("  Графики:       ", plotsdir())
```

Запустите:

```
julia --project=. scripts/test_setup.jl
```

1.8.3.4 Итоговая структура проекта

```
lab01
├── presentation
├── project/
│   ├── Project.toml          # Зависимости проекта
│   ├── Manifest.toml         # Точные версии
│   ├── src/                  # Исходный код пакета
│   ├── data/                 # Данные
│   ├── scripts/              # Скрипты
│   ├── plots/                # Графики
│   ├── papers/               # Статьи
│   ├── docs/                 # Документация
│   └── test/                # Тесты
└── report
```

1.8.4 Задание

— Создать рабочий каталог для всего курса.

- Создать рабочее пространство для программ в рамках лабораторной работы.
- Выполнить все задания по тексту лабораторной работы.
- Установить необходимые пакеты.
- Выполнить предложенный код.
- Преобразовать код в литературный стиль.
- Сгенерировать из литературного кода:
  - чистый код;
  - jupyter notebook;
  - документацию в формате Quarto.
- Выполнить код из jupyter notebook.
- Интегрировать документацию в формате Quarto в отчёт.
- Добавить в код в литературном стиле вычисление для набора параметров.
- Сгенерировать из литературного кода с параметрами:
  - чистый код;
  - jupyter notebook;
  - документацию в формате Quarto.
- Выполнить код из jupyter notebook с параметрами.
- Интегрировать документацию с параметрами в формате Quarto в отчёт.

Список литературы

1. A Multi-Language Computing Environment for Literate Programming and Reproducible Research / E. Schulte [et al.] // Journal of Statistical Software. — 2012. — Vol. 46, no. 3. — ISSN 1548-7660. — DOI: 10.18637/jss.v046.i03.
2. *Knuth D. E.* Literate Programming // The Computer Journal. — 1984. — Feb. — Vol. 27, no. 2. — P. 97–111. — ISSN 1460-2067. — DOI: 10.1093/comjnl/27.2.97.
3. The Story in the Notebook / M. B. Kery [et al.] // Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems. — ACM, 04/2018. — P. 1–11. — DOI: 10.1145/3173574.3173748.