

Лабораторная работа 2

ДОКУМЕНТИРОВАНИЕ ТРЕБОВАНИЙ И ПРОЕКТИРОВАНИЕ АРХИТЕКТУРЫ СИСТЕМЫ ПРИ ПОМОЩИ ЯЗЫКА UML. РАБОТА С ВЕТКАМИ В СИСТЕМЕ GIT

2.1 Подготовка к выполнению работы

1. В рабочем каталоге (например, D:\GIT_PRACTICE) создать подкаталоги, в которых будет выполняться вся дальнейшая работа (например, D:\GIT_PRACTICE\requirements и models).

2. Загрузить и установить Visual Paradigm Community Edition, который будет использоваться в данной работе для создания UML-диаграмм.

2.2 Документирование требований

2.2.1 Пользовательские истории

Традиционным способом документирования требований являются списки требований, которые могут занимать сотни или даже тысячи страниц для сложных систем. В современном анализе такие списки требований крайне неэффективны, хотя продолжают использоваться и по сей день.

Одной из альтернатив большим, предопределенным спискам требований являются *пользовательские истории* (user story), которые определяются обычным языком. В целом, в 1990-х были введены методики, призванные решить проблемы анализа требований, среди которых:

- 1) унифицированный язык моделирования **UML** (Unified Modeling Language);
- 2) сценарии использования;
- 3) гибкая методология разработки.

В настоящее время широко используется фреймворк гибкой разработки программного обеспечения **Scrum**. Фреймворк представляет собой набор принципов, на которых строится процесс разработки, позволяющий

в жестко фиксированные и небольшие по времени итерации, называемые **спринтами** (sprints), предоставить конечному пользователю работающий продукт с новыми возможностями, для которых определен наибольший приоритет (рисунок 2.1).

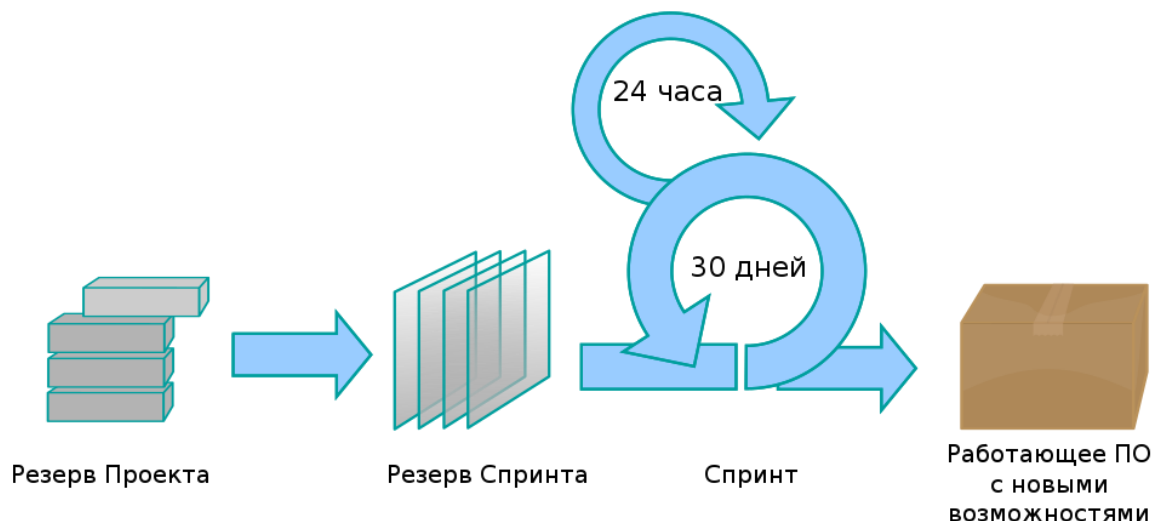


Рисунок 2.1

Помимо спринта, основными определениями Scrum являются:

1. Резерв проекта (project backlog) – список требований к функциональности, упорядоченный по их степени важности, подлежащих реализации. Элементами этого списка как раз таки и являются пользовательские истории, называемые в данном случае элементами резерва (backlog items).

2. Резерв спринта (sprint backlog) – содержит функциональность, выбранную **владельцем продукта** из резерва проекта. Все функции разбиты по задачам, каждая из которых оценивается командой.

Более подробно о фреймворке Scrum можно прочитать по ссылке:

<https://ru.wikipedia.org/wiki/Scrum>

Обязательными полями при формировании пользовательских историй, на примере Scrum, являются:

1. ID – уникальный идентификатор, порядковый номер, применяемый для идентификации историй в случае их переименования.

2. Название (name) – краткое описание истории. Поскольку название должно быть однозначным, чтобы и разработчики, и владелец продукта могли понять, о чем идет речь и отличить одну историю от другой, зачастую истории имеют следующую структуру:

Будучи пользователем <**тип пользователя**>, я хочу сделать <**действие**>, чтобы получить <**результат**>

Такая структура удобна тем, что понятна как разработчикам, так и заказчикам.

3. Важность (importance) – степень важности данной истории, по мнению владельца продукта. Обычно представляет собой натуральное число из последовательности Фибоначчи (1, 2, 3, 5, 8, 13, 21, 34, 55). Чем выше значение, тем выше приоритет пользовательской истории.

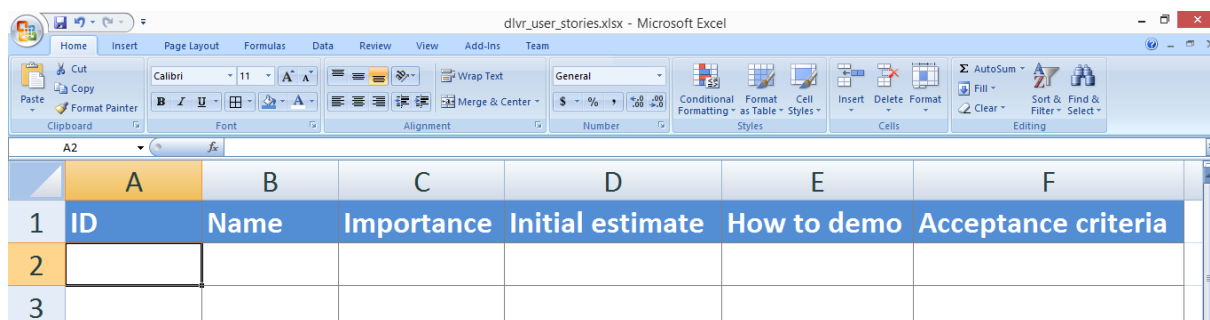
4. Предварительная оценка (initial estimate) – начальная оценка объема работ, необходимого для реализации истории по сравнению с другими историями. Измеряется при помощи **абстрактной метрики оценки** (story points) сложности, которая не учитывает затраты в человеко-часах. В качестве шкалы также используется ряд Фибоначчи.

5. Как продемонстрировать (how to demo) – краткое пояснение того, как завершенная задача будет продемонстрирована в конце спринта. Данное поле может представлять собой код автоматизированного теста.

6. Критерии приемки (acceptance criteria) – значимые детали реализации истории, уточняющие требования владельца продукта, собранные всеми участниками команды при планировании спринта.

В самом простом случае, для документирования пользовательских историй можно использовать электронные таблицы Excel или, например, Google Sheets, что более предпочтительно при командной работе.

Создав в рабочем каталоге (D:\GIT_PRACTICE\requirements) файл Excel (например, dlvr_user_stories.xlsx), необходимо открыть его и сформировать структуру таблицы, которая будет содержать пользовательские истории (рисунок 2.2):



	A	B	C	D	E	F
1	ID	Name	Importance	Initial estimate	How to demo	Acceptance criteria
2						
3						

Рисунок 2.2

Для рассматриваемой в качестве примеров выполнения лабораторного практикума предметной области (приобретение некоторой фирмой товаров у различных поставщиков), пользовательские истории могут быть следующими (рисунок 2.3):

ID	Name	Importance	Initial estimate	How to demo	Acceptance criteria
1	Будучи сотрудником отдела маркетинга, я хочу просматривать информацию о наличии продукции на складе, чтобы обработать заказ клиента	8	2	Тест shouldReturnListOfAvailableProducts()	Успешное прохождение теста
2	Будучи сотрудником отдела маркетинга, я хочу работать с информацией о поставщиках, чтобы сформировать заказ на поставку	5	5	Тесты shouldCreateLESupplier(), shouldCreatePESupplier(), shouldUpdateLESupplier(), shouldUpdatePESupplier(), shouldRemoveSupplier(), shouldReturnListOfSuppliers()	Успешное прохождение тестов
3	Будучи сотрудником отдела снабжения, я хочу работать с информацией о договорах, чтобы зарегистрировать в системе заключение договора на поставку	3	3	Тесты shouldCreateContract(), shouldUpdateContract(), shouldRemoveContract(), shouldReturnListOfContracts()	Успешное прохождение тестов
4	Будучи сотрудником отдела снабжения, я хочу работать с информацией о поставленных товарах, чтобы обновить данные о наличии продукции на складе	5	5	Тесты shouldCreateSuppliedProduct(), shouldUpdateSuppliedProduct(), shouldRemoveSuppliedProduct(), shouldReturnListOfSuppliedProductsByContractNumber()	Успешное прохождение тестов
5	Будучи сотрудником отдела снабжения, я хочу сформировать приходную накладную, чтобы оформить приход товаров на склад	2	8	Тест shouldReturnPurchaseInvoice()	Успешное прохождение теста
6	Будучи сотрудником бухгалтерии, я хочу сформировать счет на оплату, чтобы оплатить поставку	1	8	Тест shouldReturnPaymentInvoice()	Успешное прохождение теста

Рисунок 2.3

После того, как список пользовательских историй для **индивидуальной предметной области** будет сформирован, его необходимо ***зафиксировать*** в системе управления версиями Git при помощи команды git commit. Дальнейшие изменения списка пользовательских историй после обсуждения с преподавателем также необходимо фиксировать в Git с указанием соответствующих ***комментариев***.

2.2.2 Сценарии использования

Сценарий использования (также: прецедент, вариант использования, англ. Use Case) представляет собой спецификацию последовательности действий в языке UML, которые может осуществлять система, подсистема или класс, взаимодействуя с ***внешними действующими лицами*** (actors).

Прецеденты служат для документирования функциональных требований к программным системам. На диаграммах вариантов использования в UML прецедент отображается в виде эллипса. Внутри эллипса или под ним указывается имя элемента.

К прецедентам в UML применимы следующие виды отношений:

1. Ассоциация (association) – может указывать на то, что действующее лицо инициирует соответствующий вариант использования.
2. Расширение (extend) – разновидность отношения зависимости между базовым вариантом использования и его специальным случаем.
3. Включение (include) – определяет взаимосвязь базового варианта использования с другим вариантом использования, функциональное поведение которого всегда задействуется базовым вариантом использования.
4. Обобщение (generalization) – моделирует соответствующую общность ролей.

При помощи инструментария Visual Paradigm Community Edition (или другого CASE-средства) необходимо выполнить документирование сформированных ранее пользовательских историй в виде сценариев использования.

Для рассматриваемой в качестве примеров выполнения лабораторного практикума предметной области (приобретение некоторой фирмой товаров у различных поставщиков), сценарии использования могут быть следующими:

1. Взаимодействие системы с сотрудником отдела маркетинга (рисунок 2.4):

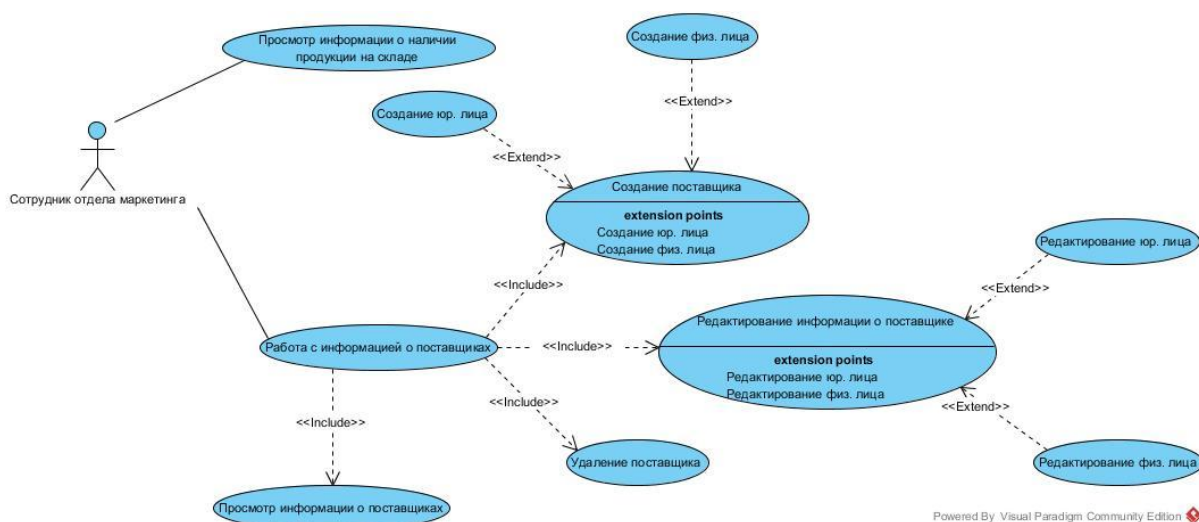


Рисунок 2.4

2. Взаимодействие системы с сотрудником бухгалтерии (рисунок 2.5):

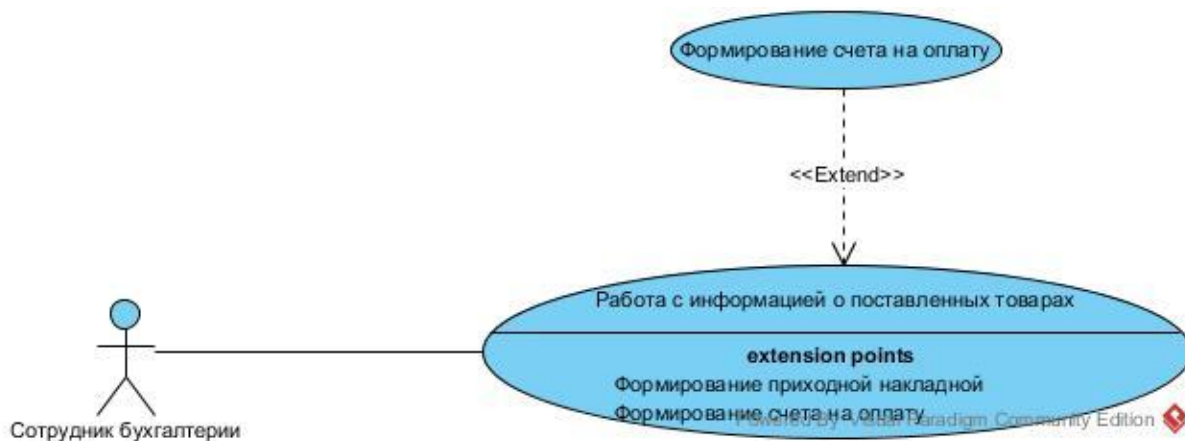


Рисунок 2.5

3. Взаимодействие системы с сотрудником отдела снабжения (рисунок 2.6):

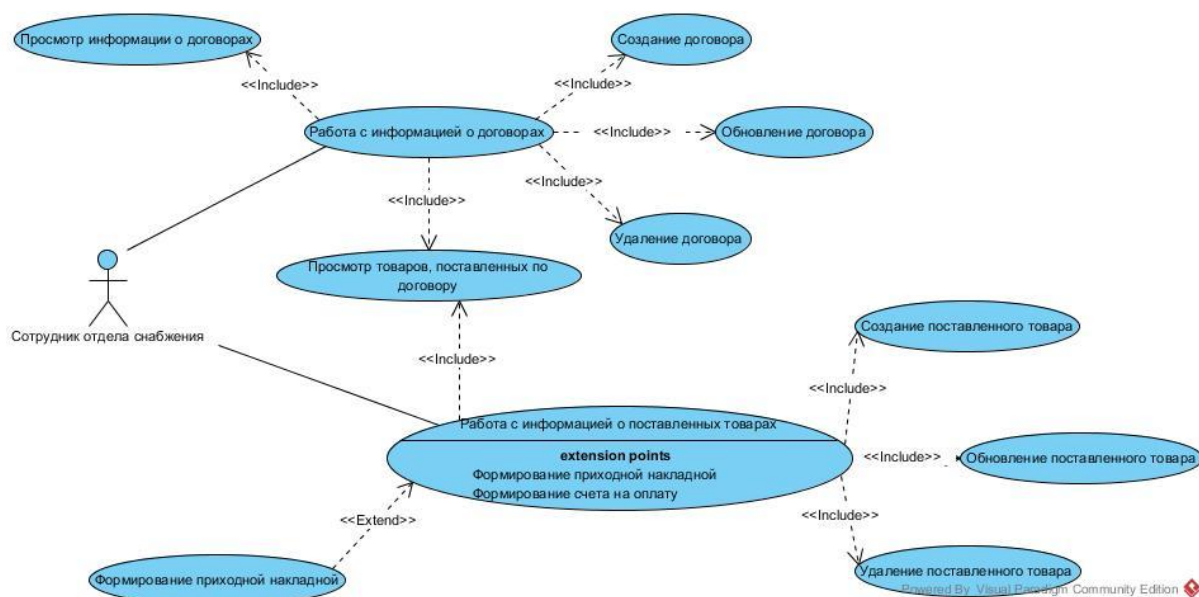


Рисунок 2.6

Созданный в Visual Paradigm проект необходимо сохранить в соответствующем подкаталоге (заранее созданном) рабочего каталога (например, D:\GIT_PRACTICE\models) и **зафиксировать** изменения в системе управления версиями Git. Дальнейшие изменения сценариев использования после обсуждения с преподавателем также необходимо фиксировать в Git с указанием соответствующих **комментариев**.

2.2.3 Детальные требования

На предыдущих этапах выполнения работы было выполнено документирование **требований заказчика** (С-требования) к разрабатываемой, согласно **заданной предметной области**, информационной системе в виде пользовательских историй и сценариев использования.

Теперь же необходимо сформировать **детальные требования** (D-требования), которые будут использоваться для проектирования и разработки программного обеспечения. При этом D-требования должны быть

получены из ранее сформированных С-требований, быть отслеживаемыми и согласованными с требованиями заказчика.

Для рассматриваемой в качестве примеров выполнения лабораторного практикума предметной области (приобретение некоторой фирмой товаров у различных поставщиков), документирование детальных требований необходимо осуществить при помощи следующих UML-диаграмм:

1. Диаграмма деятельности (Activity) для сценария «Создание договора» (рисунок 2.7):

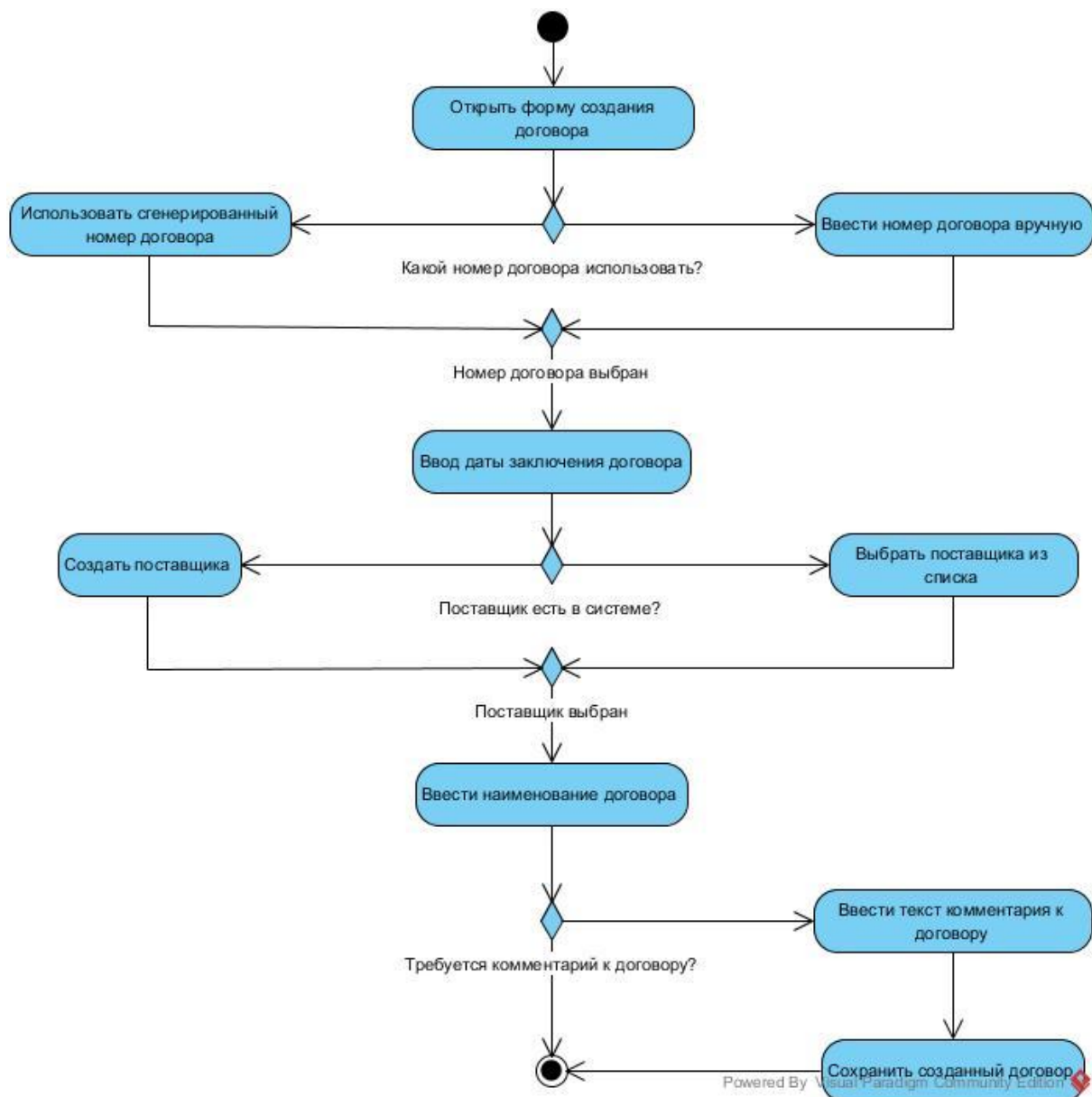


Рисунок 2.7

2. Диаграмма последовательности (Sequence) для сценария «Создание договора» (рисунок 2.8):

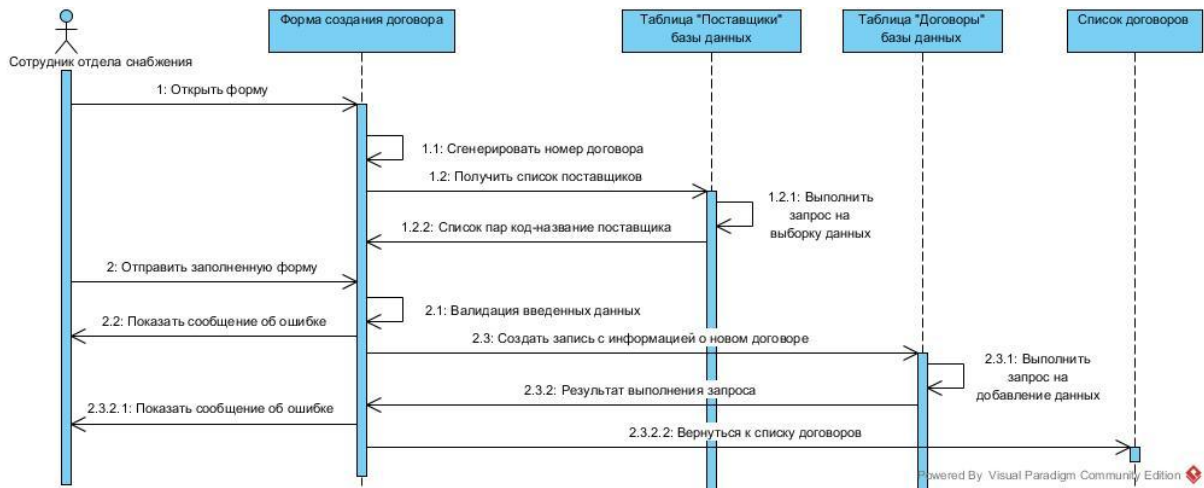


Рисунок 2.8

3. Диаграмма классов (рисунок 2.9):

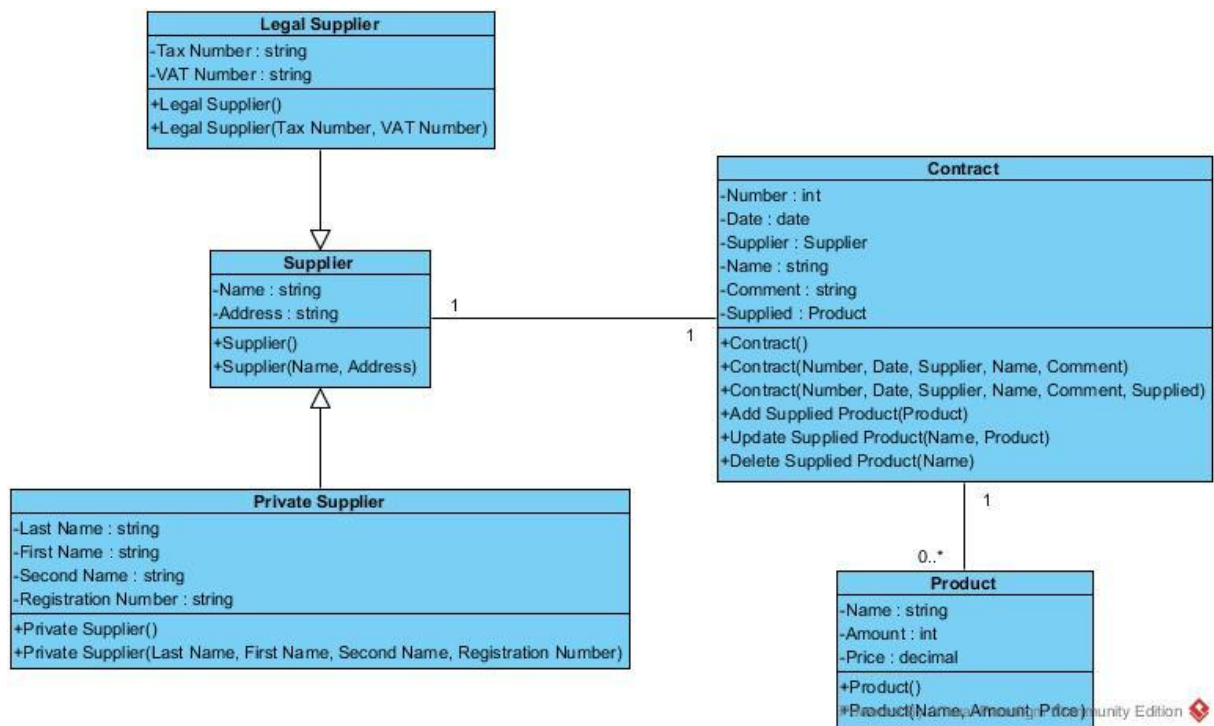


Рисунок 2.9

Изменения в проекте необходимо **зафиксировать** в системе управления версиями Git. Дальнейшие изменения диаграмм деятельности, последовательности (данные диаграммы необходимо разработать **для каждого сценария использования**) и классов после обсуждения с преподавателем также необходимо фиксировать в Git с указанием соответствующих **комментариев**.

2.3 Проектирование архитектуры системы

Разбиение программной системы на структурные компоненты и связи (зависимости) между компонентами демонстрирует статическая структурная диаграмма языка моделирования UML – диаграмма **компонентов** (Component diagram).

В качестве физических компонентов могут выступать файлы, библиотеки, модули, исполняемые файлы, пакеты и т.п.

Компоненты связываются через **зависимости**, когда соединяется требуемый интерфейс одного компонента с имеющимся интерфейсом другого компонента. Таким образом, иллюстрируются отношения **клиент-источник** между двумя компонентами. Зависимость показывает, что один компонент предоставляет сервис, необходимый другому компоненту. Зависимость изображается стрелкой от интерфейса или порта клиента к импортируемому интерфейсу.

Когда диаграмма компонентов используется, чтобы показать внутреннюю структуру компонентов, предоставляемый и требуемый интерфейсы составного компонента могут **делегироваться** в соответствующие интерфейсы внутренних компонентов. Делегация показывается связь внешнего контракта компонента с внутренней реализацией этого поведения внутренними компонентами.

Для рассматриваемой в качестве примеров выполнения лабораторного практикума предметной области (приобретение некоторой фирмой то-

варов у различных поставщиков) необходимо разработать диаграмму компонентов проектируемой программной системы (рисунок 2.10):

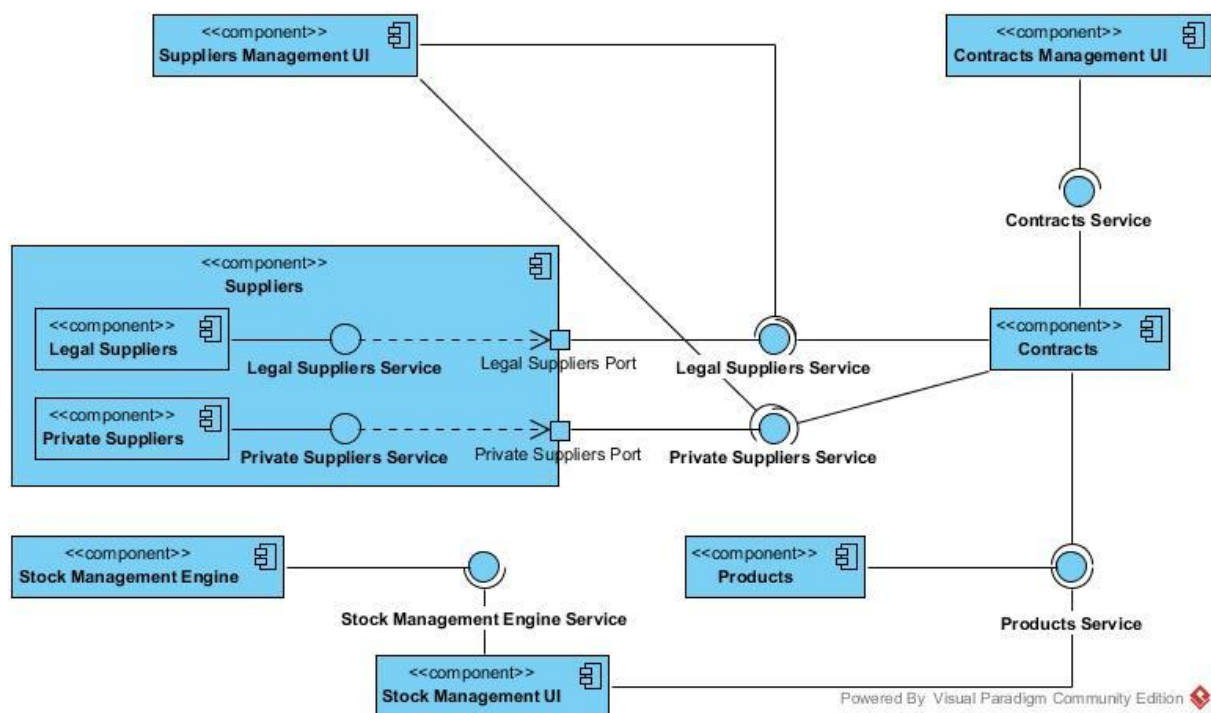


Рисунок 2.10

Диаграмма развёртывания (Deployment diagram) в UML моделирует физическое развертывание артефактов на узлах. Диаграмма развертывания должна демонстрировать:

- 1) «узлы» – аппаратные компоненты (например, web-сервер, сервер базы данных, сервер приложения);
- 2) «артефакты» – программные компоненты, которые работают на каждом узле (например, web-приложение, база данных);
- 3) связи различных частей этого комплекса друг с другом.

Узлы представляются как прямоугольные параллелепипеды с артефактами, расположенными в них, изображенными в виде прямоугольников. Узлы могут иметь подузлы, которые представляются как вложенные прямоугольные параллелепипеды. Один узел диаграммы развертывания

может концептуально представлять множество физических узлов, таких как кластер серверов баз данных.

Существует два типа узлов:

1) узел устройства – физический вычислительный ресурс со своей памятью и сервисами для выполнения программного обеспечения (например, персональный компьютер, мобильный телефон и т.п.);

2) узел среды выполнения – это программный вычислительный ресурс, который работает внутри внешнего узла, и который предоставляет собой сервис, выполняющий другие исполняемые программные элементы.

Для рассматриваемой в качестве примеров выполнения лабораторного практикума предметной области (приобретение некоторой фирмой товаров у различных поставщиков) необходимо разработать диаграмму компонентов проектируемой программной системы (рисунок 2.11):

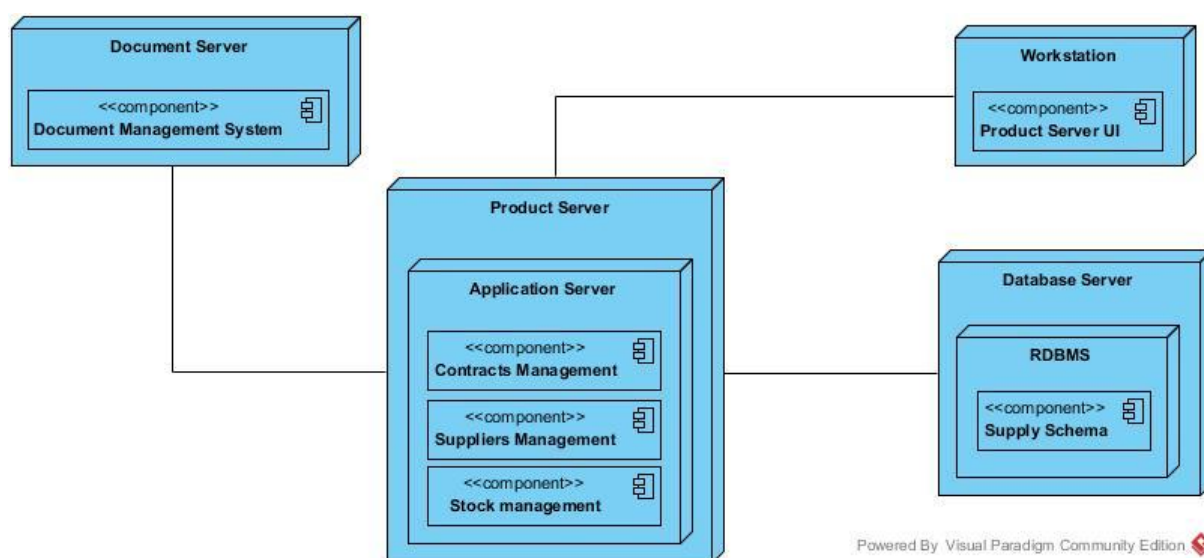


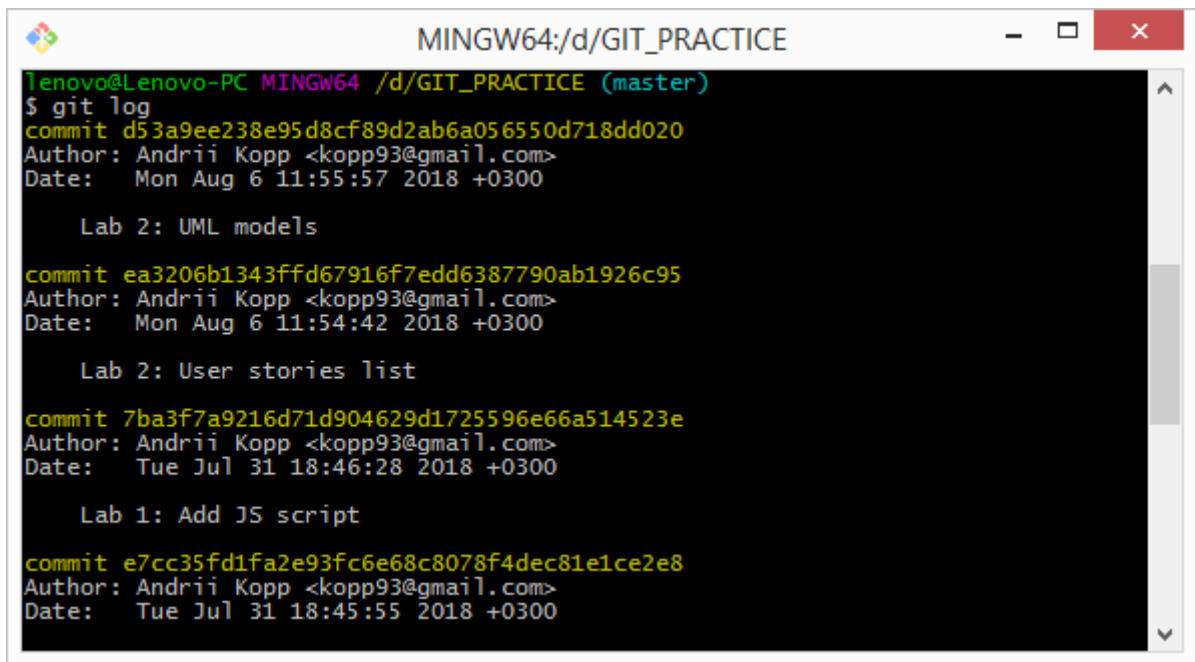
Рисунок 2.11

Созданные диаграммы необходимо **зафиксировать** в системе управления версиями Git. Дальнейшие изменения диаграмм компонентов и развертывания после обсуждения с преподавателем также необходимо фиксировать в Git с указанием соответствующих **комментариев**.

2.4 Работа с ветками в системе Git

2.4.1 Ветвление и слияние

В результате работы над лабораторным практикумом в ветке master, используемой системой Git по умолчанию, уже имеется несколько коммитов (рисунок 2.12).

A screenshot of a terminal window titled 'MINGW64:/d/GIT_PRACTICE'. The terminal shows the output of the 'git log' command. The output lists four commits in reverse chronological order. Each commit entry includes a commit hash, the author's name and email, the date, and a description of the commit. The commits are: 1. 'commit d53a9ee238e95d8cf89d2ab6a056550d718dd020' by Andrii Kopp, dated Mon Aug 6 11:55:57 2018, with description 'Lab 2: UML models'. 2. 'commit ea3206b1343ffd67916f7edd6387790ab1926c95' by Andrii Kopp, dated Mon Aug 6 11:54:42 2018, with description 'Lab 2: User stories list'. 3. 'commit 7ba3f7a9216d71d904629d1725596e66a514523e' by Andrii Kopp, dated Tue Jul 31 18:46:28 2018, with description 'Lab 2: User stories list'. 4. 'commit e7cc35fd1fa2e93fc6e68c8078f4dec81e1ce2e8' by Andrii Kopp, dated Tue Jul 31 18:45:55 2018, with description 'Lab 1: Add JS script'. The terminal window has a standard Windows-style title bar with minimize, maximize, and close buttons.

```
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git log
commit d53a9ee238e95d8cf89d2ab6a056550d718dd020
Author: Andrii Kopp <kopp93@gmail.com>
Date: Mon Aug 6 11:55:57 2018 +0300

    Lab 2: UML models

commit ea3206b1343ffd67916f7edd6387790ab1926c95
Author: Andrii Kopp <kopp93@gmail.com>
Date: Mon Aug 6 11:54:42 2018 +0300

    Lab 2: User stories list

commit 7ba3f7a9216d71d904629d1725596e66a514523e
Author: Andrii Kopp <kopp93@gmail.com>
Date: Tue Jul 31 18:46:28 2018 +0300

    Lab 2: User stories list

commit e7cc35fd1fa2e93fc6e68c8078f4dec81e1ce2e8
Author: Andrii Kopp <kopp93@gmail.com>
Date: Tue Jul 31 18:45:55 2018 +0300

    Lab 1: Add JS script
```

Рисунок 2.12

Представим, что после того, как было выполнено документирование требований и проектирование архитектуры создаваемой системы, далее необходимо перейти к работе над созданием прототипа информационной системы.

Так как создание прототипа является обособленной задачей в рамках традиционного жизненного цикла разработки программного обеспечения (рисунок 2.13), необходимо создать новую ветку в системе управления версиями Git и работать на ней.

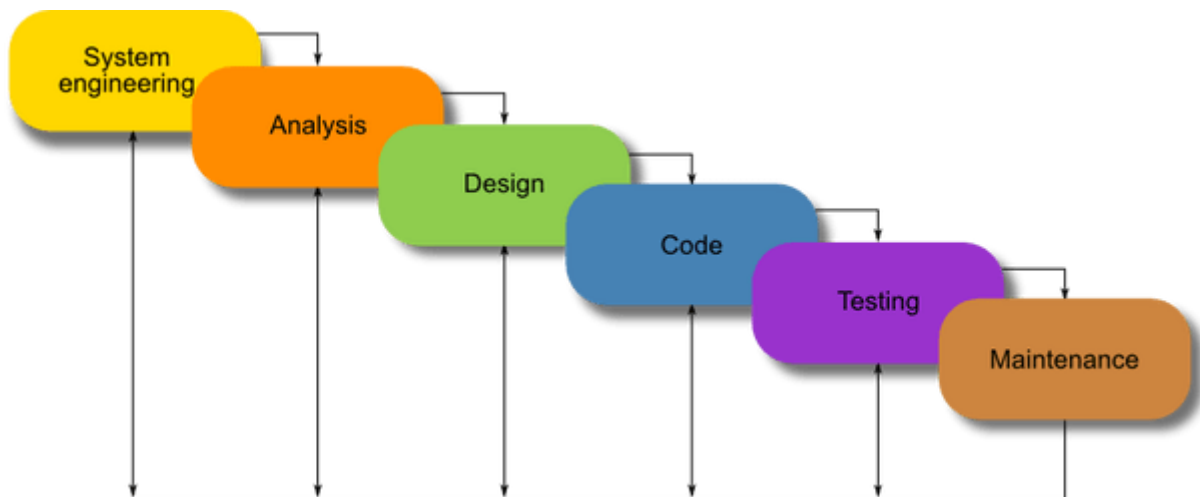


Рисунок 2.13

Для создания новой ветки и перехода на нее, необходимо использовать команду `git checkout` с ключом `-b` (рисунок 2.14):

```
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git checkout -b design
Switched to a new branch 'design'
```

Рисунок 2.14

Выполнение команды `checkout` с ключом `-b` является сокращением для двух команд:

```
git branch design # создание новой ветки
git checkout design # переход на новую ветку
```

После создания, новая ветка указывает на тот же коммит, что и ветка `master`, поскольку никаких изменений в ветке `design` еще не было зафиксировано (рисунок 2.15).



```
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (design)
$ git log
commit d53a9ee238e95d8cf89d2ab6a056550d718dd020
Author: Andrii Kopp <kopp93@gmail.com>
Date: Mon Aug 6 11:55:57 2018 +0300

    Lab 2: UML models
```

Рисунок 2.15

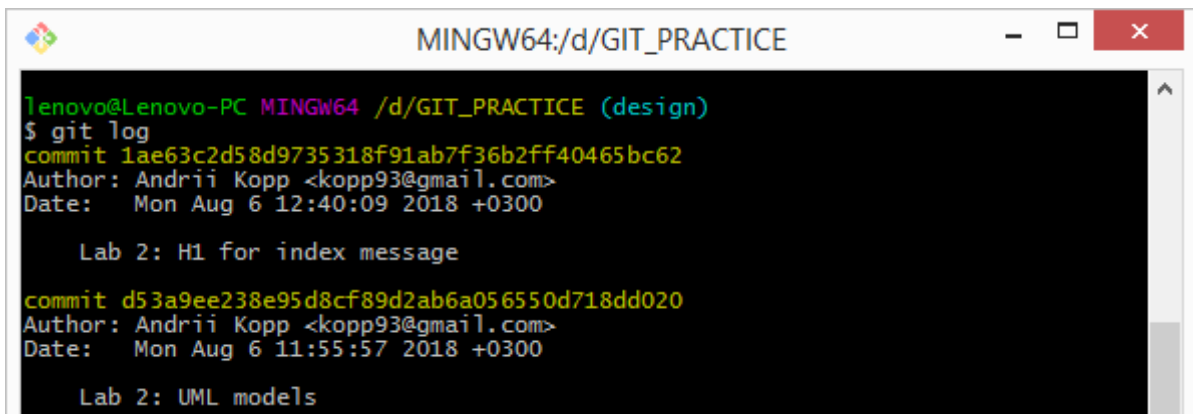
Конечно же, во время работы над прототипом создаваемой системы будут сделаны и зафиксированы некоторые изменения. Например, появилась необходимость использовать заголовок первого уровня для сообщения, выводимого на странице index.html (рисунок 2.16):



```
index.html - Notepad
File Edit Format View Help
<!DOCTYPE html>
<html>
<head>
<title>Index</title>
</head>
<body>
<h1>
<div id="hello"></div>
</h1>
<script src="script.js"></script>
</body>
</html>
```

Рисунок 2.16

Внесенные изменения необходимо зафиксировать. После коммита ветка design, в которой выполнялась работа, будет указывать уже на последний коммит, связанный с добавлением заголовка в файл index.html, т.е. сдвинется вперед, относительно ветки master (рисунок 2.17).

A screenshot of a terminal window titled 'MINGW64:/d/GIT_PRACTICE'. The prompt is 'lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (design)'. The user has entered '\$ git log'. The output shows two commits. The first commit has hash '1ae63c2d58d9735318f91ab7f36b2ff40465bc62', author 'Andrii Kopp <kopp93@gmail.com>', and date 'Mon Aug 6 12:40:09 2018 +0300'. The commit message is 'Lab 2: H1 for index message'. The second commit has hash 'd53a9ee238e95d8cf89d2ab6a056550d718dd020', the same author, and date 'Mon Aug 6 11:55:57 2018 +0300'. The commit message is 'Lab 2: UML models'.

```
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (design)
$ git log
commit 1ae63c2d58d9735318f91ab7f36b2ff40465bc62
Author: Andrii Kopp <kopp93@gmail.com>
Date:   Mon Aug 6 12:40:09 2018 +0300

    Lab 2: H1 for index message

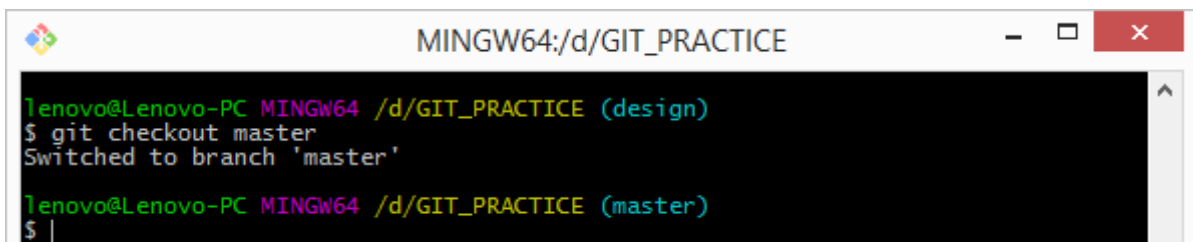
commit d53a9ee238e95d8cf89d2ab6a056550d718dd020
Author: Andrii Kopp <kopp93@gmail.com>
Date:   Mon Aug 6 11:55:57 2018 +0300

    Lab 2: UML models
```

Рисунок 2.17

Предположим, что по какой-то причине потребовалось изменить цвет выводимого сообщения на странице index.html на красный. Причем, сделать это необходимо следующим образом:

1. Убедившись, что все изменения на ветке design зафиксированы, переключиться на ветку master (рисунок 2.18):

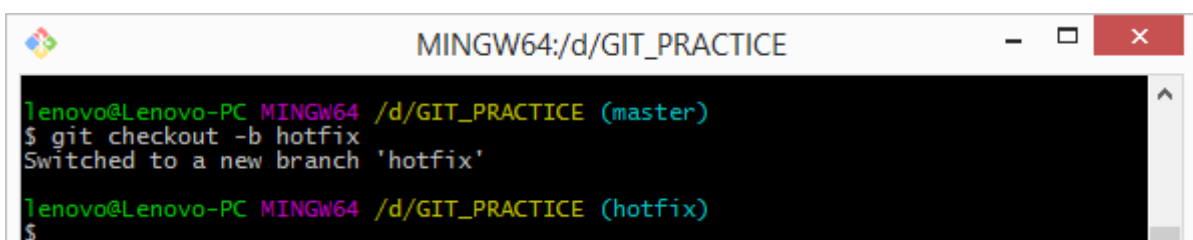
A screenshot of a terminal window titled 'MINGW64:/d/GIT_PRACTICE'. The prompt is 'lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (design)'. The user has entered '\$ git checkout master'. The output is 'Switched to branch 'master''. The prompt now shows '(master)'. The user has entered '\$ |' and the cursor is at the end of the line.

```
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (design)
$ git checkout master
Switched to branch 'master'

lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ |
```

Рисунок 2.18

2. Создать ветку, в которой будет выполняться работа (рисунок 2.19):

A screenshot of a terminal window titled 'MINGW64:/d/GIT_PRACTICE'. The prompt is 'lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)'. The user has entered '\$ git checkout -b hotfix'. The output is 'Switched to a new branch 'hotfix''. The prompt now shows '(hotfix)'. The user has entered '\$' and the cursor is at the end of the line.

```
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git checkout -b hotfix
Switched to a new branch 'hotfix'

lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (hotfix)
$
```

Рисунок 2.19

3. Внести требуемые изменения в файл index.html и сделать коммит с соответствующим комментарием (рисунок 2.20):



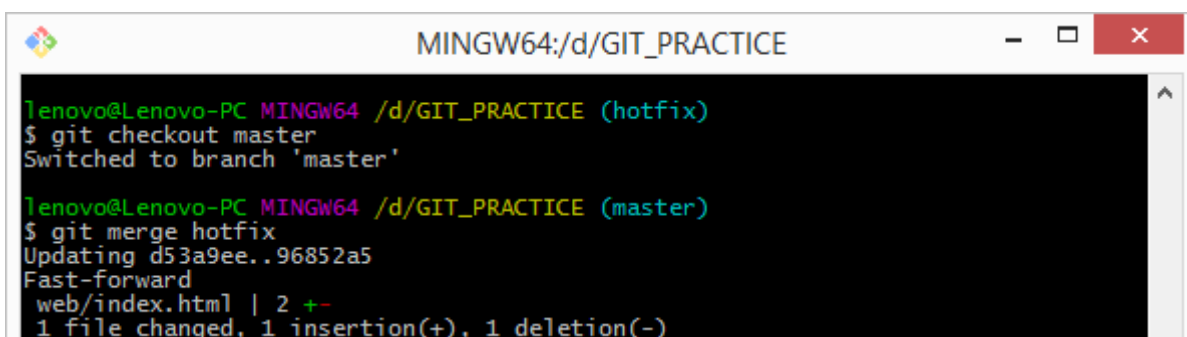
```
MINGW64:/d/GIT_PRACTICE

lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (hotfix)
$ cat web/index.html
<!DOCTYPE html>
<html>
<head>
<title>Index</title>
</head>
<body>
<div id="hello" style="color: red"></div>
<script src="script.js"></script>
</body>
</html>
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (hotfix)
$ git add web/index.html

lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (hotfix)
$ git commit -m "Lab 2: Index page hotfix"
[hotfix 96852a5] Lab 2: Index page hotfix
1 file changed, 1 insertion(+), 1 deletion(-)
```

Рисунок 2.20

4. Слить изменения в ветку master, чтобы включить их в проект, при помощи команды git merge (рисунок 2.21):



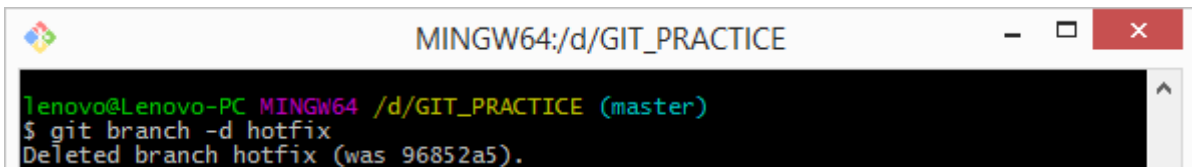
```
MINGW64:/d/GIT_PRACTICE

lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (hotfix)
$ git checkout master
Switched to branch 'master'

lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git merge hotfix
Updating d53a9ee..96852a5
Fast-forward
 web/index.html | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)
```

Рисунок 2.21

5. Удалить не нужную больше ветку hotfix (ветка master после слияния указывает на то же место, так как система Git просто *переместила указатель вперед* из-за отсутствия расходящихся изменений, которые нужно было бы сливать воедино) при помощи команды git branch с опцией -d (рисунок 2.22):

A screenshot of a Windows terminal window titled "MINGW64:/d/GIT_PRACTICE". The prompt is "lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)". The user has entered the command "\$ git branch -d hotfix", and the output is "Deleted branch hotfix (was 96852a5).".

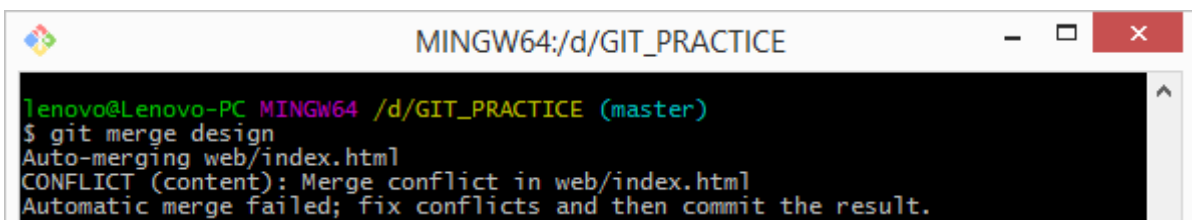
```
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git branch -d hotfix
Deleted branch hotfix (was 96852a5).
```

Рисунок 2.22

После того как проблема решена, можно вернуться обратно к ветке design и продолжить работу. Однако необходимо помнить, что работа, сделанная на ветке hotfix, не включена в коммиты на ветке design. Если необходимо, ветку master можно слить в ветку design посредством команды `git merge master`. Кроме того, интеграцию изменений можно отложить до тех пор, пока изменения на ветке design не будет решено включить в основную ветку проекта master.

2.4.2 Конфликты при слиянии

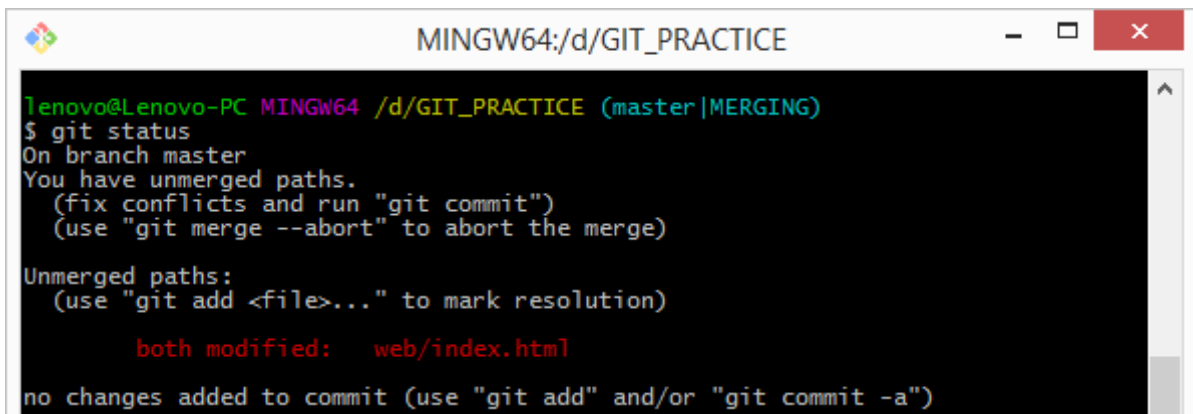
Процесс слияния веток не всегда проходит гладко. В нашем случае решение задачи в ветке design изменяет тот же файл (`index.html`), что и ветка hotfix, в результате чего будет получен конфликт слияния (рисунок 2.23):

A screenshot of a Windows terminal window titled "MINGW64:/d/GIT_PRACTICE". The prompt is "lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)". The user has entered the command "\$ git merge design". The output shows "Auto-merging web/index.html", followed by a conflict message: "CONFLICT (content): Merge conflict in web/index.html", and finally "Automatic merge failed; fix conflicts and then commit the result.".

```
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git merge design
Auto-merging web/index.html
CONFLICT (content): Merge conflict in web/index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Рисунок 2.23

Система Git не создаст новый коммит для слияния веток, а приостановит этот процесс до тех пор, пока пользователь не разрешит конфликт. Для просмотра файлов, не прошедших слияние, необходимо выполнить команду `git status` (рисунок 2.24):

A screenshot of a terminal window titled 'MINGW64:/d/GIT_PRACTICE'. The prompt is 'lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master|MERGING)'. The user has entered '\$ git status'. The output shows 'On branch master', 'You have unmerged paths.', instructions to fix conflicts or abort the merge, and 'Unmerged paths: (use "git add <file>..." to mark resolution)'. A red line indicates 'both modified: web/index.html'. The final line says 'no changes added to commit (use "git add" and/or "git commit -a")'.

```
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master|MERGING)
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

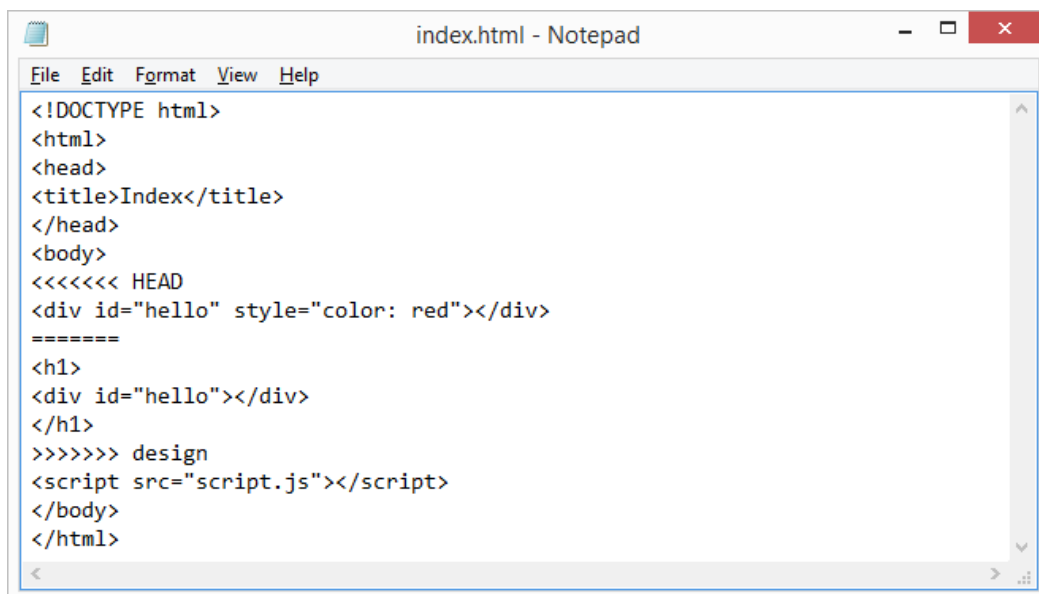
Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:   web/index.html

no changes added to commit (use "git add" and/or "git commit -a")
```

Рисунок 2.24

Система Git добавляет стандартные маркеры к файлам (здесь это index.html), которые имеют конфликт (unmerged), так что можно открыть такие файлы вручную и разрешить эти конфликты. Файл index.html будет выглядеть следующим образом (рисунок 2.25):

A screenshot of a Notepad window titled 'index.html - Notepad'. The menu bar includes 'File', 'Edit', 'Format', 'View', and 'Help'. The text content shows an HTML document with a conflict. The first part, marked with '<<<<<<<< HEAD', is a red-styled 'hello' message. This is followed by '=====' and then the second part, marked with '>>>>>>> design', which is a 'design' message. The document is enclosed in standard HTML tags like <!DOCTYPE html>, <html>, <head>, <title>, </head>, <body>, </body>, and </html>.

```
index.html - Notepad
File Edit Format View Help
<!DOCTYPE html>
<html>
<head>
<title>Index</title>
</head>
<body>
<<<<<<< HEAD
<div id="hello" style="color: red"></div>
=====
<h1>
<div id="hello"></div>
</h1>
>>>>>>> design
<script src="script.js"></script>
</body>
</html>
```

Рисунок 2.25

В файле index.html все, что выше ===== это версия из HEAD (ветка master, так как именно на ней была выполнена команда merge). Все, что находится ниже – версия в ветке design. Для разрешения конфликта необходимо либо выбрать одну из этих частей, либо как-то объединить содер-

жимое по своему усмотрению. Например, конфликт может быть разрешен следующим образом (рисунок 2.25):

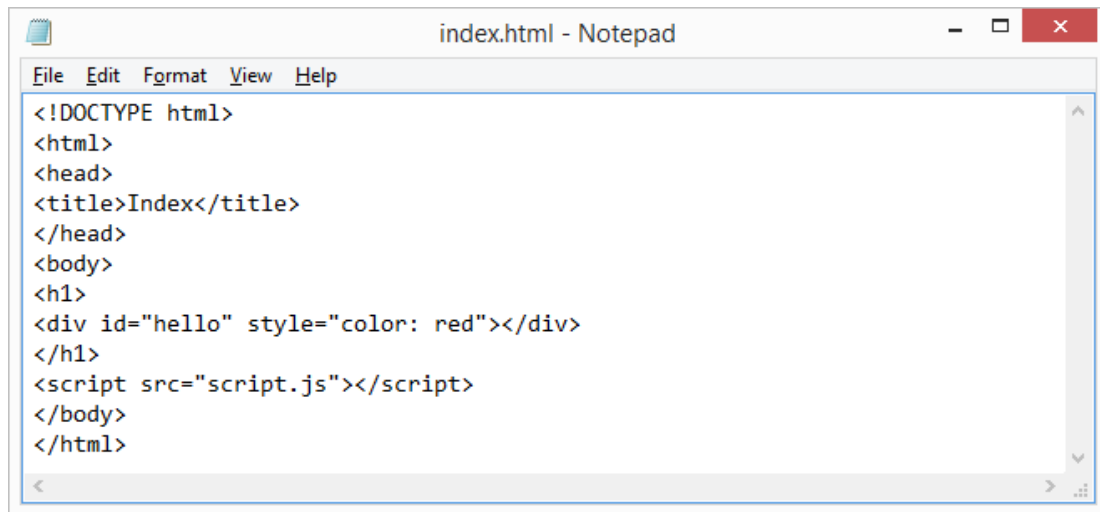


Рисунок 2.25

После разрешения конфликтов, для каждого конфликтного файла необходимо выполнить команду `git add`. Индексирование для системы Git будет означать, что все конфликты разрешены (рисунок 2.26):

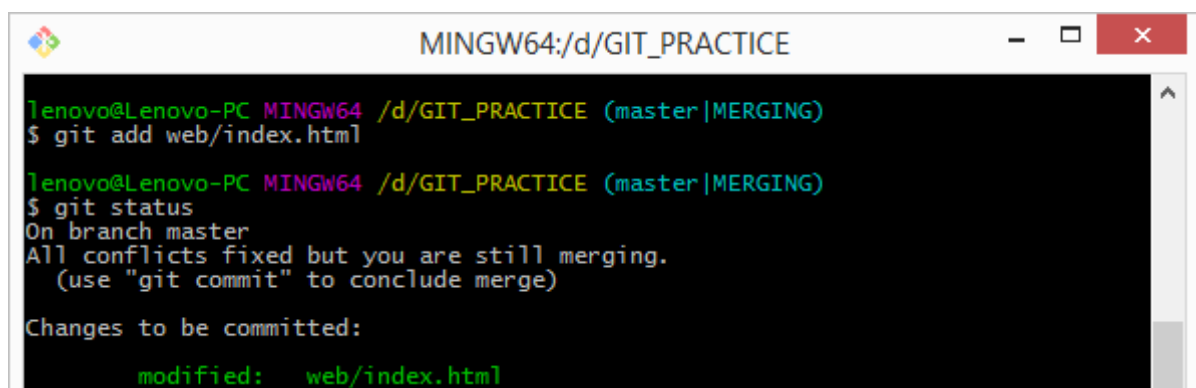
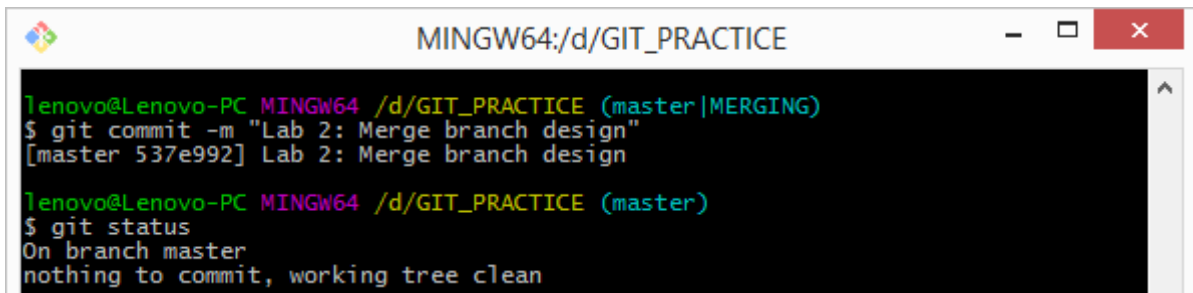


Рисунок 2.26

Удостоверившись, что все файлы, имевшие конфликты, были проиндексированы, можно выполнить `git commit` (рисунок 2.27):



```
lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master|MERGING)
$ git commit -m "Lab 2: Merge branch design"
[master 537e992] Lab 2: Merge branch design

lenovo@Lenovo-PC MINGW64 /d/GIT_PRACTICE (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Рисунок 2.27

В комментарии к коммиту рекомендуется указывать информацию о том, как был разрешен конфликт, если это не очевидно и может быть полезно для других пользователей в будущем.

Требования к отчету:

- 1) кратко описать основные этапы выполнения лабораторной работы, типы разработанных с помощью языка UML моделей, использованные команды системы управления версиями Git;
- 2) привести сформированный список пользовательских историй, созданные диаграммы и их краткое описание, результаты выполнения требуемых команд в командной строке системы Git;
- 3) продемонстрировать полученные результаты в виде списка пользовательских историй, набора диаграмм на языке UML, а также истории коммитов.

Вопросы для самопроверки

1. Что такое фреймворк Scrum? Назначение и основные особенности Scrum.
2. Структура пользовательской истории. Назначение и основные особенности полей, используемых при формировании пользовательских историй.

3. Что такое сценарий использования? Назначение и основные особенности сценариев использования.

4. Каким образом сценарии использования изображаются в языке UML? Виды отношений и их назначение.

5. Что такое С-требования и D-требования, в чем их различие и как они связаны между собой?

6. Назначение и основные особенности диаграммы деятельности в языке UML.

7. Назначение и основные особенности диаграммы последовательности в языке UML.

8. Назначение и основные особенности диаграммы классов в языке UML.

9. Каким образом структурные компоненты программной системы и связи между ними изображаются в языке UML?

10. Каким образом связываются компоненты?

11. Каким образом демонстрируется связь внешнего контракта компонента с реализацией этого поведения внутренними компонентами?

12. Каким образом физическое развертывание артефактов на узлах изображается в языке UML?

13. Существующие типы узлов, их назначение и основные особенности.

14. В чем заключается отличие артефактов от узлов? Приведите примеры узлов и артефактов.

15. При помощи какой команды можно создать новую ветку в системе Git?

16. Какая команда в системе Git используется для перехода между ветками?

17. Для чего используется ключ -b команды git checkout?

18. Для чего используется команда git merge?

19. Каким образом можно удалить ветку в системе Git?
20. По каким причинам могут возникнуть конфликты при слиянии веток в системе Git?
21. Каким образом можно просмотреть список файлов, не прошедших слияние?
22. Как система Git «помогает» разрешить конфликты в файлах, не прошедших слияние?
23. Что необходимо сделать для завершения процесса слияния веток после того, как все конфликты были разрешены?
24. Как избежать возникновения конфликтов при слиянии веток?