Министерство науки и высшего образования российской федерации
Федеральное государственное автономное образовательное
учреждение высшего образования
«Южно-Уральский государственный университет
(национальный исследовательский университет)»
Институт естественных и точных наук
Кафедра прикладной математики и программирования

Современные технологии разработки программных систем искусственного интеллекта

Методические указания по выполнению лабораторных работ студентов бакалавриата «Прикладная математика и искусственный интеллект» по направлению 01.03.02 «Прикладная математика и информатика»

2023

Содержание

1. Цели и задачи дисциплины

Цели преподавания дисциплины:

- познакомить бакалавров с основными особенностями и проблемами разработки программного обеспечения;
- сформировать представление о современных тенденциях развития разработки ПО;
- изучить методические основы создания современных программных систем искусственного интеллекта;
- изучить требования предъявляемые к современным технологиям создания программного обеспечения;
- познакомить с технологиями создания ПО ведущих компаний в области разработки программных продуктов, использующих искусственный интеллект.

Задачи изучения дисциплины:

- познакомить студентов с современными технологиями разработки программных систем искусственного интеллекта;
- познакомить современными подходами к выполнению основных технологических операций;
- подготовить к командной работе над программными системами искусственного интеллекта.

2. Методы и формы организации обучения

Для успешного освоения дисциплины применяются различные образовательные технологии, которые обеспечивают достижение планируемых результатов обучения согласно основной образовательной программе, с учетом требований к объему занятий в интерактивной форме.

Для контроля освоения компетенций используются следующие формы контроля: защита лабораторных работ, опрос по изучаемым разделам дисциплины, тесты.

3. Место дисциплины в структуре ООП

Дисциплина «Современные технологии разработки программных систем искусственного интеллекта» относится к вариативной части профессионального цикла ООП.

Успешное овладение дисциплиной предполагает предварительные знания таких дисциплин как «Основы защиты данных в интеллектуальных системах», «Компьютерные сети», «Объектно-ориентированное программирование» в объеме, предусмотренном направлением 01.03.02 – «Прикладная математика и информатика».

4. Лабораторные работы

Лабораторные работы дисциплины "Современные технологии разработки программных систем искусственного интеллекта" позволяет получить практические навыки использования изучаемых структур данных и эффективных алгоритмов решения различных задач.

Порядок выполнения лабораторных работ

1) изучить теоретический материал по теме лабораторной работы;
2) выполнить лабораторную работу для заданного варианта задания;
3) показать результаты преподавателю;
4) ответить на контрольные вопросы по лабораторной работе.

4.1 Проведение интервью с заказчиком

Цель лабораторной работы – изучение основных способов выявления потребностей и требований.

Теория

Интервью является способом выявления проблем и потребностей.

Нужно различать проблему и потребность

Проблема - это разница между желаемым и воспринимаемым.

Потребность - это отражение некой личной, рабочей или бизнес-проблемы, решение которой оправдывает замысел, покупку или использование новой системы.

Функция (решение) — обслуживание, предоставляемое системой для выполнения одной или нескольких потребностей заказчика.

Анализ проблемы - это процесс осознания реальных проблем и потребностей пользователя и предложение решений для удовлетворения этих потребностей.

Пример проблемы:

Товар не продается

Товар не продается, так как у него низкое качество (уточнение причины проблемы)

Пример потребности:

Нужно находить товар с приемлимым соотношением цена/качество

Важно задавать контекстно-свободные вопросы, которые не навязывают конкретного решения, а позволяют выслушать заказчика, узнать о его проблемах.

Рекомендуется подготовить набор вопросов заранее.

Кратко записывайте ответы в блокнот (не на диктофон!)

Не страшно, если отклонитесь от сценария.

Не перебивайте, пока заказчик красочно описывает текущие проблемы, уточняющие вопросы можно задать потом.

После интервью, пока его данные еще свежи в вашей памяти, зафиксируйте три потребности или проблемы с наивысшими приоритетами, выявленные вами в беседе с данным заказчиком (пользователем).

Методические рекомендации

Студенты разделяются на группы из 3-4 человек и выбирается предметная область, например, обучение иностранному языку.

В каждой группе 1-2 заказчика, являющиеся "экспертами" в выбранной предметной области и 2 аналитика.

Для заказчика

- Выбрать тему, в которой вы разбираетесь, связанную с автоматизацией
- Описывать проблемы и потребности, но не решения
- Придумать должность

Для аналитика

- Подготовить набор вопросов заранее (см. план интервью, необходимо задать не менее 1 вопроса из каждого раздела).
- Кратко записывайте ответы на листке.

План интервью

I. Определение профиля заказчика

Имя, компания, должность (может быть выяснено заранее)

Каковы ваши обязанности?

Кому необходимы результаты вашей деятельности?

Что считается успехом в вашей деятельности?

Какие проблемы влияют на успешность вашей деятельности?

Какие тенденции, если такие существуют, делают вашу работу проще или сложнее?

II. Оценка проблемы

Для каких проблем (прикладного типа) вы ощущаете нехватку хороших решений?

Назовите их. (Не забывайте спрашивать: "А ещё?")

Для каждой проблемы выясняйте следующее.

- Почему существует эта проблема?
- Как она решается в настоящее время?
- Как заказчик (пользователь) хотел бы ее решать?

III. Понимание пользовательской среды

Кто такие пользователи?

Какое у них образование?

Каковы их навыки в компьютерной области?

Имеют ли пользователи опыт работы с данным типом приложений?

Какая платформа используется? Каковы ваши планы относительно будущих платформ?

Используются ли дополнительные приложения, которые имеют отношение к данному приложению? Если да, то пусть о них немного расскажут.

Каковы ожидания заказчика относительно удобства продукта?

Сколько времени необходимо для обучения?

В каком виде должна быть представлена справочная информация для пользователя (в интерактивном или в виде печатной копии)?

IV. Резюме (перечисляются основные пункты, чтобы проверить, все ли правильно вы поняли)

Итак, вы сказали мне ... (перечислите описанные заказчиком проблемы своими словами)

Адекватно ли этот список представляет проблемы, которые имеются при существующем решении?

Какие ещё проблемы (если такие существуют) вы испытываете?

V. Предположения аналитика относительно проблемы заказчика

(проверенные или непроверенные предположения, те проблемы, которые не были упомянуты)

Какие проблемы, если они есть, связаны с (перечислите все потребности или дополнительные проблемы, которые, по-вашему, может испытывать заказчик или пользователь)

Для каждой из указанных проблем выясните следующее.

- Является ли она реальной?
- Каковы ее причины?
- Как она решается в настоящее время?
- Как бы заказчик (пользователь) хотел ее решать?
- Насколько важно для заказчика (пользователя) решение этой проблемы в сравнении с другими, упомянутыми им?

VI. Оценка предлагаемого вами решения (если это уместно)

(Охарактеризуйте основные возможности предлагаемого вами решения, а потом задайте пользователю следующие вопросы.)

Что, если вы сможете ...

Как вы расцениваете важность этого?

VII. Оценка возможности

Кто в организации нуждается в данном приложении?

Сколько пользователей указанных типов будет использовать его?

Насколько значимо для вас успешное решение?

VIII. Оценка необходимого уровня надежности и производительности, а также потребности в сопровождении

Каковы ваши ожидания относительно надежности?

Какой, по-вашему, должна быть производительность?

Будете ли вы заниматься поддержкой продукта или этим будут заниматься другие?

Испытываете ли вы потребности в поддержке?

Что вы думаете о доступе для сопровождения и обслуживания?

Каковы требования относительно безопасности?

Какие требования относительно установки и конфигурации?

Существуют ли специальные требования по лицензированию?

Как будет распределено программное обеспечение?

Есть ли требования на маркировку и упаковку?

IX. Другие требования

Существуют ли законодательные требования, требования информационной среды, инструкции или другие стандарты, которых необходимо придерживаться?

Нет ли других требований, о которых нам следовало бы знать?

X. Окончание

Существуют ли другие вопросы, которые мне следовало бы вам задать?

Если мне еще понадобится задать вам несколько вопросов, могу ли я вам позвонить?

Будете ли вы принимать участие в обсуждении требований?

Заключение аналитика

После интервью, пока его данные еще свежи в вашей памяти, зафиксируйте три потребности или проблемы с наивысшими приоритетами, выявленные вами в беседе с данным заказчиком (пользователем).

Примеры контрольных вопросов

1. В чем различие между проблемой и потребностью?
2. В чем различие между потребностью и решением?

Критерии оценивания

1. Степень погружения в предметную область: от 0 до 3 баллов в зависимости от достоверность описания выбранной предметной области.
2. Выявлены технологические потребности/проблемы от 0 до 3 баллов, оценивается по заключению аналитика, по 1 баллу за каждую проблему, связанную с разработкой ПО.
3. Есть краткая запись результатов интервью в соответствии с планом: 4 балла, если есть, 0, если нет, 2 при ошибках, неполном выполнении плана интервью.

Итого 10 баллов

4.2 Мозговой штурм

Цель лабораторной работы – изучение основных способов определения решения (функций ПО).

Теория

Функция — обслуживание, предоставляемое системой для выполнения одной или нескольких потребностей заказчика.

Для большой группы участников рекомендуется записывать идеи на небольших листах толстым маркером и прикреплять на доску. Нельзя, чтобы идеи записывал на доске секретарь, так как может получиться затор в творческом процессе.

Правила:

- Не допускается критика или дебаты
- Дайте свободу фантазии
- Генерируйте как можно больше идей
- Переделывайте и комбинируйте чужие идеи

Нельзя говорить "это дурацкая идея", "это невозможно", "это уже было" или "почти то же самое", но можно и нужно хвалить идеи для стимуляции творческого процесса. Наличие "сумасбродных" идей является индикатором качества процесса. Возможно, кто-то найдет рациональное зерно в такой идее и предложит более реалистичный упрощенный вариант.

Генерация идей происходит, пока перерывы в появлении идей не станут слишком большими.

Методические рекомендации

Роли аннулируются, все участники группы из 3-4 человек могут рассматривать проблему с любой точки зрения, как эксперта в этой области, так и разработчика ПО.

Каждый участник должен записывать свои идеи своими словами на листке бумаги и озвучивать их, если в этот момент не говорит кто-то другой.

Количество идей не ограничивается 5, хотя баллы начисляются только за первые 5 идей, связанных с компьютерами и ПО, поэтому можно сгенерировать 10-20 и более, и не только из области компьютеров или реалистичные, так как все идеи будут использованы на ЛР 3.

Список идей отдается преподавателю для оценки, на каждом листке должна быть написана фамилия, инициалы и группа. Листки возвращаются на следующем занятии для выполнения ЛР3.

Примеры контрольных вопросов

1. Почему "сумасбродные" идеи тоже нужно озвучивать?
2. Почему нельзя критиковать идеи?

Критерии оценивания

1. Участвовал в мозговом штурме: 3 балла, если да, 0, если задание по генерации было выполнено вне группы.
2. Предложил не менее 5 функций: 2 балла, если есть, 0, если менее 5 функций
3. Предложенные функции связаны с искусственным интеллектом и ПО: по 1 баллу за каждую функцию, но не более 5 баллов

Итого 10 баллов

4.3 Подведение итогов мозгового штурма и выбор функций для 1-й версии

Цель лабораторной работы – изучение основных способов определения решения (функций ПО).

Теория

Масштаб проекта определяется:

- набором функций;
- ресурсами;
- время на реализацию.

Если необходимый для реализации требуемых функций объем работ равен имеющимся ресурсам, умноженным на выделенное время, масштаб проекта является достижимым и проблем не возникает.

Наша задача в том, чтобы применить ограниченные ресурсы с наибольшей выгодой для заказчика.

Базовый уровень – это разбитое на элементы множество функций или требований, которые намечено реализовать в конкретной версии приложения.

Базовый уровень должен обладать следующими свойствами:

- должен быть приемлемым, как минимум, для заказчика;
- должен иметь разумную вероятность успеха с точки зрения команды разработчиков.

Способ определения:

1. Перечислить функции в порядке приоритета.

2. Определить приблизительный уровень трудозатрат для каждой из предлагаемых функций.

3. Оценить риск, связанный с каждой функцией.

4. Базовый уровень можно ограничить критическими функциями и одной-двумя важными. Включение полезных функций - на усмотрение команды. То, что осталось «за бортом» базового уровня – функции на будущее.

Методические рекомендации

Группе из 3-4 студентов возвращаются списки функций, предложенных ими на предыдущей лабораторной работе.

Сначала отсекаются непродуктивные идеи.

Ведущий зачитывает идею, при необходимости уточняет определение у автора. Если хотя бы один участник высказывается за идею, то она остается на доске. Похожие идеи могут объединяться и прикрепляться на одну кнопку. Можно разделять идеи на доске по темам.

Список идей вносится в таблицу (Excel/Calc) и каждый участник должен отнести идею к одной из трех групп:

- критическая, без этой функции система никому не нужна (9 баллов);
- важная, система без этой функции сильно потеряет, пользователи будут приобретать (использовать) продукт, но могут перейти на конкурирующий продукт, если он предоставит подобную функцию (3 балла);
- полезная, система будет немного удобнее в использовании (1 балл).

К каждой группе нужно отнести ровно 1/3 идей.

Голосование проводится тайно, для этого каждый участник выставляет баллы в собственной (подписанной его ФИО) колонке.

Для подведения итогов колонки открываются, результаты суммируются и список сортируется в порядке уменьшения суммы баллов.

Для определения набора функций для первой версии необходимо учесть трудоемкость, риск и ограничения времени (бюджета) на разработку.

Вместо конкретных значений можно использовать качественные оценки:

Трудоемкость

- низкая (несколько часов, дней для 1 разработчика)
- средняя (1-2 недели для 1 разработчика)
- высокая (работа для нескольких разработчиков в течении недель)

Риск

- низкий (используются известные технологии, высокая точность оценки трудоемкости)
- средний (необходимо изучение новых технологий, трудоемкость может быть выше в 1.5-2 раза)
- высокий (об технологиях и способах решения неизвестно, трудоемкость может быть выше оценки в несколько раз)

В первую версию включаем функции из верхней части упорядоченного списка функций, чтобы сумма трудоемкости не превышала заданную с учетом риска и количества разработчиков. Например, срок разработки (2-3 недели) умножается на количество разработчиков (3-4 человека). Функции с высоким риском включаются только, если они являются критическими для системы. Из менее важных функции можно выбрать с низкой трудоемкостью и/или низким риском так, чтобы они дополняли более важные функции, а также приближали суммарную трудоемкость к заданному лимита для первой версии.

Примеры контрольных вопросов

1. Почему голосование проводится тайно?
2. Как определяется трудоемкость функции?

Критерии оценивания

1. Выполнено отсечение, уточнение формулировки - 2 балла, должно остаться не более 20-25 функций и отброшены нереалистичные функций, оценка снижается на 1 балл за каждую ошибку.
2. Каждый участник выполнил оценку важности функций - 2 балла, баллы не начисляются участникам, которые не участвовали в оценке важности функций
3. Выполнена реалистичная оценка трудоемкости - 2 балла, оценка снижается на 1 балл за каждую некорректную оценку более чем в 2 раза
4. Выполнена реалистичная оценка рисков - 2 балла, оценка снижается на 1 балл за каждую некорректную оценку более чем в 2 раза
5. Выполнена сортировка и выбраны функции для 1-й версии - 2 балла, или 0 баллов, если сортировка не выполнена, или суммарная трудоемкость существенно отличается от времени на разработку 1 версии (2-3 недели)

Итого 10 баллов

4.4 Написание спецификации для функциональных и нефункциональных требований

Цель лабораторной работы – изучение способов определения требований к ПО.

Теория

Требование к ПО:

- функциональное требование - свойство ПО, необходимое пользователю для решения проблемы при достижении поставленной цели;
- нефункциональное требование - свойство ПО, которым должна обладать система или её компонент, чтобы удовлетворить требования контракта, стандарта, спецификации или иной формальной документации.

Спецификация требований к ПО - это пакет информации, полностью описывающий внешнее поведение системы, т.е. набор артефактов следующего содержания: «Вот то, что система должна делать, чтобы предоставить эти функции».

Высококачественный пакет спецификаций должен быть:

- корректным;
- недвусмысленным;
- полным;
- непротиворечивым;
- упорядоченным по важности и стабильности;
- поддающимся проверке (верифицируемым);
- модифицируемым;
- трассируемым;
- понимаемым.

Способы описания:

- естественный язык
- диаграммы вариантов использования
- формальные способы:
  - псевдокод;
  - конечные автоматы;
  - деревья и таблицы решений;
  - диаграммы деятельности, BPMN-, EPC-диаграммы (workflow);
  - модели сущность-связь (для данных);
  - объектные диаграммы, диаграммы последовательности (UML);
  - схемы потоков данных (DFD).

Рассмотрим следующую фразу на английском языке «Mary had a little lamb (У Мэри был маленький барашек)». Проблема неоднозначности естественного языка может проанализирована тремя методами:

- Метод ключевых слов (по словарю):

  - to have - иметь/ получать/ обмануть/ владеть/ приобретать/ терпеть/ иметь с собой/ взять в жёны (мужья)/ победить/ иметь в составе/ принимать (пищу)/ родить

- lamb - барашек/ простак/ ягнятина/ агнец/ неопытный игрок на бирже/ овчинная шкура

- Эвристика запоминания - часть, которую сложно вспомнить, является непонятной
- Метод ударения: Mary (У Мэри, а не Алисы) had (теперь барашка нет) a (один, а не стадо) little (карликовый?) lamb (барашек, а не поросёнок или баран)

Методические рекомендации

Задание выполняется индивидуально, каждый участник выбирает одну из функций, вошедшую в список на предыдущей лабораторной работы, и одно из 5 нефункциональных требований: практичность, надежность, производительность, возможность сопровождения, ограничения проектирования.

Функциональные требования

Функция	Спецификация
Функция 63. Система обнаружения неполадок будет предоставлять информацию об обнаруженных дефектах, чтобы помочь пользователю оценить состояние проекта	SR63.1. Информация будет предоставляться в виде отчета-гистограммы, где по оси X откладывается время, а по оси Y — количество обнаруженных дефектов SR63.2. Пользователь может задавать временной период в днях, неделях или месяцах SR63.3. Пример отчета об обнаруженных дефектах представлен на прилагаемом рисунке

Количество пунктов в спецификации не ограничивается 3, а определяется критерием полноты спецификации. Также каждый пункт должен определять один тестовый набор или проверяемое вручную условие. SR63.2 в примере это не три отдельных условия, так как выбор реализуется комбо-боксом, в котором при проверке должны выявлены быть указанные три варианта.

Нефункциональные требования (выбирается один из пунктов и указываются только характеристики, соответствующие предметной области, которым вы можете дать оценку). Нефункциональные требования задаются не для одной функции, а для системы в целом или одной из подсистем (например, веб-сервер)

1. Практичность

Как правило, указывается следующее.

- Время, необходимое для обучения рядовых пользователей и пользователей с большими полномочиями, чтобы они научились эффективно выполнять определенные действия.

- Время выполнения типичных задач; или же практичность новой системы, сравнивается с практичностью известных систем, которые пользователь знает и любит.

- Требования соответствия общепринятым стандартам практичности, таким как CUA IBM или опубликованные компанией Microsoft стандарты GUI для системы Windows 98.

2. Надежность

К характеристикам надежности относятся.

- Доступность. Указывается, какой процент времени система доступна (xx.xx %), определяются часы использования и доступа для обслуживания, операции при ухудшении параметров системы и т.д.

- Среднее время между отказами (mean time between failures, MTBF). Обычно выражается в часах, но может указываться в днях, месяцах и годах.

- Среднее время восстановления (mean time to repair, MTTR) Сколько времени система может находиться в нерабочем состоянии после сбоя.

- Точность. С помощью некоего известного стандарта указывается требуемая точность (разрешающая способность) выводимой системой информации.

- Максимально допустимый коэффициент ошибок и дефектов. Как правило, выражается как число ошибок, приходящееся на K.LOS (тысячу строк кода), или число ошибок, приходящихся на отдельную функцию.

- Доля ошибок или дефектов различных типов. Обычно ошибки разбиваются на следующие категории: незначительные, серьезные и критические. Требования должны определять, что понимается под "критической" ошибкой (такой, как полная потеря данных или невозможность использовать определенную часть функциональных возможностей системы).

3. Производительность

Здесь описываются характеристики производительности системы. Следует указать время ответа для различных ситуаций. Если требуется, указываются названия соответствующих вариантов использования.

- Время ответа для транзакции (среднее, максимальное)

- Пропускная способность (транзакций в секунду)

- Емкость (число пользователей или транзакций, которые может обслужить система)

- Режимы снижения производительности (допустимые режимы работы при ухудшении параметров системы)

- Использование ресурсов (память, диск, каналы связи)

4. Возможность сопровождения

Указываются требования, способствующие улучшению возможности сопровождения и обслуживания создаваемой системы, в том числе стандарты кодирования, определенные соглашения, библиотеки классов, доступ для обслуживания и вспомогательные обслуживающие программы.

5. Ограничения проектирования

Возможные источники: экономические, политические, технические, системные, эксплуатационные, на разработку.

Примеры контрольных вопросов

1. Каким критериям дожна удовлетворять спецификация?
2. Что такое верифицируемость?

Критерии оценивания

1. Написана спецификация для функционального требования - 2 балла или 0 баллов, если отсутствует
2. Написана спецификация для нефункционального требования - 2 балла или 0 баллов, если отсутствует
3. Спецификация является полной - 2 балла или 0 баллов, если есть существенная необходимость в уточнениях для начала разработки
4. Спецификация является верифицируемой - 2 балла, оценка снижается на 1 балл за каждый пункт спецификации, который невозможно проверить
5. Спецификация является понимаемой и недвусмысленной - 2 балла, оценка снижается на 1 балл за каждый пункт спецификации с неправильным термином или ошибкой в формулировке.

Итого 10 баллов

4.5 Проектирование архитектуры системы искусственного интеллекта с использованием UML

Цель лабораторной работы – изучение способов определения архитектуры ПО.

Теория

Для сложных систем выполняется процесс декомпозиции до достижения следующих результатов:

- распределение и разбиение функций оптимизировано так, чтобы добиться достижения общей функциональности системы с минимальными затратами и максимальной гибкостью;
- каждая подсистема может быть определена, спроектирована и построена небольшой или средней командой;
- каждая подсистема может быть изготовлена в рамках физических ограничений и существующих технологий;
- каждую подсистему можно надежно протестировать, причем существует способ имитации интерфейсов с другими подсистемами;
- размеры и распределение подсистем выполнено исходя из общесистемных соображений.

Проектирование - итерационный процесс, при помощи которого требования к ПО транслируются в инженерные представления ПО. Вначале эти представления дают только концептуальную информацию (на высоком уровне абстракции), последующие уточнения приводят к формам, которые близки к текстам на языках программирования.

Обычно в проектировании выделяют две ступени: архитектурное проектирование и детальное проектирование.

Архитектурное проектирование формирует абстракции высокого уровня, детальное проектирование уточняет эти абстракции, добавляет подробности алгоритмического уровня.

Также выделяют проектирование интерфейса, цель которого сформировать интерфейс пользователя.

Этапы архитектурного проектирования:

1. Структурирование системы. Программная система структурируется в виде совокупности относительно независимых подсистем. Также определяются взаимодействия между подсистемами.

2. Моделирование управления. Разрабатывается базовая модель управления взаимоотношениями между частями системы.

3. Модульная декомпозиция. Каждая определенная на первом этапе подсистема разбивается на отдельные модули.

Здесь определяются типы модулей и типы их взаимосвязей.

Подсистема — это система, операции которой не зависят от сервисов, предоставляемых другими подсистемами. Подсистемы состоят из модулей и имеют определенные интерфейсы, с помощью которых взаимодействуют с другими подсистемами.

Модуль - это обычно компонент системы, который предоставляет один или несколько сервисов для других модулей. Модуль может использовать сервисы, поддерживаемые другими модулями. Как правило, модуль никогда не рассматривается как независимая система. Модули обычно состоят из ряда других, более простых компонентов.

Диаграмма компонентов показывает разбиение программной системы на структурные компоненты и связи (зависимости) между компонентами. В качестве физических компонентов могут выступать файлы, библиотеки, модули, исполняемые файлы, пакеты и т. п.

Для компонента используется нотация, показанная на рисунке 1.

Рисунок 1 — Обозначение компонента в UML.

Диаграмма развёртывания служит для моделирования вычислительных узлов и артефактов, развёрнутых на них (рисунок 2).

Методические рекомендации

Задание выполняется в группе из 2 человек. Задание: уарисовать диаграмму компонентов/размещения UML для архитектуры системы с использованием искусственного интеллекта.

Примеры контрольных вопросов

1. Каким символом изображается компонент?

2. Что является компонентом?

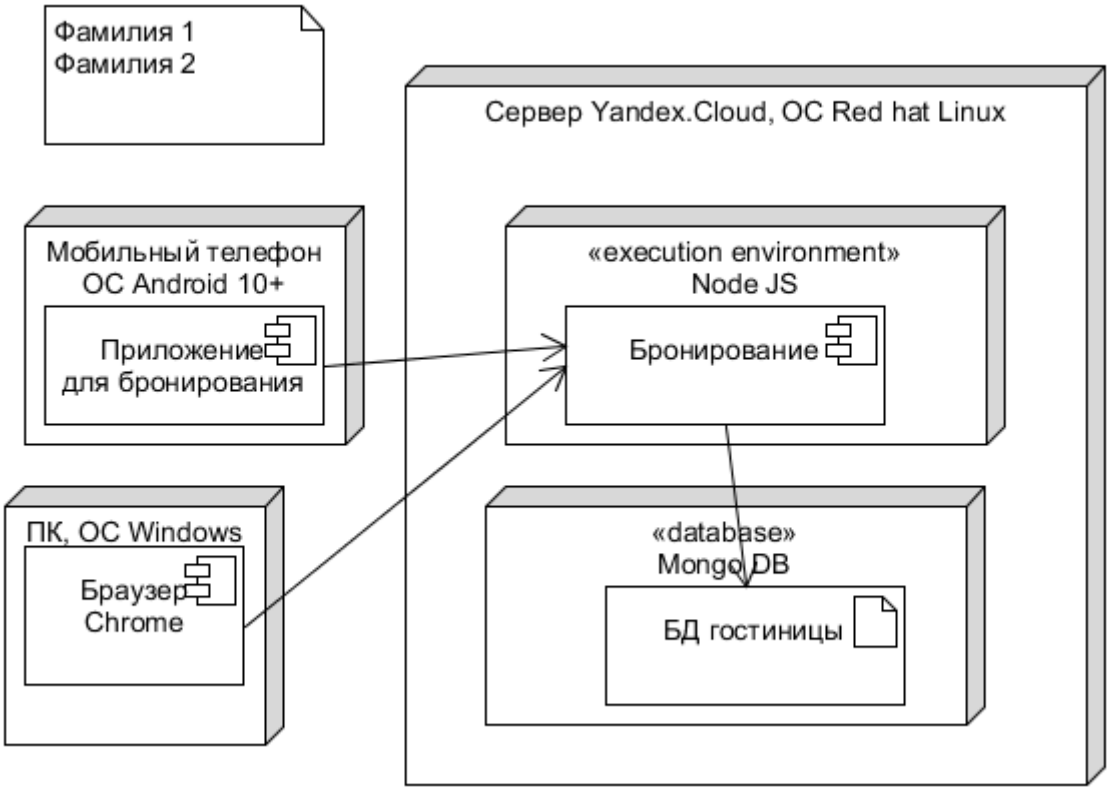


Рисунок 2 — Пример диаграммы развертывания UML.

Критерии оценивания

1. Диаграмма нарисована с помощью средств рисования диаграмм UML - 2 балла, или 0 баллов, если использованы не соответствующие инструменты
2. Выбраны правильные обозначения UML для всех элементов диаграммы - 4 балла, оценка снижается на 1 балл за каждую ошибку
3. Диаграмма является полной - 2 балла, или 0 баллов, если важный компонент был пропущен
4. Выбраны компоненты, соответствующие предметной области- 2 балла, оценка снижается на 1 балл за каждую ошибку

Итого 10 баллов

4.6 Разработка диаграммы классов UML для системы искусственного интеллекта

Цель лабораторной работы – изучение объектно-ориентированного подхода к разработке ПО

Теория

Диаграмма классов демонстрирует общую структуру иерархии классов системы, их атрибутов (полей), методов, интерфейсов и взаимосвязей между ними.

Для классов и связей между ними используются обозначения на рисунке 3.

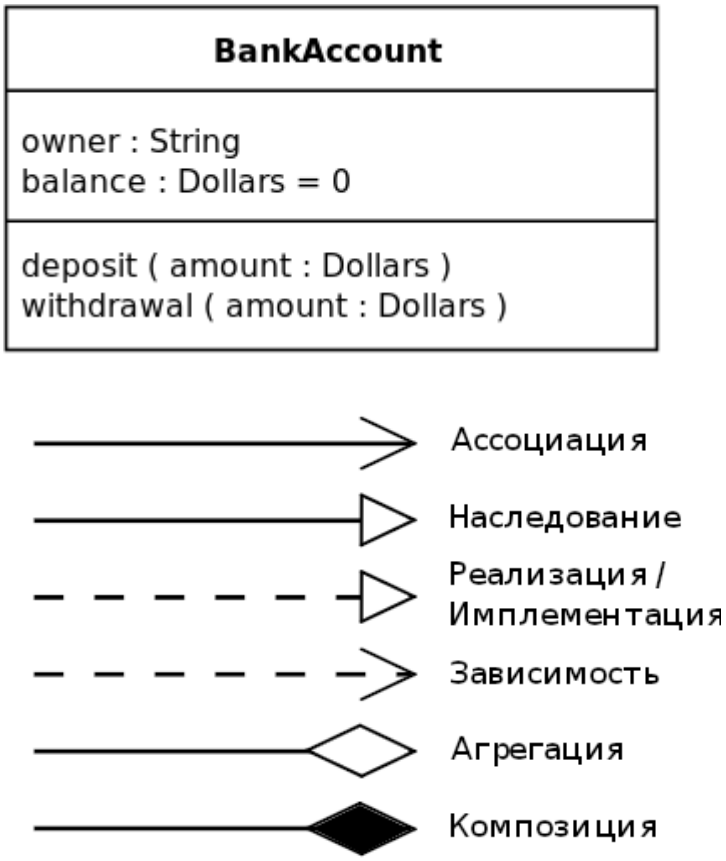


Рисунок 3 — Обозначения UML для классов и связей.

В начале проектирования или на обзорной диаграмме класс может представлен так:

По мере уточнения определения в классе добавляются поля и методы, сначала просто имена, затем остальные характеристики.

Для поля: видимость имя:тип[кратность]= нач_значение {ограничение}

Для метода: видимость имя(параметры):тип_результата {ограничение}

где видимость + для public, - для private, # - для protected

Общие элементы класса (static) выделяются подчеркиванием, абстрактные — курсивом.

Методические рекомендации

Нарисовать диаграмму классов UML для системы искусственного интеллекта. Задание выполняется в группе из 2 человек

Пример выполнения (рисунок 4):

Цифровой диктофон – это бытовое электронное устройство, предназначенное для записи и воспроизведения речи. Звуковые сообщения записываются через встроенный микрофон и сохраняются в памяти

устройства. Сообщения воспроизводятся через встроенный громкоговоритель.

Диктофон хранит до 99 звуковых сообщений. Длина каждого сообщения ограничена размером свободной памяти. Диктофон осуществляет прямой (по номеру сообщения) доступ к любому сообщению из памяти. Пользователь имеет возможность воспроизводить сообщения, хранящиеся в памяти диктофона, стирать их, записывать новые. Интерфейс с пользователем осуществляется при помощи экранного меню и управляющих кнопок на корпусе диктофона. При помощи 4 кнопок-стрелок осуществляется навигация по пунктам меню. Кнопки «ОК», «Отмена» служат для подтверждения или отмены пользователем выбора той или иной опции меню (структуру меню нужно разработать самостоятельно). Имеются также кнопки «Воспроизведение», «Пауза» и «Запись» для работы со звуковыми сообщениями.

Во время записи сообщения на экране отображается время, в течение которого ведется запись, при воспроизведении – длительность воспроизведенной части сообщения.

Если диктофон не используется, через 30 секунд он автоматически переходит в режим сбережения энергии. В этом режиме никакие операции над звуковыми сообщениями не возможны. Энергия расходуется только на сохранение памяти диктофона в неизменном состоянии. Переход из режима сбережения энергии в обычный режим осуществляется при нажатии пользователем любой кнопки.

В диктофоне имеется датчик уровня заряда батарей. При падении уровня заряда ниже установленного предела диктофон автоматически переходит в режим сбережения энергии. Переход в обычный режим становится возможным только после восстановления нормального уровня заряда батарей.

Примеры контрольных вопросов

1. Какие элементы записываются во втором разделе символа класса?
2. Как обозначается связь наследования?

Критерии оценивания

1. Диаграмма нарисована с помощью средств рисования диаграмм UML - 2 балла, или 0 баллов, если использованы не соответствующие инструменты
2. Выбраны правильные обозначения UML для всех элементов диаграммы и связей между ними - 3 балла, оценка снижается на 1 балл за каждую ошибку
3. Правильно указаны поля классов, их типы и методы - 3 балла, оценка снижается на 1 балл за каждую ошибку
4. Диаграмма является полной - 2 балла, или 0 баллов, если важный элемент, описанный в задании, был пропущен

Итого 10 баллов

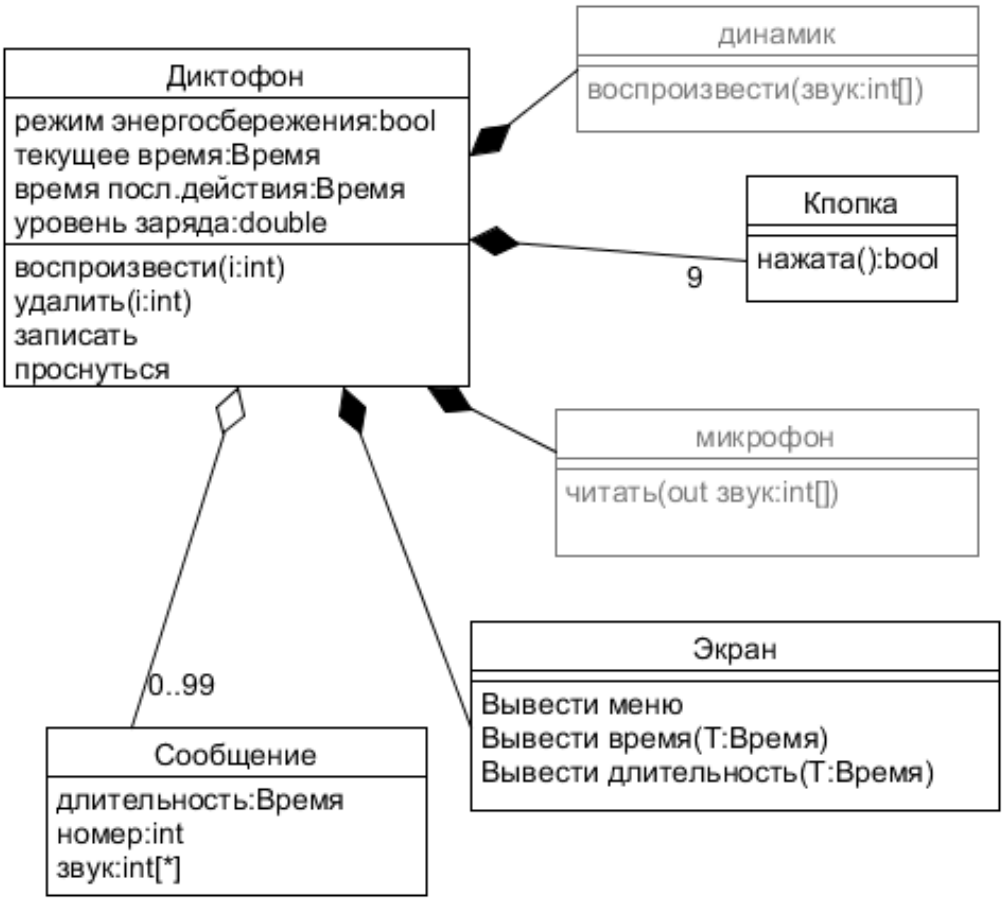


Рисунок 4 — Диаграмма классов для примера.

4.7 Разработка компонент системы искусственного интеллекта

Цель лабораторной работы – изучение способов детального проектирования ПО, разработки спецификации модуля или компонента.

Методические рекомендации

Задание выполняется в группе из 2 человек. Разработать компонент «Выбор блюд в ресторане» с использованием искусственного интеллекта. Для этого выбрать средства для разработки компонента, указать входные данные для построения модели, указать способ интеграции интеллектуальной компоненты в систему.

Примеры контрольных вопросов

1. Как оценить влияние входных данных?
2. Как проводить тестирование компонента с использованием искусственного интеллекта?

Критерии оценивания

1. Выбраны правильные средства для разработки компонент - 3 балла, оценка снижается на 1 балл за каждую ошибку
2. Правильно указаны входные данные для построения модели - 3 балла, оценка

снижается на 1 балл за каждую ошибку
3. Определен способ интеграции интеллектуальной компоненты в систему - 2 балла, или 0 баллов, если способ не указан
4. Выполнена автономная проверка компоненты на тестовых данных - 2 балла, или 0 баллов, если тестирование не выполнялось

Итого 10 баллов

4.8 Разработка через тестирование

Цель лабораторной работы – изучение метода разработки через тестирование с использованием современных сред разработки и библиотек.

Теория

Разработка через тестирование (test-driven development, TDD) — техника разработки программного обеспечения, которая основывается на повторении очень коротких циклов разработки: сначала пишется тест, покрывающий желаемое изменение, затем пишется код, который позволит пройти тест, и под конец проводится рефакторинг нового кода к соответствующим стандартам. Разработка через тестирование требует от разработчика создания автоматизированных модульных тестов, определяющих требования к коду непосредственно перед написанием самого кода.

Тест содержит проверки условий, которые могут либо выполняться, либо нет. Когда они выполняются, говорят, что тест пройден. Прохождение теста подтверждает поведение, предполагаемое программистом.

Разработчики часто пользуются библиотеками для тестирования для создания и автоматизации запуска наборов тестов (NUnit, JUnit/CppUnit, gtest, Boost.Test)

Цикл разработки через тестирование

1. Добавление теста

При разработке через тестирование добавление каждой новой функциональности в программу начинается с написания теста.

Неизбежно этот тест не будет проходить, поскольку соответствующий код ещё не написан. Если же написанный тест прошёл, это означает, что либо предложенная «новая» функциональность уже существует, либо тест имеет недостатки.

2. Запуск всех тестов: убедиться, что новые тесты не проходят

На этом этапе проверяют, что только что написанные тесты не проходят. Этот этап также проверяет сами тесты: написанный тест может проходить всегда и соответственно быть бесполезным.

Новые тесты должны не проходить по объяснимым причинам. Это увеличит уверенность (хотя не будет гарантировать полностью), что тест действительно тестирует то, для чего он был разработан.

3. Написать код

На этом этапе пишется новый код так, что тест будет проходить. Этот код не обязательно должен быть идеален.

Допустимо, чтобы он проходил тест каким-то неэлегантным способом. Это приемлемо, поскольку последующие этапы улучшат и отполируют его.

Важно писать код, предназначенный именно для прохождения теста. Не следует добавлять лишней и, соответственно, не тестируемой функциональности.

4. Запуск всех тестов: убедиться, что все тесты проходят

Если все тесты проходят, программист может быть уверен, что код удовлетворяет всем тестируемым требованиям.

После этого можно приступить к заключительному этапу цикла.

5. Рефакторинг

Когда достигнута требуемая функциональность, на этом этапе код может быть почищен. Рефакторинг — процесс изменения внутренней структуры программы, не затрагивающий её внешнего поведения и имеющий целью облегчить понимание её работы, устранить дублирование кода, облегчить внесение изменений в ближайшем будущем.

6. Повторить цикл

Описанный цикл повторяется, реализуя всё новую и новую функциональность.

Шаги следует делать небольшими, от 1 до 10 изменений между запусками тестов. Если новый код не удовлетворяет новым тестам или старые тесты перестают проходить, программист должен вернуться к отладке.

Возьмём задачу преобразования арабских чисел в римские и напишем 1-й тест в файле roman_unittest.cpp:

```
#include "gtest/gtest.h"
#include "roman.hpp"
TEST(Roman, III) {
  EXPECT_EQ("I", int2roman(1));
}
```

где roman.hpp содержит строки:

```
#include <string>
std::string int2roman(int n);
```

Начальная реализация функции в файле roman.cpp:

```
#include "roman.hpp"
std::string int2roman(int n) {
   return "";
}
```

Первый запуск дает ошибку:

```
[ RUN      ] Roman.III
roman_unittest.cpp:4: Failure
Expected equality of these values:
  "I"
  int2roman(1)
     Which is: ""
[  FAILED  ] Roman.III (0 ms)
```

Для исправления ошибки заменим return "" на return "I" и снова запустим тест:

```
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from Roman
[ RUN      ] Roman.III
[       OK ] Roman.III (0 ms)
[----------] 1 test from Roman (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (0 ms total)
[  PASSED  ] 1 test.
```

Тест пройден. Добавим новый тест: EXPECT_EQ("II", int2roman(2)); и получим снова ошибку.

Исправляем код функции на следующий.

```
std::string int2roman(int n) {
  if(n==2) return "II";
  return "I";
}
```

После прохождения теста выполняем рефакторинг - одна точка выхода из функции:

```
std::string int2roman(int n) {
  std::string res="I";
  if(n==2) res="II";
  return res;
}
```

Добавляем новый тест:

```
TEST(Roman, III) {
 EXPECT_EQ("I", int2roman(1));
 EXPECT_EQ("II", int2roman(2));
 EXPECT_EQ("III", int2roman(3));
}
```

и снова изменяем код:

```
std::string int2roman(int n) {
  std::string res="I";
  if(n==2) res="II";
  else if(n==3) res="III";
  return res;
}
```

На шаге рефакторинга код улучшается с помощью цикла:

```
std::string int2roman(int n) {
  std::string res;
  while(n-- > 0)
    res+='I';
  return res;
```

```
}
```
Далее добавляем группу тестов для 10 и 20:
```
TEST(Roman, X) {
  EXPECT_EQ("X", int2roman(10));
  EXPECT_EQ("XX", int2roman(20));
}
```
Снова изменяем код, добавляя второй цикл для 10-к.
```
std::string int2roman(int n) {
   std::string res;
   while(n >= 10) {
     res+='X';
     n-=10;
   }
   while(n-- > 0)
     res+='I';
   return res;
}
```
Улучшаем код, вводя внешний цикл (для 10 и 1) вместо двух отдельных циклов:
```
static vector<char> rdigit{'X','I'};
static vector<int> rvalue{10,1};
std::string int2roman(int n) {
   std::string res;
   for(int i=0; i<rdigit.size(); ++i)
     while(n>=rvalue[i]) {
       res+=rdigit[i];
       n-=rvalue[i];
     }
   }
   return res;
}
```
И так далее.

Методические рекомендации

Разработать функцию, согласно выбранному варианту, методом TDD. История разработки кода сохраняется после добавления каждого теста в папке history как файл solve#.cpp

Начальные файлы

solve.hpp:
```
int solve(параметры);
```
solve.cpp:
```
#include "solve.hpp"
int solve(параметры) {
// решение
}
```

solve_unittest.cpp:

```
#include "solve.hpp"
#include <gtest/gtest.h>
TEST(TestSet, Test1) {
  EXPECT_EQ(solve(аргументы1), результат1);
  EXPECT_EQ(solve(аргументы2), результат2);
}
```

Для проверки полноты тестов можно отправить решение на сайт ipc.susu.ru, добавив main.

```
int main() {
  cin>>параметры;
  cout<<solve(параметры)<<”\n”;
}
```

Примеры контрольных вопросов

1. Что делать, если после изменений перестает работать предыдущий тест?
2. Какое правило рефакторинга для возврата результата?

Критерии оценивания

1. Представлена история модификаций кода - 2 балла, иначе 0 баллов.
2. Предложено не менее 6 тестов - 4 балла, если менее 6, то оценка снижается на (6-количество тестов) баллов
3. Выполнен рефакторинг кода после добавления тестов - 2 балла, иначе 0 баллов
4. Набор тестов полный - 2 балла, иначе 0 баллов.

Итого 10 баллов

4.9 Тестирование модуля как белого ящика

Цель лабораторной работы – изучение основных способов тестирования ПО.

Теория

Стратегия белого ящика, или стратегия тестирования, управляемого логикой программы, позволяет исследовать внутреннюю структуру программы. В этом случае тестирующий получает тестовые данные путем анализа логики программы. Наиболее правильной является сочетание тестирования программы как черного и как белого ящиков.

Минимальным критерием является выполнение каждого оператора программы по крайней мере один раз. Наиболее полным критерием является *комбинаторное покрытие условий*. Он требует создания такого числа тестов, чтобы все возможные комбинации результатов условия в каждом решении выполнялись по крайней мере один раз.

Методические рекомендации

Задание выполняется индивидуально в MinIDE или Visual Studio C++.

Для определения критериев используются пункты меню Тестирование (рисунок 5): «Выполнить тесты для модуля» и «Показать покрытие тестами».

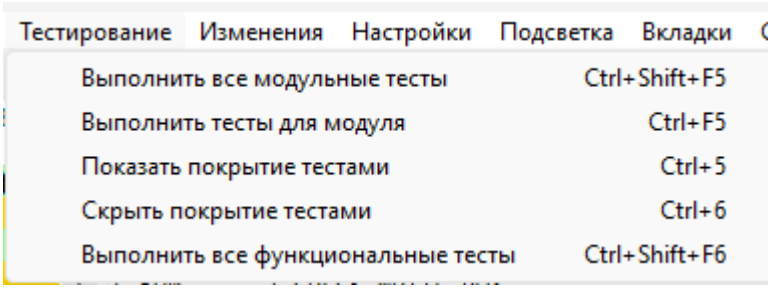


Рисунок 5 — Меню тестирования в MinIDE.

Зеленый и оранжевый цвет на рисунке 6 показывает, что эти операторы были выполнены по крайней мере 1 раз, оранжевый — что критерий комбинаторного покрытия условий для этого оператора ветвления не выполняется.

В окне вывода доступна статистика по обоим критериям.

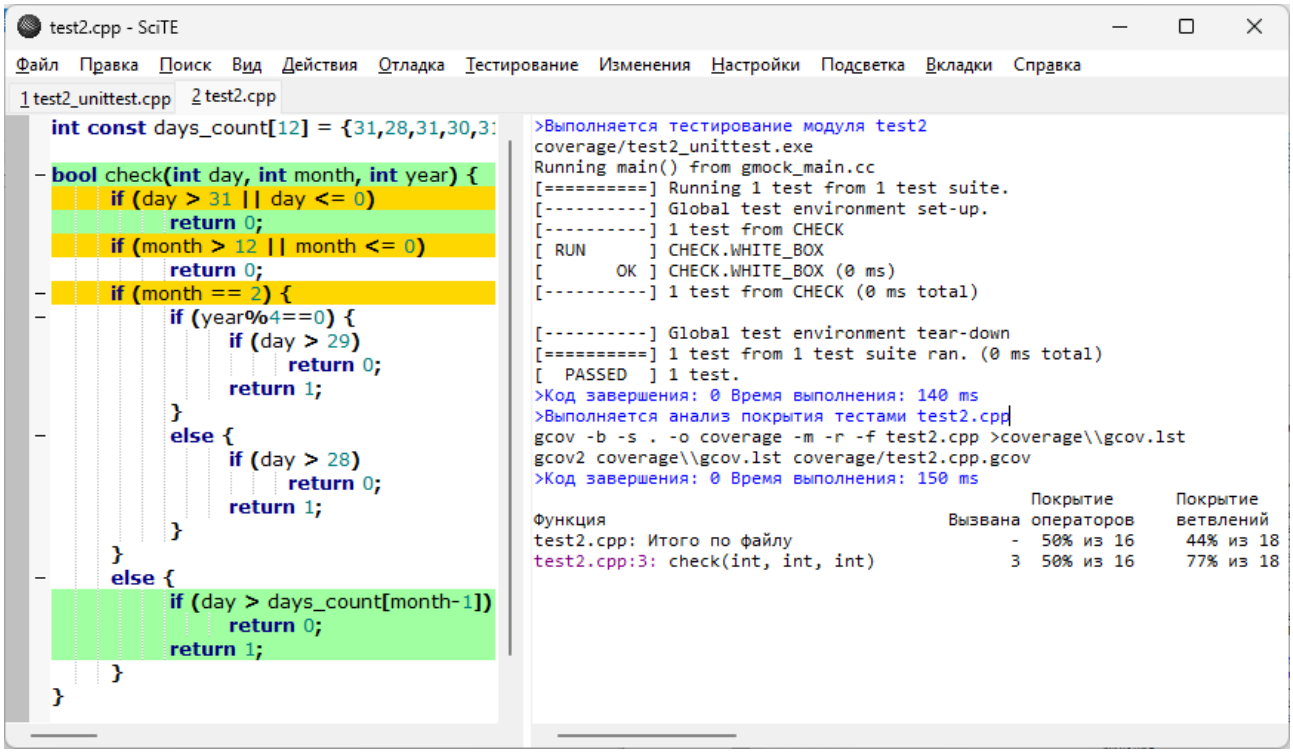


Рисунок 6 — Результаты покрытия тестами в MinIDE.

Результат тестируемой функции в этой лабораторной работе игнорируется, но на практике покрытие кода используется для оценки качества тестов, разработанных в ходе TDD.

```
#include "gtest/gtest.h"
bool check(int day, int month, int year);
TEST(CHECK,WHITE_BOX)
{ check(1,1,1);
  ...
}
int const days_count[12] = {31,28,31,30,31,30,31,31,30,31,30,31};
bool check(int day, int month, int year) {
if (day > 31 || day <= 0) return 0;
```

```
if (month > 12 || month <= 0) return 0;
if (month == 2) {
  if (year%4==0) {
    if (day > 29) return 0;
   return 1;
  }
  else {
    if (day > 28) return 0;
   return 1;
  }
}
else {
  if (day > days_count[month-1]) return 0;
  return 1;
}
}
```

Примеры контрольных вопросов

1. Что означает оранжевый цвет в листинге покрытия?
2. Является ли покрытие операторов достаточным?

Критерии оценивания

1. Выполнено знакомство со средствами проверки покрытия кода - 4 балла, иначе 0 баллов
2. Выполнено покрытие операторов на 100% - 4 балла, иначе 0 баллов
3. Выполнено комбинаторное покрытие условий на 100% - 2 балла, иначе 0 баллов

Итого 10 баллов

4.10 Тестирование модуля как чёрного ящика и тестирование системы

Цель лабораторной работы – изучение основных способов тестирования ПО.

Теория

При использовании этой стратегии программа рассматривается как черный ящик. Тестовые данные используются только в соответствии со спецификацией программы, т.е. без учета знаний о ее внутренней структуре.

В основе методологии тестирования "эквивалентное разбиение" лежат два положения.

Во-первых, каждый тест должен включать столько различных входных условий, сколько это возможно, с тем чтобы минимизировать общее число необходимых тестов.

Во-вторых, необходимо пытаться разбить входную область программы на конечное число классов эквивалентности так, что каждый тест, являющийся представителем некоторого класса, эквивалентен любому другому тесту этого класса.

Иными словами, если один тест класса эквивалентности обнаруживает ошибку, то следует ожидать, что и все другие тесты этого класса эквивалентности будут обнаруживать ту же самую ошибку. Наоборот, если тест не обнаруживает ошибки, то следует ожидать, что ни один тест этого класса эквивалентности не будет обнаруживать ошибки.

Разработка тестов методом эквивалентного разбиения осуществляется в два этапа: 1) выделение классов эквивалентности и 2) построение тестов.

Как показывает опыт, тесты, исследующие *граничные условия*, приносят большую пользу, чем тесты, которые их не исследуют. Граничные условия — это ситуации, возникающие непосредственно на, выше или ниже границ входных и выходных классов эквивалентности.

Человек, обладающий практическим опытом, часто подсознательно применяет метод проектирования тестов, называемый предположением об ошибке. При наличии определенной программы он интуитивно предполагает вероятные типы ошибок и затем разрабатывает тесты для их обнаружения.

Методические рекомендации

Необходимо написать набор тестов, используя методы тестирования программы как черного ящика (эквивалентное разбиение, граничные значения, предположение об ошибке), который выявляет ошибки во всех программах или большей их части.

Примеры контрольных вопросов

1. Что такое граничное значение?
2. Что такое класс эквивалентности?

Критерии оценивания

1. Выполнена разработка тестов для подзадачи 1, определяющих все ошибки в наборе программ - 4 балла, иначе 0 баллов
2. Выполнена разработка тестов для подзадачи 2, определяющих все ошибки в наборе программ - 6 баллов, если в половине программ из набора - 4 балла, иначе 0 баллов

Итого 10 баллов

4.11 Использование библиотек для логирования, локализация ошибки с помощью отладчика в IDE

Цель лабораторной работы – изучение основных способов определения решения (функций продукта).

Теория

Отладка — это процесс локализации и исправления ошибок, обнаруженных при тестировании программного обеспечения.

Локализацией называют процесс определения оператора программы, выполнение которого вызвало нарушение нормального вычислительного процесса. Для исправления ошибки необходимо определить ее причину, т. е. определить оператор или фрагмент кода, содержащие ошибку.

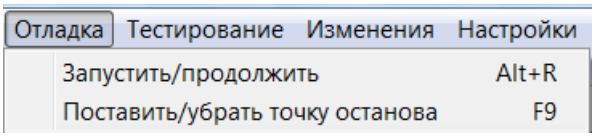
Методы "грубой силы" требуют анализа большого количества данных и игнорируют процесс обдумывания.

При использовании отладки с помощью «расстановки операторы печати по всей программе» требуется изменять программу при отладке; эти изменения могут скрыть ошибку (например, если ошибка связана с неинициализированным указателем), нарушить критические временные отношения. Вместо временной расстановки операторов печати можно на этапе проектирования предусмотреть журналирование событий с помощью библиотеки google glog, Boost.log, или P7 и включать разные уровни журналирования на период бета-тестирования и постоянной эксплуатации.

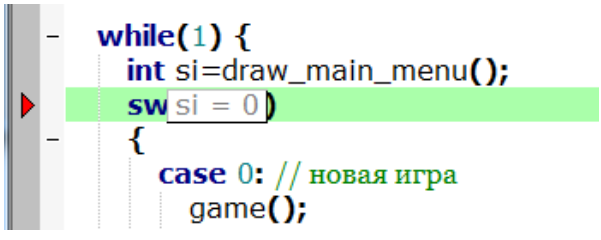
```
#include <glog/logging.h>
int main(int argc, char* argv[]) {
  google::InitGoogleLogging(argv[0]);
  LOG(INFO) << "Start";
  LOG_IF(ERROR, argc<2) << "No arguments";
}
```

Отладка с использованием отладчика также изменяет программу, но эта работа выполняется компилятором (Debug/Release).

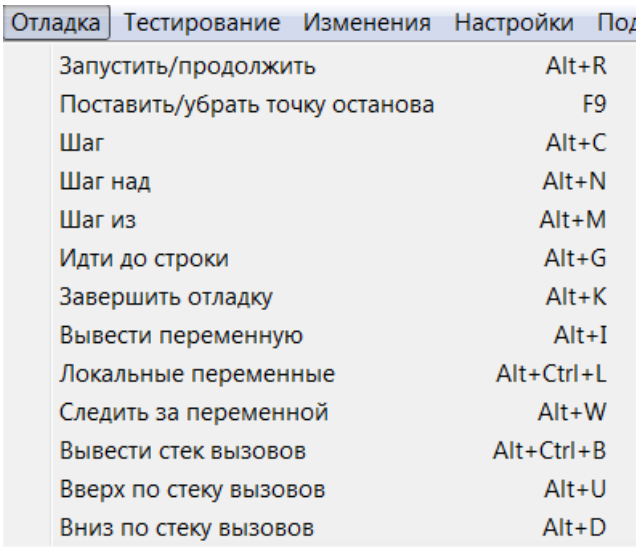
В среде MinIDE можно использовать отладчик. Сначала устанавливается точка останова на строке, непосредственно следующей за той строкой, результаты выполнения которой нужно проанализировать, за с помощью пункта «Поставить точку останова» из меню «Отладка» (рисунок 7а). После запуска программы с помощью пункта меню «Запустить/продолжить» из меню «Отладка» можно посмотреть значения интересующих переменных, подводя к ним курсор (рисунок 7б). Во время выполнения отладки в меню «Отладка» добавляются новые пункты для пошагового выполнения программы, просмотра стека вызовов и состояния локальных переменных в функции (рисунок 7в). Досрочное завершения выполнения программы и отладки завершается с помощью пункта меню «Завершить отладку».



а)



б)



в)

Рисунок 7 — Работа с отладчиком в MinIDE

Методические рекомендации

Имеется тест, выявляющий ошибку в программе. Используя отладчик и/или логирование определите ошибочный оператор. Напишите причину ошибку, не исправляя эти операторы.

```
#include<iostream>
using namespace std;
int main()
{ int n,k,a,c,kol[3]={0};
  char s[3][20100];
  cin>>n>>k;
  for(int i=0;i<k;++i)
  {  cin>>a>>c;
    if(c==8) {
      if(kol[a-1]!=0) kol[a-1]--;
    }
    else if(c==13)
    {  s[a-1][kol[a-1]++]='/';
      s[a-1][kol[a-1]++]='/';
    }
    else
      s[a-1][kol[a-1]++]=c;
  }
  for(int i=0;i<n;++i)
  { s[i][kol[i]]='\0';
    cout<<s[i]<<endl;
  }
}
```

Тест, выявивший ошибку

```
2 7
2 48
2 13
2 49
2 8
2 8
2 8
2 50
```

Ожидаемый ответ

0//2

Примеры контрольных вопросов

1. Как выполнить программу по-операторно?
2. Как посмотреть значение переменной?

Критерии оценивания

1. Выполнен запуск на тесте в отладчике, продемонстрировано способы пошагового выполнения программы, просмотра текущего - 4 балла, иначе 0 баллов
2. Подключена библиотека для логирования, расставлены операции логирования в программе - 2 балла, иначе 0 баллов
3. Локализована ошибка и указана причина - 4 балла, иначе 0 баллов
Итого 10 баллов

4.12 Работа с системой контроля версий

Цель лабораторной работы – получение практических навыков по работе с системой контроля версий.

Теория

Система контроля версий (Version Control System, VCS) представляет собой программное обеспечение, которое позволяет отслеживать изменения в документах, при необходимости производить их откат, определять, кто и когда внес исправления и т.п.

Методические рекомендации

Задание выполняется в группе от 2 до 4 человек

1. Зарегистироваться на githib (все)
2. Создать начальные файлы проекта (один из группы)

SciTE.properties:

```
build.goal=all
```

program.cpp:

```
#include "graphics.h"
#include "picture.hpp"
int main()
{ initwindow(800,600);
  house(); // дом
  sun(); // солнце
  man(); // человек
  fence(); // забор
  getch();
  closegraph();
}
```

picture.hpp:

```
void  house(); // 1 участник house.cpp
void  sun(); // 2 участник sun.cpp
void  man(); // 3 участник man.cpp
void  fence(); // 4 участник fence.cpp
```

.gitignore

```
*.o
*.exe
project.api
```

Через Settings/Manage access предоставить доступ к проекту остальным участникам группы

3. Установить и запустить GithubDesktop (все)

4. Скачать проект

5. Create new branch «название как функция»

6. Добавить файл «имя функции».cpp:

```
#include "graphics.h"
#include "picture.hpp"
void «имя функции» () {
... рисование части картинки
}
```

7. Выполнять цикл Pull request/Merge/Fetch по слиянию всех веток в master

8. Проверить результат

9. Исправить ошибки (размеры, позиция) для получения согласованной картинки в соответствующих ветках

10. Отправить исправления в masterПримеры контрольных вопросов

Примеры контрольных вопросов

1. Почему нельзя изменять в ветке master?

2. Может ли разработчик использовать несколько веток? Для каких целей?

Критерии оценивания

1. Выполнена регистрация на github и подключение к проекту - 2 балла, иначе 0 баллов
2. Получен начальный код и создана собственная ветка - 2 балла, иначе 0 баллов
3. Написан код модуля и выполнено слияние ветки разработчика с основной веткой- 2 балла, иначе 0 баллов
4. Получена основная ветка с изменениями от всех разработчиков и выполнено её слияние с веткой и проверка работы программы - 2 балла, иначе 0 баллов
5. Внесены исправления и выполнено слияние ветки разработчика с основной веткой - 2 балла, иначе 0 баллов

Итого 10 баллов

5. Учебно-методическое и информационное обеспечение дисциплины

5.1 Основная литература

1. Зубкова, Т. М. Технология разработки программного обеспечения : учебное пособие / Т. М. Зубкова. — Оренбург : ОГУ, 2017. — 468 с. — ISBN 978-5-7410-1785-2. — Текст : электронный // Лань : электронно-библиотечная система. https://e.lanbook.com/book/110632

2. Иванов, Д. Моделирование на UML / Д. Иванов, Ф. Новиков. — Санкт-Петербург : НИУ ИТМО, 2010. — 200 с. — Текст : электронный // Лань : электронно-библиотечная система. https://e.lanbook.com/book/40879

5.2 Дополнительная литература

1. Остроух, А. В. Системы искусственного интеллекта / А. В. Остроух, Н. Е. Суркова. — 3-е изд., стер. — Санкт-Петербург : Лань, 2023. — 228 с. — ISBN 978-5-507-46441-8. — Текст : электронный https://e.lanbook.com/book/310199

2. Джесутасан, Р. Реинжиниринг бизнеса: Как грамотно внедрить автоматизацию и искусственный интеллект / Р. Джесутасан ; перевод с английского Е. Милицкая. — Москва : Альпина Паблишер, 2019. — 278 с. — ISBN 978-5-9614-2634-2. — Текст : электронный https://e.lanbook.com/book/140499

3. Ехлаков, Ю. П. Управление программными проектами. Стандарты, модели : учебное пособие для вузов / Ю. П. Ехлаков. — 3-е изд., стер. — Санкт-Петербург : Лань, 2021. — 244 с. — ISBN 978-5-8114-8362-4. — Текст : электронный // Лань : электронно-библиотечная система. https://e.lanbook.com/book/175498

4. Котляров, В. П. Основы тестирования программного обеспечения : учебное пособие / В. П. Котляров. — 2-е изд. — Москва : ИНТУИТ, 2016. — 248 с. — ISBN 5-9556-0027-2. — Текст : электронный // Лань : электронно-библиотечная система. https://e.lanbook.com/book/100352