Федеральное государственное бюджетное образовательное учреждение
высшего образования
«КАЛИНИНГРАДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Н. Я. Великите

ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА НАУКОЁМКОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Учебно-методическое пособие по изучению дисциплины
для студентов магистратуры по направлению подготовки
15.04.04 Автоматизация технологических процессов и производств

Калининград
Издательство ФГБОУ ВО «КГТУ»
2025

УДК 004.4.24(076)

Рецензент
кандидат технических наук, доцент кафедры теории машин и механизмов и деталей машин ФГБОУ ВО «Калининградский государственный технический университет» О. С. Витренко

Великите, Н. Я.
Проектирование и разработка наукоёмкого программного обеспечения: учебно-методическое пособие по изучению дисциплины для студентов магистратуры по направлению подготовки 15.04.04 Автоматизация технологических процессов и производств / Н. Я. Великите – Калининград: Изд-во ФГБОУ ВО «КГТУ», 2025. – 51 с.

Учебно-методическое пособие является руководством по изучению дисциплины «Проектирование и разработка наукоёмкого программного обеспечения» для студентов магистратуры по направлению подготовки 15.04.04 Автоматизация технологических процессов и производств. Содержит характеристику дисциплины, цель и планируемые результаты изучения дисциплины, тематический план с описанием для каждой темы формы проведения занятия, вопросы для изучения, методические материалы к занятиям.
Табл. 2, рис. 6, список лит. – 6 наименований

Учебно-методическое пособие по изучению дисциплины рекомендовано к использованию в учебном процессе в качестве локального электронного методического материала методической комиссией института цифровых технологий 29 апреля 2025 г., протокол № 3

УДК 004.4.24(076)

ОГЛАВЛЕНИЕ

1. Введение…….....…....4

2. Тематический план……..…….....…….....……....5

3. Содержание дисциплины……...…8

4. Методические указания по выполнению лабораторных работ……..44

5. Методические указания по самостоятельной работе…..……45

6. Контроль и аттестация…….……...48

7. Список литературы……..... …….....50

1. ВВЕДЕНИЕ

Учебно-методическое пособие представляет собой материалы для изучения дисциплины «Проектирование и разработка наукоёмкого программного обеспечения» для студентов магистратуры по направлению подготовки 15.04.04 Автоматизация технологических процессов и производств.

Целью освоения дисциплины «Проектирование и разработка наукоёмкого программного обеспечения» является: формирование у обучающихся теоретических знаний и практических навыков, необходимых при создании полноценных программных систем: анализ требований, детального проектирования архитектуры приложения, обеспечения качества.

Задачами преподавания дисциплины, отражающимися в ее содержании, являются: рассмотрение моделей жизненного цикла программного обеспечения; рассмотрение методологии моделирования программного обеспечения. Объектно-ориентированная методология UML; применение полученных знаний для разработки технического советника с помощью программирования на Python и в конструкторе для создания чат бота с использованием искусственного интеллекта.

В результате освоения дисциплины обучающийся должен:

знать: основные принципы построения современного программного обеспечения, типичные формы применения шаблонов проектирования.

уметь: строить модель программного обеспечения применять основные паттерны проектирования, создавать эффективные сетевые и многопоточные приложения.

владеть: навыками применения современных методов проектирования программного обеспечения; современных методов оценки качества программного обеспечения

Место дисциплины в структуре основной профессиональной образовательной программы

Дисциплина «Проектирование и разработка наукоёмкого программного обеспечения» по учебному плану программы магистратуры 15.04.04 «Автоматизация технологических процессов и производств» (набор 2024 г.) относится к Блоку 1 Обязательной части Модуля «Наукоёмкие информационные технологии».

Общая трудоемкость дисциплины составляет 5 зачетных единиц (з.е.), т. е. 180 академических часов контактной и самостоятельной учебной работы студента; работы, связанной с текущей и промежуточной аттестацией по дисциплине.

Основными видами аудиторных учебных занятий по дисциплине являются лекции и лабораторные занятия.

В ходе изучения дисциплины предусматривается применение эффективных методик обучения, которые предполагают постановку вопросов проблемного характера с разрешением их, как непосредственно в ходе занятий, так и в ходе самостоятельной работы.

Контроль знаний в ходе изучения дисциплины осуществляется в виде текущего контроля, РГР, а также промежуточной аттестации в форме Экзамена.

Текущий контроль (контроль выполнения заданий на самостоятельную работу), выполнение РГР предназначен для проверки хода и качества усвоения студентами учебного материала и стимулирования их учебной работы. Он может осуществляться в ходе всех видов занятий в форме, избранной преподавателем или предусмотренной рабочей программой дисциплины.

Текущий контроль предполагает постоянный контроль преподавателем качества усвоения учебного материала, активизацию учебной деятельности студентов на занятиях, побуждение их к самостоятельной систематической работе. Он необходим обучающимся для самоконтроля на разных этапах обучения. Их результаты учитываются выставлением преподавателем оценок в журнале учета успеваемости и в ходе аттестации.

При текущем контроле успеваемости учитывается: выполнение обучающимся всех лабораторных работ, предусмотренных рабочей программой дисциплины, а именно выполнение заданий на лабораторных занятиях; РГР, самостоятельную работу обучающихся; посещаемость аудиторных занятий.

В данном документе представлены методические материалы по изучению дисциплины, включающие тематический план занятий с перечнем вопросов для изучения, рекомендуемой литературой, методическими указаниями.

Помимо данного учебно-методического пособия по изучению дисциплины, студентам следует использовать материалы, размещенные в соответствующем данной дисциплине разделе ЭИОС, в которые более оперативно вносятся изменения для адаптации дисциплины под конкретную группу.

2. ТЕМАТИЧЕСКИЙ ПЛАН

При подготовке к лекционным занятиям студенту необходимо:

1. Повторить ранее изученный материал.

2. Регулярно конспектировать лекционный материал.

3. Доработка конспектов после занятий предполагает повторить изученный материал, дополнить записи на основе прочтения рекомендованной литературы или изучения методических материалов.

4. Изучение дополнительных материалов:

Использование рекомендованной литературы и пособий помогает глубже понять изучаемую тему.

Тематический план изучения дисциплины «Проектирование и разработка наукоёмкого программного обеспечения» представлен в таблице 1.

Таблица 1 –Тематический план

Темы раздела	Раздел (модуль) дисциплины	Тема	Объем аудиторной работы, ч	Объем самостоятельной работы, ч
		Лекции		
1	Введение в технологию разработки наукоёмкого ПО.	Тема 1. Основные понятия Жизненного цикла (ЖЦ) разработки Программного обеспечения (ПО)	6	6
		Тема 2. Модели ЖЦ разработки ПО. ЖЦ тестирования ПО и его роль в ЖЦ разработки	6 12	6 12
2	Объектно-ориентированный анализ и проектирование ПО. Язык UML	Тема 3. Основные элементы языка UML. Сущности, отношения, диаграммы	4	4
		Тема 4. Классы. Диаграммы вариантов использования.	6	6
		Тема 5. Диаграммы последовательности. Диаграммы кооперации.	4	2
		Тема 6. Диаграммы состояний. Диаграммы деятельностей	4	2
		Тема 7. Компоненты, диаграммы развёртывания	2	2
			32	**30**
		Лабораторные занятия		
1.1	Введение в технологию разработки наукоёмкого ПО	Лаб. раб. № 1. Введение в промпт-инжиниринг. Системный промпт и базы данных. Разработка Telegram-бота с регистрацией пользователей в базе данных SQLite. Библиотеки Python	6	4
1.2		Лаб. раб. № 2. Создание ИИ-ассистента в Савви: добавление системного промпта и базы знаний. Интеграция ассистента с Telegram. Интеграция с Google Календарём. Доработка ИИ-ассистента: оповещения в Telegram. Тестирование ИИ-ассистента	6	4

Темы раздела	Раздел (модуль) дисциплины	Тема	Объем аудиторной работы, ч	Объем самостоятельной работы, ч
2.2	Объектно-ориентированный анализ и проектирование ПО. Язык UML	Лаб. раб. № 3. Диаграммы классов	4	7
2.3		Лаб. раб. № 4. Диаграммы вариантов использования	4	7
2.4		Лаб. раб. № 5. Диаграммы последовательности	4	7
2.5		Лаб. раб. № 6. Диаграммы деятельностей	4	7
2.6		Лаб. раб. № 7. Диаграмма кооперации	4	7
			32	**43**
		Рубежный (текущий) и итоговый контроль		
		Тестирование (ЭИОС)	6 (РЭ)	
		РГР, Экзамен	2,25(КА)	34,75
			72,25	**107,75**
		Всего		**180**

3. СОДЕРЖАНИЕ ДИСЦИПЛИНЫ

Курс разбит на разделы, в которых даны указания литературы, рекомендуемой для изучения.

Номера в скобках [ ] означают пособия, из приведенного списка литературы; например, [1, гл. 2, §3, 4,6]обозначает литературу, в котором рекомендуется изучить главу 2, §3, 4, 6. и т. д.

3.1 Раздел 1. Введение в технологию разработки наукоёмкого ПО.

Тема 1. Основные понятия Жизненного цикла (ЖЦ) разработки Программного обеспечения (ПО)

Вопросы для изучения

1. Основные понятия технологии разработки наукоемкого программного обеспечения.
2. Жизненный цикл разработки программного обеспечения.

Методические указания к изучению

Лекции по данной теме построены на основе использования активных форм обучения:

– лекция-беседа (преимущество лекции-беседы состоит в том, что она позволяет привлекать внимание студентов к наиболее важным вопросам темы, определять содержание и темп изложения учебного материала с учетом особенностей студентов),

– проблемная лекция (с помощью проблемной лекции обеспечивается достижение трех основных дидактических целей: усвоение студентами теоретических знаний; развитие теоретического мышления; формирование познавательного интереса к содержанию учебного предмета и профессиональной мотивации будущего специалиста),

– лекция с заранее запланированными ошибками (эта форма проведения лекции необходима для развития у студентов умений оперативно анализировать профессиональные ситуации, выступать в роли экспертов, оппонентов, рецензентов, вычленять неверную или неточную информацию).

На каждой лекции этой темы применяется сочетание этих форм обучения в зависимости от подготовленности студентов и опросов, вынесенных на лекцию.

Присутствие на лекции не должно сводиться лишь к автоматической записи изложения предмета преподавателем. Более того, современный насыщенный материал каждой темы не может (по времени) совпадать с записью в тетради из-за разной скорости процессов – мышления и автоматической записи.

Каждый студент должен разработать для себя систему ускоренного фиксирования на бумаге материала лекции. Поэтому, лектором рекомендуется формализация записи посредством использования общепринятых логико-

математических символов, сокращений, алгебраических (формулы) и геометрических (графики), системных (схемы, таблицы) фиксаций изучаемого материала. Овладение такой методикой, позволяет каждому студенту не только ускорить процесс изучения, но и повысить его качество, поскольку успешное владение указанными приемами требует переработки, осмысления и структуризации материала.

Начальные сведения по лекционному материалу этой темы представлены в теоретической части.

Трудозатраты, связанные с созданием программного обеспечения (ПО) прямо связаны с качеством и сложностью создаваемого ПО. Так трудозатраты на создание программного продукта в три раза больше, чем обычной программы, а трудозатраты на создание простого приложения (редактор текстов), среднего уровня сложности (трансляторы) и высокого уровня сложности (операционные системы) возрастают еще в три раза. Поэтому крайне важны используемые технологии для разработки промышленного программного продукта.

Технология ПО включает в себя такие понятия, как методы, инструменты, организационные мероприятия, направленные на создания промышленного ПО. Под технологией программирования понимают организованную совокупность методов, средств и их программного обеспечения, организационно–административных установлений, направленных на разработку, распространение и сопровождение программной продукции.

Создание ПО включает: Существенные задачи – моделирование сложных концептуальных структур объектов реального мира большой сложности с помощью абстрактных программных объектов; Второстепенные задачи – создание представлений этих абстрактных объектов в программе и описание их поведения. И если 90 % усилий разработчиков (затрат) связано не с существенными задачами, то сведя все затраты к нулю не получим роста в порядки (сравните с электроникой). Если просмотреть историю, то все усилия были направлены на преодоление второстепенных задач.

Пути преодоления существенной сложности: массовый рынок (главное не скорость, а качество и сопровождение); лучший способ повысить производительность труда – купить; быстрое макетирование для установления технических требований к ПО; наращивать программы постепенно, добавляя функциональность (хорошо протестированную); хорошая команда.

Трудности создания ПО:

1. Внутренние, присущие ему (задание технических требований, проектирование и проверка): Сложность – размеры, одинаковых нет, они не просто объединяются, масштабирование это непростое увеличение, это добавление новых элементов и связей, нелинейный рост сложности – сложность присуща (существенна) программным объектам и от нее абстрагироваться нельзя; Согласованность – между людьми, между интерфейсами, которые появились не понятно, как и когда (10 и более лет); Изменяемость – постоянно изменяются требования и применение ПО; Незримость – ПО невидимо, различные графо-

вые модели дают некоторое представление только с определенной точки зрения, но само ПО не плоское и его трудно передать такими моделями.

2. Сопутствующие, внутренне ему не присущие. Движущей силой использования принципов программной инженерии было опасение крупных аварий, к которым могла привести (и привела) разработка все более сложных систем неуправляемыми художниками (не производительность в разы, хотя она и возросла от 3 до 5 раз). Данные МО США по цене исправления одной ошибки следующие: обнаруженные и исправленные на стадии требований – 139 $, на стадии кодирования – 1000 $, на стадии тестирования – 7000 $, на стадии внедрения – 14000 $.

Методологии, технологии и инструментальные средства составляют основу проекта любой информационной системы (ИС). Методология реализуется через конкретные технологии и поддерживающие их стандарты, методики и инструментальные средства, которые обеспечивают выполнение процессов ЖЦ. Под технологией программирования понимают организованную совокупность методов, средств и их программного обеспечения, организационно административных установлений, направленных на разработку, распространение и сопровождение программной продукции.

Одним из базовых понятий методологии проектирования информационных систем (ИС) является понятие жизненного цикла ее программного обеспечения (ЖЦ ПО).

ЖЦ ПО – это непрерывный процесс, который начинается с момента принятия решения о необходимости его создания и заканчивается в момент его полного изъятия из эксплуатации. Основным нормативным документом, регламентирующим ЖЦ ПО, является международный стандарт ИСО/МЭК 12207 – 95 «Информационная технология. Процессы жизненного цикла программных средств» или ISO/IEC 12207 (ISO – InternationalOrganizationofStandardization – Международная организация по стандартизации, IEC – InternationalElectrotechnicalCommission – Международная комиссия по электротехнике). Он определяет структуру ЖЦ, содержащую процессы, действия и задачи, которые должны быть выполнены во время создания ПО. Структура ЖЦ ПО по стандарту ИСО/МЭК 12207 – 95 базируется на трех группах процессов:

– основные процессы ЖЦ ПО (приобретение, поставка, разработка, эксплуатация, сопровождение);

– вспомогательные процессы, обеспечивающие выполнение основных процессов (документирование, управление конфигурацией, обеспечение качества, верификация, аттестация, оценка, аудит, решение проблем);

– организационные процессы (управление проектами, создание инфраструктуры проекта, определение, оценка и улучшение самого ЖЦ, обучение).

Разработка включает в себя все работы по созданию ПО и его компонент в соответствии с заданными требованиями, включая оформление проектной и эксплуатационной документации, подготовку материалов, необходимых для проверки работоспособности и соответствующего качества программных про-

дуктов, материалов, необходимых для организации обучения персонала и т.д. Разработка ПО включает в себя, как правило, анализ, проектирование и реализацию (программирование). Эксплуатация включает в себя работы по внедрению компонентов ПО в эксплуатацию, в том числе конфигурирование базы данных и рабочих мест пользователей, обеспечение эксплуатационной документацией, проведение обучения персонала и т. д., и непосредственно эксплуатацию, в том числе локализацию проблем и устранение причин их возникновения, модификацию ПО.

Управление проектом связано с вопросами планирования и организации работ, создания коллективов разработчиков и контроля за сроками и качеством выполняемых работ. Техническое и организационное обеспечение проекта включает выбор методов и инструментальных средств для реализации проекта, определение методов описания промежуточных состояний разработки, разработку методов и средств испытаний ПО, обучение персонала и т.п. Обеспечение качества проекта связано с проблемами верификации, проверки и тестирования ПО.

Верификация – это процесс определения того, отвечает ли текущее состояние разработки, достигнутое на данном этапе, требованиям этого этапа. Проверка позволяет оценить соответствие параметров разработки с исходными требованиями. Проверка частично совпадает с тестированием, которое связано с идентификацией различий между действительными и ожидаемыми результатами и оценкой соответствия характеристик ПО исходным требованиям. В процессе реализации проекта важное место занимают вопросы идентификации, описания и контроля конфигурации отдельных компонентов и всей системы в целом. Управление конфигурацией является одним из вспомогательных процессов, поддерживающих основные процессы жизненного цикла ПО, прежде всего процессы разработки и сопровождения ПО.

При создании проектов сложных ИС, состоящих из многих компонентов, каждый из которых может иметь разновидности или версии, возникает проблема учета их связей и функций, создания унифицированной структуры и обеспечения развития всей системы.

Управление конфигурацией позволяет организовать, систематически учитывать и контролировать внесение изменений в ПО на всех стадиях ЖЦ. Общие принципы и рекомендации конфигурационного учета, планирования и управления конфигурациями ПО отражены в стандарте ISO 12207–2.

Каждый процесс характеризуется определенными задачами и методами их решения, исходными данными, полученными на предыдущем этапе, и результатами. Результатами анализа, в частности, являются функциональные модели, информационные модели и соответствующие им диаграммы.

ЖЦ ПО носит итерационный характер: результаты очередного этапа часто вызывают изменения в проектных решениях, выработанных на более ранних этапах.

Рекомендуемые источники: [1, гл. 1, 2].

Вопросы для самоконтроля

1. Что такое жизненный цикл разработки ПО? Опишите основные этапы жизненного цикла разработки программного обеспечения.

2. Какова роль анализа требований в жизненном цикле разработки ПО? Почему этот этап важен и какие ошибки часто допускаются на этапе анализа?

3. Какой этап считается основным в процессе проектирования ПО? Каким образом выбираются архитектурные решения?

4. Расскажите о процессе внедрения ПО в эксплуатацию. Что включает в себя этап развертывания системы?

5. Зачем нужен этап сопровождения и поддержки ПО после завершения разработки? Какие типы изменений могут потребоваться в уже готовой системе 6. Чем отличается прототипирование от пилотной версии продукта? Когда каждый из подходов используется?

7. На каком этапе возникает наибольший риск ошибок в проекте? Какие меры принимаются для минимизации рисков?

8. Описать процесс документирования программного обеспечения. Зачем нужна документация и какие типы документов обычно создаются?

9. Что такое рефакторинг и зачем он применяется? Расскажите о принципах рефакторинга.

10. Обсудите концепцию DevOps в контексте жизненного цикла разработки ПО. Чем DevOps отличается от традиционных подходов к разработке и эксплуатации ПО?

3.2 Раздел 1. Введение в технологию разработки наукоёмкого ПО.

Тема 2. Модели ЖЦ разработки ПО. ЖЦ тестирования ПО и его роль в ЖЦ разработки

Вопросы для изучения

1. Каскадная (водопадная) или последовательная модель; Итеративная и инкрементальная – эволюционная (гибридная, смешанная) модель; Спиральная (spiral) модель или модель Боэма

2. Основы методологии RationalUnifiedProcess; Обобщённая модель ЖЦ ПО; Agile-практики: eXtremeProgramming (XP), жизненный цикл тестирования ПО, стандарты качества программного обеспечения.

Методические указания к изучению

Лекции по данной теме построены на основе использования активных форм обучения:

– лекция-беседа (преимущество лекции-беседы состоит в том, что она позволяет привлекать внимание студентов к наиболее важным вопросам темы, определять содержание и темп изложения учебного материала с учетом особенностей студентов);

– проблемная лекция (с помощью проблемной лекции обеспечивается достижение трех основных дидактических целей: усвоение студентами теоретических знаний; развитие теоретического мышления; формирование познавательного интереса к содержанию учебного предмета и профессиональной мотивации будущего специалиста),

– лекция с заранее запланированными ошибками (Эта форма проведения лекции необходима для развития у студентов умений оперативно анализировать профессиональные ситуации, выступать в роли экспертов, оппонентов, рецензентов, вычленять неверную или неточную информацию).

На каждой лекции этой темы применяется сочетание этих форм обучения в зависимости от подготовленности студентов и опросов, вынесенных на лекцию.

Присутствие на лекции не должно сводиться лишь к автоматической записи изложения предмета преподавателем. Более того, современный насыщенный материал каждой темы не может (по времени) совпадать с записью в тетради из-за разной скорости процессов – мышления и автоматической записи.

Каждый студент должен разработать для себя систему ускоренного фиксирования на бумаге материала лекции. Поэтому, лектором рекомендуется формализация записи посредством использования общепринятых логико-математических символов, сокращений, алгебраических (формулы) и геометрических (графики), системных (схемы, таблицы) фиксаций изучаемого материала. Овладение такой методикой, позволяет каждому студенту не только ускорить процесс изучения, но и повысить его качество, поскольку успешное владение указанными приемами требует переработки, осмысления и структуризации материала.

Начальные сведения по лекционному материалу этой темы представлены в теоретической части, см. ниже.

Под моделью ЖЦ понимается структура, определяющая последовательность выполнения и взаимосвязи процессов, действий и задач, выполняемых на протяжении ЖЦ. Модель ЖЦ зависит от специфики ИС и специфики условий, в которых последняя создается и функционирует. Стандарт ISO/IEC 12207 не предлагает конкретную модель ЖЦ и методы разработки ПО. Его регламенты являются общими для любых моделей ЖЦ, методологий и технологий разработки. Стандарт описывает структуру процессов ЖЦ ПО, но не конкретизирует в деталях, как реализовать или выполнить действия и задачи, включенные в эти процессы.

К настоящему времени наибольшее распространение получили следующие основные модели ЖЦ:

– Каскадная (водопадная) или последовательная модель (70–85 гг.);

– Итеративная и инкрементальная – эволюционная (гибридная, смешанная) модель (86–90 гг.);

– Спиральная (spiral) модель или модель Боэма (88–90-е гг.).

Легко обнаружить, что в разное время и в разных источниках приводится

разный список моделей и их интерпретация. Например, ранее, инкрементальная модель понималась как построение системы в виде последовательности сборок (релизов), определенной в соответствии с заранее подготовленным планом и заданными (уже сформулированными) и неизменными требованиями. Сегодня об инкрементальном подходе чаще всего говорят в контексте постепенного наращивания функциональности создаваемого продукта.

Может показаться, что индустрия пришла, наконец, к общей «правильной» модели. Однако, каскадная модель, многократно «убитая» и теорией, и практикой, продолжает встречаться в реальной жизни. Спиральная модель является ярким представителем эволюционного взгляда, но, в то же время, представляет собой единственную модель, которая уделяет явное внимание анализу и предупреждению рисков. Коротко рассмотрим каждую из моделей жизненного цикла.

Каскадная модель

В изначально существовавших однородных ИС каждое приложение представляло собой единое целое. Для разработки такого типа приложений применялся каскадный способ. Его основной характеристикой является разбиение всей разработки на этапы, причем переход с одного этапа на следующий происходит только после того, как будет полностью завершена работа на текущем (рисунок 1).

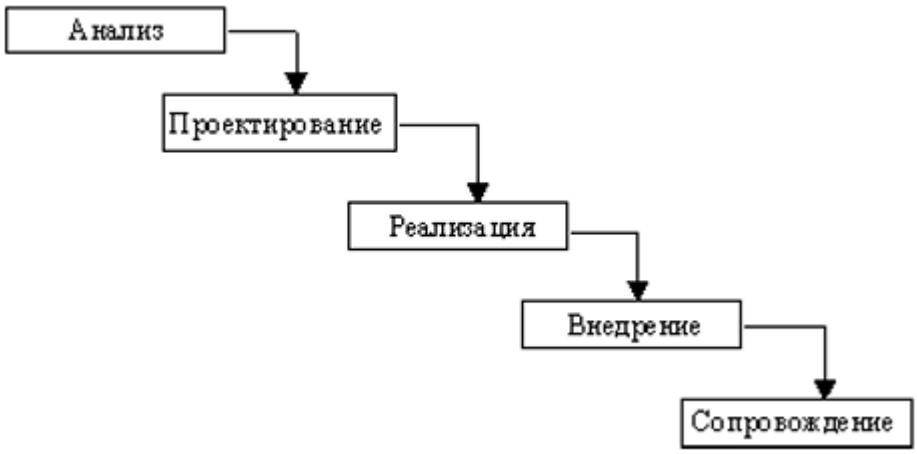

Рисунок 1 – Каскадная схема разработки ПО

Каждый этап завершается выпуском полного комплекта документации, достаточной для того, чтобы разработка могла быть продолжена командой специалистов на следующем этапе.

Положительные стороны применения каскадного подхода заключаются в следующем:

– на каждом этапе формируется законченный набор проектной документации, отвечающий критериям полноты и согласованности;

– выполняемые в логичной последовательности этапы работ позволяют планировать сроки завершения всех работ и соответствующие затраты.

Каскадный подход хорошо зарекомендовал себя при построении ИС, для которых в самом начале разработки можно достаточно точно и полно сформулировать все требования, с тем чтобы предоставить разработчикам свободу реализовать их как можно лучше с технической точки зрения.

В эту категорию попадают сложные расчетные системы, системы реального времени и другие подобные задачи. Однако, в процессе использования этого подхода обнаружился ряд его недостатков, вызванных прежде всего тем, что реальный процесс создания ПО никогда полностью не укладывался в такую жесткую схему. В процессе создания ПО постоянно возникала потребность в возврате к предыдущим этапам и уточнении или пересмотре ранее принятых решений.

Итеративная, инкрементальная модель – эволюционный подход

Для преодоления перечисленных проблем была предложена итеративная модель. Она предполагает разбиение жизненного цикла проекта на последовательность итераций, каждая из которых напоминает «мини-проект», включая все фазы жизненного цикла в применении к созданию меньших фрагментов функциональности, по сравнению с проектом, в целом.

Цель каждой итерации – получение работающей версии программной системы, включающей функциональность, определенную интегрированным содержанием всех предыдущих и текущей итерации. Результата финальной итерации содержит всю требуемую функциональность продукта.

Таким образом, с завершением каждой итерации, продукт развивается инкрементально. С точки зрения структуры жизненного цикла такую модель называют итеративной (iterative). С точки зрения развития продукта – инкрементальной (incremental). Опыт индустрии показывает, что невозможно рассматривать каждый из этих взглядов изолировано. Чаще всего такую смешанную эволюционную модель называют просто итеративной (говоря о процессе) и/или инкрементальной (говоря о наращивании функциональности продукта).

Эволюционная модель подразумевает не только сборку работающей (с точки зрения результатов тестирования) версии системы, но и её развертывание в реальных операционных условиях с анализом откликов пользователей для определения содержания и планирования следующей итерации. «Чистая» инкрементальная модель не предполагает развертывания промежуточных сборок (релизов) системы и все итерации проводятся по заранее определенному плану наращивания функциональности, а пользователи (заказчик) получает только результат финальной итерации как полную версию системы.

С другой стороны, итеративную разработку называют по-разному: инкрементальной, спиральной, эволюционной и постепенной. Разные идеологи вкладывают в эти термины разный смысл, но эти различия не имеют широкого признания и не так важны, как противостояние итеративного метода и метода

водопада. Поскольку, в идеале, на каждом шаге мы имеем работающую систему, итеративная модель имеет следующие преимущества:

– можно очень рано начать тестирование пользователями;

– можно принять стратегию разработки в соответствии с бюджетом, полностью защищающую от перерасхода времени или средств (в частности, за счет сокращения второстепенной функциональности).

Таким образом, значимость эволюционного подхода на основе организации итераций особо проявляется в снижении неопределенности с завершением каждой итерации. В свою очередь, снижение неопределенности позволяет уменьшить риски.

Внедрение любой новой методологии или нового подхода разработки ПО существенно упрощается, если есть поддерживающий ее набор инструментов, позволяющий, как избежать тяжелого рутинного ручного труда, так и обеспечить выполнение процессов и задач, выполнить которые без средств автоматизации невозможно или несоизмеримо с затратами.

Применение итерационной модели очень трудно реализовать без применения инструментов автоматизации: CASE-инструментов планирования и моделирования, средств автоматизации управления основными и вспомогательными процессами разработки, средств автоматизации выполнения процессов разработки, и т. д. Наиболее известным и распространенным вариантом эволюционной модели является спиральная модель.

Спиральная модель ЖЦ разработки ПО

Спиральная модель, представленная на рисунке 2, была впервые сформулирована Барри Боэмом (BarryBoehm) в 1988 г. Отличительной особенностью этой модели является специальное внимание рискам, влияющим на организацию жизненного цикла. Большая часть этих рисков связана с организационными и процессными аспектами взаимодействия специалистов в проектной команде.

Также спиральная модель ЖЦ делает упор на начальные этапы ЖЦ: анализ и проектирование. На этих этапах реализуемость технических решений проверяется путем создания прототипов. Каждый виток спирали соответствует созданию фрагмента или версии ПО, на нем уточняются цели и характеристики проекта, определяется его качество и планируются работы следующего витка спирали. Таким образом, углубляются и последовательно конкретизируются детали проекта, и в результате выбирается обоснованный вариант, который доводится до реализации.

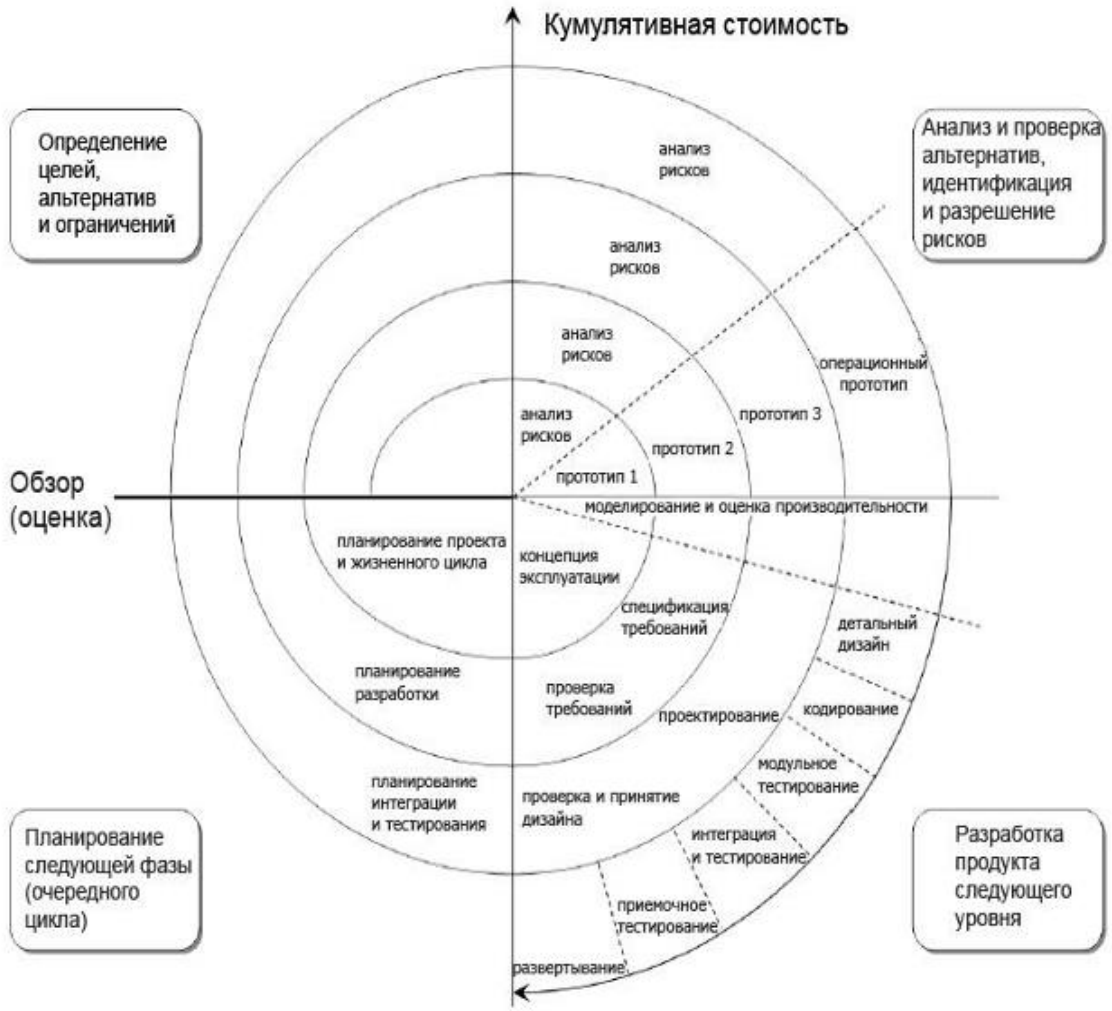

Рисунок 2 – Спиральная модель ЖЦ разработки ПО

Разработка итерациями отражает объективно существующий спиральный цикл создания системы. Неполное завершение работ на каждом этапе позволяет переходить на следующий этап, не дожидаясь полного завершения работы на текущем. При итеративном способе разработки недостающую работу можно будет выполнить на следующей итерации.

Главная же задача – как можно быстрее показать Заказчику работоспособный продукт, тем самым активизируя процесс уточнения и дополнения требований. Перечислим основные преимущества спиральной модели:

– модель уделяет специальное внимание раннему анализу возможностей повторного использования;

– модель предполагает возможность эволюции жизненного цикла, развитие и

изменение программного продукта;

– модель предоставляет механизмы достижения необходимых параметров качества как составную часть процесса разработки программного продукта;

– модель уделяет специальное внимание предотвращению ошибок и отбрасыванию ненужных, необоснованных или неудовлетворительных альтернатив на ранних этапах проекта;

– модель позволяет контролировать источники проектных работ и соответствующих затрат;

– модель не проводит различий между разработкой нового продукта и расширением (или сопровождением) существующего;

– модель позволяет решать интегрированный задачи системной разработки, охватывающей и программную, и аппаратную составляющие создаваемого продукта.

Основная проблема спирального цикла – определение момента перехода на следующий этап. Для ее решения необходимо ввести временные ограничения на каждый из этапов жизненного цикла. Переход осуществляется в соответствии с планом, даже если не вся запланированная работа закончена. План составляется на основе статистических данных, полученных в предыдущих проектах, и личного опыта разработчиков.

Однако, организация ролей (ответственности членов проектной команды), детализация этапов жизненного цикла и процессов, определение активов (артефактов), значимых на разных этапах проекта, практики анализа и предупреждения рисков – все это вопросы уже конкретного процессного фреймворка или, как принято говорить, методологии разработки.

Ниже рассмотрены наиболее успешные на сегодняшний день методологии разработки качественного программного обеспечения и более подробно остановимся на методологии RationalUnifiedProcess (RUP).

Основы методологии Rational Unified Process

На основе существующих стандартов и моделей обеспечения качества создаются своды правил, описывающих правильную организацию и управление процессами создания современных программных продуктов. В них сформирован и документирован набор проверенных на практике принципов, методов и процессов качественной и производительной работы над проектами по созданию программного обеспечения. Так как взглядов на детализацию описания жизненного цикла разработки ПО может быть много – безусловно, методологии тоже различаются.

Наиболее распространенные на сегодняшний день методологии разработки ПО приведены ниже:

– Rational Unified Process (RUP);

– Enterprise Unified Process (EUP);

– Microsoft Solutions Framework (MSF) вдвухпредставлениях:

MSFforAgileиMSFforCMMI (анонсированная изначально как«MSF Formal»);

– Agile-практики: eXtreme Programming (XP), Feature Driven Development (FDD), Dynamic Systems Development Method (DSDM), SCRUM, идругие.

Одной из наиболее успешных в данной области рынка на сегодняшний день является корпорация IBM RationalSoftware и ее идеологи, столпы объектно ориентированного подхода GradyBooch, IvarJacobson и JamesRumbaugh. IBM RationalSoftware выпустила RationalUnifiedProcess (сокращено RUP, в переводе – Универсальный процесс разработки программных систем фирмы IBM Rational) – базу знаний, представляющую собой путеводитель для всех участников проекта по разработке большой программной системы.

RUP – методологическая основа для всего, что выпускает Rational. То есть данный продукт является энциклопедией (методологическим руководством) того, как нужно строить эффективное информационное производство. Также RUP регламентирует этапы разработки ПО, документы, сопровождающие каждый этап, и продукты самой Rational для каждого этапа. В RUP заложены все самые современные идеи. Продукт постоянно обновляется, включая в себя все новые и новые возможности. К достоинствам данной методологии стоит отнести чрезвычайную гибкость, то есть RUP не диктует, что необходимо сделать, а только рекомендует использовать то или иное средство. Как уже говорилось выше, внедрение любой методологии существенно упрощается с применением автоматизации процессов данной методологии. Что возможно только в случае, когда присутствуют: набор технологий, поддерживающих данную методологию, и набор инструментов, реализующих данные технологии. Методология RUP в этом смысле является одной из наиболее «благополучных», поскольку ее поддерживает набор инструментов IBM Rational.

Методология RUP представляется в виде понятного и легко доступного каждому участнику ИТ проекта Web-сайта, содержимое которого может быть настроено под требования команды разработчиков любого размера (средствами RUP Process Workbench и RUP Builder, входящими в состав RUP) и индивидуально под каждого члена проектной команды (MyRUP). Как уже упоминалось выше, на пути выпуска ПО существует ряд проблем.

Вот что предлагает RUP для решения подобных проблем:

– Выпускать программное обеспечение, пользуясь принципом промышленного подхода. То есть так, как поступают любые заводы и фабрики: определяя стадии, потоки, уточняя обязанности каждого участника проекта. Именно промышленный подход позволит достаточно оперативно выпускать новые версии ПО, которые при этом будут надежными и качественными.

– Расширять кругозор специалистов для снятия барьеров. Ведь в подавляющем большинстве случаев специалисты из разных отделов просто говорят на разных языках. Соответственно, снятие языкового барьера должно вести к ускорению работы над программным обеспечением.

– Использовать итеративную разработку вместо каскадной, существующей в настоящее время. Принцип итерации заключается в повторяемости определенной последовательности процессов с целью доведения элемента до безошибочного состояния.

– Обязательное управление требованиями. Всем известно, что по ходу разработки в систему вносятся изменения (самой группой разработчиков или заказчиком – неважно). Rational предлагает мощную систему контроля управления требованиями: их обнаружение и документирование, поддержку соглашений между разработчиками и заказчиками.

– Полный контроль всего происходящего в проекте посредством создания специальных архивов.

– Унифицированный документооборот, приведенный в соответствие со всеми известными стандартами. Это значит, что каждый этап в разработке (начало, работа и завершение) сопровождаются унифицированными документами, которыми должен пользоваться каждый участник проекта.

– Использование визуального моделирования.

– Применение не только механизмов Объектно-ориентированного программирования, но и ОО-мышления и подхода. Методология RUP основана на следующих основных принципах современной программной инженерии:

– Итеративная разработка;
– Управление требованиями;
– Компонентная архитектура;
– Визуальное моделирование;
– Управление изменениями;
– Постоянный контроль качества.

ЖЦ ПО по методологии RationalUnifiedProcessRationalUnifiedProcess во всех тонкостях описывает процесс разработки программного обеспечения. В соответствие с RUP, жизненный цикл программного продукта состоит из серии относительно коротких итераций (рисунок 3).

Рисунок 3 – Итерационный ЖЦ программного продукта

Итерация – это законченный цикл разработки, приводящий к выпуску конечного продукта или некоторой его сокращенной версии, которая расширяется от итерации к итерации, чтобы, в конце концов, стать законченной системой. Каждая итерация включает, как правило, задачи планирования работ, анализа, проектирования, реализации, тестирования и оценки достигнутых результатов. Однако соотношения этих задач могут существенно меняться.

В соответствие с соотношением различных задач в итерации они группируются в фазы. В первой фазе – Начало – основное внимание уделяется задачам анализа. В итерациях второй фазы – Разработка – основное внимание уделяется проектированию и опробованию ключевых проектных решений. В третьей фазе – Построение – наиболее велика доля задач разработки и тестирования. А в последней фазе – Передача – решаются в наибольшей мере задачи тестирования и передачи системы Заказчику.

Использование принципов продукта RationalUnifiedProcess обеспечивает строгий подход к назначению задач и ответственности в пределах группы разработки. Его цель состоит в том, чтобы гарантировать высокое качество программного продукта, отвечающего потребностям конечных пользователей, в пределах предсказуемого временного графика и бюджета.

Обобщенная модель ЖЦ ПО

На основе рассмотренных выше моделей жизненного цикла разработки ПО можно сделать вывод, что методологии разработки в IT-индустрии стремительно развиваются и оптимизируются, находя новые методы и технологии для повышения качества производимого ПО без увеличения временных и/или материальных затрат. При этом все больше возрастает роль автоматизации процессов разработки: без использования техник и инструментов автоматизации немыслима ни одна разработка ПО по современным методологиям, таким как RUP, но по какой бы методологии и модели ЖЦ ПО не велась разработка – можно выделить общие стадии и этапы, присущие каждой из них и составляющие основу любой разработки. Причем, каждый из этапов имеет свой строго определенный набор задач и функций.

К основным стадиям обобщенной модели ЖЦ ПО можно отнести:

1) анализ требований и проектирование;
2) разработка;
3) эксплуатация;
4) развитие.

Ниже перечислены основные этапы обобщенной модели ЖЦ ПО:

1) анализ требований и разработка технического задания (ТЗ);
2) проектирование системы;
3) детальное проектирование компонент;
4) кодирование и отладка компонент;
5) интеграция и комплексная отладка;

6) тестирование системы: верификация и валидация, приемочные испытания;

7) изготовление поставочного пакета ПО и документации, доставка заказчику;

8) внедрение и поддержка процесса эксплуатации;

9) сопровождение и развитие базовой версии.

Обобщенную модель ЖЦ ПО можно представить на рисунке 4.

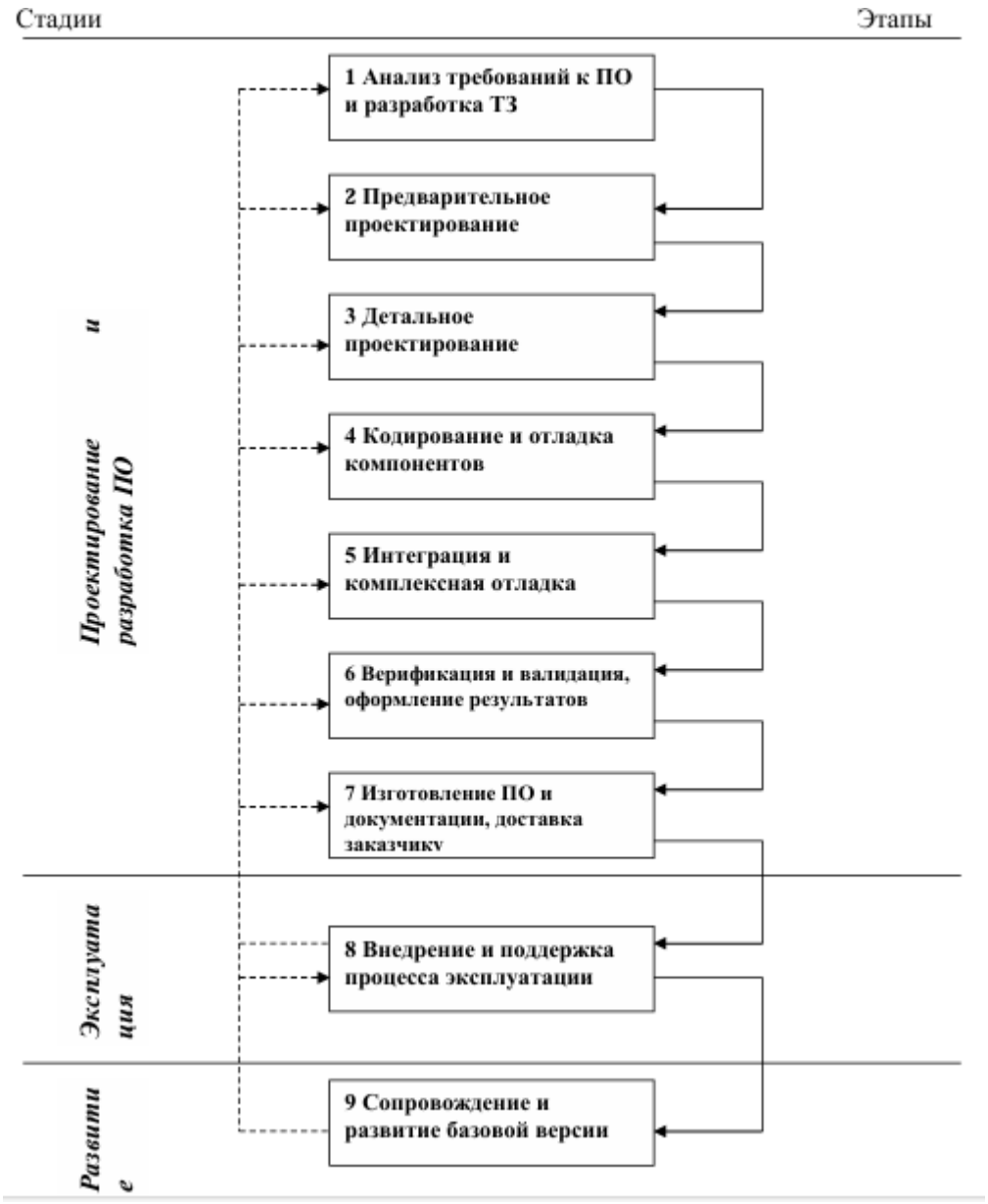

Рисунок 4 – Обобщённая схема ЖЦ проектирования программного ПО

Обобщенная модель ЖЦ ПО напоминает собой модернизированную каскадную модель разработки. Но, в отличии от последней, она не подразумевает, что каждый этап должен быть завершен до начала последующего. Также обоб-

щенная модель не утверждает, что уже пройденные этапы не могут быть пересмотрены позже в процессе разработки.

Модель, представленную на рисунке 4, можно считать отображением того, как каждый из основополагающих процессов разработки взаимодействует с остальными. Как видно из обобщенной модели ЖЦ ПО, тестирование является неотъемлемой дисциплиной любой современной методологии.

Тестирование имеет свои цели, задачи, роли, виды, методы, критерии, свою методологию и технологию. На основе анализа современных методологий и моделей качества, можно сделать вывод, что тестирование имеет свой собственный жизненный цикл – жизненный цикл тестирования программ (ЖЦ ТП). Однако, тестирование – это не изолированный процесс. Он тесно связан с другими процессами, входящими в состав методологии разработки. Поэтому ЖЦ ТП тесно завязан на ЖЦ ПО и накладывается на него. В существующей литературе не производится описания общей методологии и технологии тестирования и не описывается ЖЦ ТП. Чаще всего упор производится на определенный вид тестирования, например, функциональное тестирование.

Одним из учебных вопросов данной темы является объединение и систематизация существующих знаний для описания общей методологии тестирования, а также определению и описанию ЖЦ ТП и выявлению его роли в ЖЦ ПО.

Жизненный цикл тестирования программного обеспечения

Его роль в жизненном цикле разработки при описании основных понятий разработки и тестирования программных продуктов мы будем основываться на принципах организации процессов создания программного обеспечения, приведенных в RUP, так как данные понятия наиболее полно отражают современные стандарты создания качественного ПО (CMM, SPICE, и др.). Коротко рассмотрим современные стандарты качества.

В настоящее время существуют десятки различных подходов к обеспечению качества ПО, и у всех есть свои преимущества. Одной из первых моделей качества стал стандарт ISO (Международной организации по стандартизации) серии 9000, первая версия которого была выпущена в 1987 г. С тех пор сертификаты ISO серии 9000 сохраняют неизменную популярность и признаются во всем мире. Однако время не стоит на месте, и методики, положенные в основу стандартов серии ISO 9000, постепенно устарели. Это заставило экспертов разрабатывать более совершенные решения в области обеспечения качества, что привело к созданию в начале 90-х годов целого ряда новых стандартов и методологий. К ним относятся такие стандарты, как: Bootstrap, Trillium, ориентированный на разработку продуктов в области телекоммуникаций и ISO 12207, посвященный жизненному циклу программного обеспечения. Два же наиболее удачных и содержательных стандарта – CapabilityMaturityModel (CMM) и ISO/IEC 15504 (SPICE). Коротко остановимся на описании их основ.

Но для начала приведем краткую сводку терминов, используемых в дальнейшем изложении.

Возможность процесса разработки ПО (softwareprocesscapability) описывает ожидаемый результат, который можно достигнуть, следуя процессу разработки.

Зрелость процесса разработки ПО (softwareprocessmaturity) – это степень определенности, управляемости, измеряемости и эффективности процесса разработки ПО.

Качество (quality) – степень соответствия системы, компоненты, процесса или службы потребностям и ожиданиям клиента или пользователя.

Количественное управление процессом (quantitativeprocessmanagement) заключается в выявлении причин для особых отклонений процесса от планируемого и подобающего исправления этих причин (таким образом, качество процесса измеряется не в количестве написанных строк кода или найденных ошибок, а в соответствии процесса исходному плану).

Обеспечение качества ПО (softwarequalityassurance) предназначено для информирования руководства об успешности и зрелости процесса разработки ПО и конечных продуктах.

Определение процесса (processdefinition) – это рабочее определение набора мер, необходимых для достижения намеченных целей.

Рекомендуемая литература:[1, гл. 3, 4].

Вопросы для самоподготовки

1. Какие существуют модели жизненного цикла разработки ПО? Назовите и охарактеризуйте наиболее известные модели (например, каскадная модель, спиральная модель, Agile).

2. Опишите каскадную модель разработки ПО. Какие преимущества и недостатки она имеет?

3. Почему гибкая методология разработки (Agile) стала популярной? В каких случаях целесообразно использовать Agile-подход?

4. Объясните, почему тестирование является важным этапом в жизненном цикле разработки ПО. Какие виды тестирования вы знаете?

5. В чем разница между инкрементальной и итерационной моделями разработки ПО? Приведите примеры каждой из моделей.

6. Как управлять изменениями требований в процессе разработки ПО? Какие инструменты и методы используются для управления изменениями?

7. Назовите ключевые метрики качества ПО. Как измеряется качество программного продукта?

8. Поясните разницу между разработкой через тестирование (TDD) и традиционным подходом к разработке ПО. В чем заключается преимущество TDD?

9. Каковы основные проблемы, возникающие при управлении проектом по разработке ПО? Какие подходы применяются для их решения?

10. Различия между фазовым и непрерывным тестированием в рамках жизненного цикла разработки ПО. Плюсы и минусы каждого подхода.

3.3 Раздел 2. Объектно-ориентированный анализ и проектирование ПО. Язык UML

Тема 3.Основные элементы языка UML. Сущности, отношения, диаграммы

Вопросы для изучения

1. Объектно-ориентированная методология. Особенности объектно-ориентированного подхода к проектированию и разработке информационных систем. Абстракция, наследование, полиморфизм, инкапсуляция.

2. Словарь языка UML:

– сущности (предметы);

– отношения;

– диаграммы.

Методические указания

Методология объектно-ориентированного программирования пришла на смену процедурной или алгоритмической организации структуры программного кода, когда стало очевидно, что традиционные методы процедурного программирования не способны справиться ни с растущей сложностью программ и их разработки, ни с повышением их надежности.

Объектно-ориентированное программирование (ООП, Object-OrientedProgramming) – это совокупность принципов, технологий, а также инструментальных средств для создания программных систем на основе архитектуры взаимодействия объектов.

В соответствии с методологией ООП в качестве отдельных структурных единиц программы рассматриваются не процедуры и функции, а классы и объекты с соответствующими свойствами и методами. Как следствие, программа перестала быть последовательностью предопределенных на этапе кодирования действий, а преобразовалась в событийно управляемую. В этом случае каждая программа представляет собой бесконечный цикл ожидания заранее определенных событий. Инициаторами событий могут быть другие программы или пользователи, а при наступлении отдельного события программа выходит из состояния ожидания и реагирует на него вполне адекватным образом.

Основные принципы объектно-ориентированного программирования: абстракция, наследование, инкапсуляция, полиморфизм.

Абстракция – это характеристика сущности, которая отличает ее от других сущностей. Абстракция определяет границу представления соответствующего элемента модели и применяется для определения фундаментальных понятий ООП, таких как класс и объект. Класс представляет собой абстракцию совокупности реальных объектов, которые имеют общий набор свойств и обладают одинаковым поведением. Объект в контексте ООП рассматривается как экземпляр соответствующего класса. Объекты, которые не имеют идентичных свойств или не обладают одинаковым поведением, по определению, не могут

быть отнесены к одному классу. Классы можно организовать в виде иерархической структуры, которая по внешнему виду напоминает схему классификации в понятийной логике. Иерархия понятий строится следующим образом. В качестве наиболее общего понятия или категории берется понятие, имеющее наибольший объем и, соответственно, наименьшее содержание. Это самый высокий уровень абстракции для данной иерархии. Затем данное общее понятие конкретизируется, т. е. уменьшается его объем и увеличивается содержание. Появляется менее общее понятие, которое на схеме иерархии будет расположено на уровень ниже исходного.

Наследование – принцип, в соответствии с которым знание о наиболее общей категории разрешается применять для более частной категории. Наследование тесно связано с иерархией классов, определяющей, какие классы следует считать наиболее абстрактными и общими по отношению к другим классам. При этом если общий или родительский класс (предок) обладает фиксированным набором свойств и поведением, то производный от него класс (потомок) должен содержать этот же набор свойств и подобное поведение, а также дополнительные, которые будут характеризовать уникальность полученного класса.

Инкапсуляция – характеризует сокрытие отдельных деталей внутреннего устройства классов от внешних по отношению к нему объектов или пользователей. Клиенту, взаимодействующему с объектом класса, необязательно знать, каким образом осуществлен тот или иной элемент класса. Конкретная реализация присущих классу свойств и методов, которые определяют его поведение, является собственным делом данного класса. Более того, отдельные свойства и методы класса могут быть невидимы за его пределами, это относится к базовой идее введения различных категорий видимости для элементов класса.

Полиморфизм – свойство объектов принимать различные внешние формы в зависимости от обстоятельств. Применительно к ООП полиморфизм означает, что действия, выполняемые одноименными методами, могут различаться в зависимости от того, к какому из классов относится тот или иной метод.

Наиболее существенным обстоятельством в развитии методологии ООП явилось осознание того, что процесс написания программного кода может быть отделен от процесса проектирования структуры программы. Прежде, чем начать программирование классов, их свойств и методов, необходимо определить сами эти классы. Более того, нужно дать ответы на следующие вопросы: сколько и какие классы нужно определить для решения поставленной задачи, какие свойства и методы необходимы для придания классам требуемого поведения, а также установить взаимосвязи между классами. Эта совокупность задач не столько связана с написанием кода, сколько с общим анализом требований к будущей программе, а также с анализом конкретной предметной области, для которой разрабатывается программа.

Сущности – это абстракции, являющиеся основными элементами модели. Отношения связывают различные сущности.

Диаграмма – это графическое представление множества сущностей. Изображается она, чаще всего, как связный граф из вершин (сущностей) и дуг (отношений).

В UML имеется четыре типа сущностей:

– структурные;

– поведенческие;

– группирующие;

– аннотационные.

Сущности являются основными объектно–ориентированными блоками языка. С их помощью можно создавать корректные модели. Структурные сущности – представляют собой статические части модели, соответствующие концептуальным или физическим элементам системы. Существует семь разновидностей структурных сущностей.

В языке UML определены четыре типа отношений:

– зависимость;

– ассоциация;

– обобщение;

– реализация.

Эти отношения являются основными связующими строительными блоками в UML и применяются для создания моделей.

В рамках языка UML все представления о модели сложной системы фиксируются в виде специальных графических конструкций, получивших название диаграмм.

Диаграмма в UML – это графическое представление набора элементов, изображаемое чаще всего в виде графа с вершинами (сущностями) и ребрами (отношениями). Диаграммы рисуют для визуализации системы с разных точек зрения. Диаграмма – это в некотором смысле одна из проекций системы.

Теоретически диаграммы могут содержать любые комбинации сущностей и отношений. На практике, однако, применяется сравнительно небольшое количество типовых комбинаций.

В UML выделяют девять типов диаграмм: диаграммы классов; диаграммы объектов; диаграммы вариантов использования; диаграммы последовательностей; кооперативные диаграммы; диаграммы состояний; диаграммы деятельностей; диаграммы компонентов; диаграммы размещения.

Инструментальные средства позволяют генерировать и другие диаграммы, но девять перечисленных встречаются на практике чаще всего. В целом интегрированная модель сложной системы в нотации UML может быть представлена в виде совокупности указанных выше диаграмм (рисунок 5).

Рисунок 5 – Интегрированная модель сложной системы в нотации UML

Кроме графических элементов, которые определены для каждой канонической диаграммы, на них может быть изображена текстовая информация, которая расширяет семантику базовых элементов. В последующих темах диаграммы рассматриваются более подробно.

Рекомендуемые источники: [2, гл. 1, 4].

Вопросы для самоконтроля

1. Что такое Полиморфизм?
2. Что такое Инкапсуляция
3. Что такое наследование?
4. Что такое абстракция?
5. Что такое Диаграммы в UML?
6. Какие виды диаграмм используются в Uml?
7. Какие типы отношений существуют в UML?

3.4. Раздел 2. Объектно-ориентированный анализ и проектирование ПО. Язык UML

Тема 4. Классы. Диаграммы вариантов использования

Вопросы для изучения

1. Элементы диаграммы классов. Области видимости и действия, кратность и иерархия классов. Отношения между классами.

2. Базовые элементы диаграммы вариантов использования. Отношения в диаграмме вариантов использования.

Методические указания

Диаграмма классов (classdiagram) – диаграмма языка UML, на которой представлена совокупность декларативных или статических элементов модели, таких как классы с атрибутами и операциями, а также связывающие их отношения.

Класс (class) – абстрактное описание множества однородных объектов, имеющих одинаковые атрибуты, операции и отношения с объектами других классов. Конкретный класс (concreteclass) – класс, на основе которого могут быть непосредственно созданы экземпляры или объекты. Абстрактный класс (abstractclass) – класс, который не имеет экземпляров или объектов.

Атрибут (attribute) – содержательная характеристика класса, описывающая множество значений, которые могут принимать отдельные объекты этого класса. Каждому атрибуту класса соответствует отдельная строка текста, которая состоит из квантора видимости атрибута, имени атрибута, его кратности, типа значений атрибута и, возможно, его исходного значения.

Операция (operation) – это сервис, предоставляемый каждым экземпляром или объектом класса по требованию своих клиентов, в качестве которых могут выступать другие объекты, в том числе и экземпляры данного класса. Каждой операции класса соответствует отдельная строка, которая состоит из квантора видимости операции, имени операции, выражения типа возвращаемого операцией значения и строки-свойства данной операции.

Стереотип – это механизм, позволяющий категоризовать классы. Некоторые из стереотипов используются во время анализа, другие после спецификации используемого языка программирования. Управляющие классы (Controlclass) отвечают за координацию действий других классов. Пограничными классами (boundaryclass) называются классы, расположенные на границе системы со всем остальным миром. Классы-сущности (entity) содержат информацию, хранимую постоянно. Интерфейс (interface) – именованное множество операций, которые характеризуют поведение отдельного элемента модели.

Отношения между классами.

Кроме внутреннего устройства классов важную роль при разработке проектируемой системы имеют различные отношения между классами, которые также могут быть изображены на диаграмме классов. Совокупность допустимых типов таких отношений строго фиксирована в языке UML и определяется самой семантикой этих отношений.

Базовые отношения, изображаемые на диаграммах классов: Отношение ассоциации (Associationrelationship) Отношение обобщения (Generalizationrelationship) Отношение агрегации (Aggregationrelationship) Отношение композиции (Compositionrelationship).

Каждое из этих отношений имеет собственное графическое представление, которое отражает семантический характер взаимосвязи между объектами соответствующих классов.

В самом общем случае, диаграмма вариантов использования представляет

собой граф специального вида, который является графической нотацией для представления конкретных элементов модели и отношений между этими элементами. Базовыми элементами диаграммы вариантов использования являются вариант использования и актер.

Диаграмма вариантов использования

Вариант использования (usecase) – внешняя спецификация последовательности действий, которые система или другая сущность могут выполнять в процессе взаимодействия с актерами.

Содержание варианта использования может быть представлено в форме дополнительного пояснительного текста (сценария), который раскрывает смысл или семантику действий при его выполнении.

Актер (actor) – согласованное множество ролей, которые играют внешние сущности по отношению к вариантам использования при взаимодействии с ними. Актер представляет собой любую внешнюю по отношению к моделируемой системе сущность, которая взаимодействует с системой и использует ее функциональные возможности для достижения определенных целей или решения частных задач. Актеры взаимодействуют с системой посредством передачи и приема сообщений от вариантов использования.

Отношение (relationship) – семантическая связь между отдельными элементами модели. В языке UML имеется несколько стандартных видов отношений:

– ассоциации (associationrelationship) – специфицирует семантические особенности взаимодействия актеров и вариантов использования;

– включения (includerelationship) – указывает на то, что заданное поведение для одного варианта использования включается последовательность поведения другого варианта использования;

– расширения (extendrelationship) – определяет взаимосвязь базового варианта использования с другим вариантом использования, функциональное поведение которого задействуется базовым не всегда, а только при выполнении дополнительных условий;

– обобщения (generalizationrelationship) – указывает на родительский вариант использования.

Базовыми элементами диаграммы вариантов использования являются вариант использования и актер. Вариант использования (Usecase) – внешняя спецификация последовательности действий, которые система или другая сущность могут выполнять в процессе взаимодействия с актерами.

Вариант использования представляет собой спецификацию общих особенностей поведения или функционирования моделируемой системы без рассмотрения внутренней структуры этой системы. Несмотря на то, что каждый вариант использования определяет последовательность действий, которые должны быть выполнены проектируемой системой при взаимодействии ее с соответствующим актером, сами эти действия не изображаются на рассматривае-

мой диаграмме. Содержание варианта использования может быть представлено в форме дополнительного пояснительного текста, который раскрывает смысл или семантику действий при выполнении данного варианта использования. Такой пояснительный текст получил название текста–сценария или просто сценария. Отдельный вариант использования обозначается на диаграмме эллипсом, внутри которого содержится его краткое имя в форме существительного или глагола с пояснительными словами. Сам текст имени варианта использования должен начинаться с заглавной буквы.

Цель спецификации варианта использования заключается в том, чтобы зафиксировать некоторый аспект или фрагмент поведения проектируемой системы без указания особенностей реализации данной функциональности. В этом смысле каждый вариант использования соответствует отдельному сервису, который предоставляет моделируемая система по запросу актера, т. е. определяет один из способов применения системы. Сервис, который инициализируется по запросу актера, должен представлять собой законченную последовательность действий. Это означает, что после того как система закончит обработку запроса актера, она должна возвратиться в исходное состояние, в котором снова готова к выполнению следующих запросов. Диаграмма вариантов использования содержит конечное множество вариантов использования, которые в целом должны определять все возможные стороны ожидаемого поведения системы. Для удобства множество вариантов использования может рассматриваться как отдельный пакет. Применение вариантов использования на всех этапах работы над проектом позволяет не только достичь требуемого уровня унификации обозначений для представления функциональности подсистем и системы в целом, но и является мощным средством последовательного уточнения требований к проектируемой системе на основе их итеративного обсуждения со всеми заинтересованными специалистами.

Рекомендуемые источники: [2, гл. 3, 5].

Вопросы для самоконтроля

1. Что такое диаграмма классов? Каково её назначение?
2. Какие основные элементы входят в состав диаграммы классов?
3. Что обозначают области видимости элементов класса (public, private, protected)?
4. В чём заключается разница между атрибутами и операциями в классе?
5. Опишите, какие существуют типы отношений между классами (ассоциация, агрегация, композиция).
6. Что такое диаграмма вариантов использования? Для каких целей она применяется?
7. Какие ключевые элементы входят в состав диаграммы вариантов использования?
8. Поясните разницу между акторами и вариантами использования.

9. Как правильно изображаются связи между актёрами и вариантами использования?

3.5 Раздел 2.Объектно-ориентированный анализ и проектирование ПО. Язык UML

Тема 5. Диаграммы последовательности. Диаграммы кооперации

Вопросы для изучения

1. Графические элементы диаграммы последовательности.
2. Графические элементы диаграммы кооперации. Кооперация объектов.

Методические указания

Диаграммы последовательности

Диаграмма последовательности (Sequencediagram) – диаграмма, на которой показаны взаимодействия объектов, упорядоченные по времени их проявления. Диаграммы последовательности характеризуются двумя особенностями. Во–первых, на них показана линия жизни объекта.

Линия жизни объекта (Objectlifeline) – вертикальная линия на диаграмме последовательности, которая представляет существование объекта в течение определенного периода времени. Линия жизни объекта изображается пунктирной вертикальной линией, ассоциированной с единственным объектом на диаграмме,в течение которого объект существует в системе и, следовательно, может потенциально участвовать во всех ее взаимодействиях.

Если объект существует в системе постоянно, то и его линия жизни должна продолжаться по всей рабочей области диаграммы последовательности от самой верхней ее части до самой нижней. Большая часть объектов, представленных на диаграмме взаимодействий, существует на протяжении всего взаимодействия, поэтому их изображают в верхней части диаграммы, а их линии жизни прорисованы сверху донизу.

Объекты могут создаваться и во время взаимодействий. Линии жизни таких объектов начинаются с получения сообщения со стереотипом create. Объекты могут также уничтожаться во время взаимодействий; в таком случае их линии жизни заканчиваются получением сообщения со стереотипом destroy, а в качестве визуального образа используется большая буква X, обозначающая конец жизни объекта. (Обстоятельства жизненного цикла объекта можно указывать с помощью ограничений new, destroyed и transient.) Если объект на протяжении своей жизни изменяет значения атрибутов, состояние или роль, это можно показать, поместив копию его пиктограммы на линии жизни в точке изменения. Вторая особенность этих диаграмм – фокус управления.

Он изображается в виде вытянутого прямоугольника, показывающего промежуток времени, в течение которого объект выполняет какое–либо действие, непосредственно или с помощью подчиненной процедуры. Верхняя грань прямоугольника выравнивается по временной оси с моментом начала

действия, нижняя – с моментом его завершения (и может быть помечена сообщением о возврате). Вложенность фокуса управления, вызванную рекурсией (то есть обращением к собственной операции) или обратным вызовом со стороны другого объекта, можно показать, расположив другой фокус управления чуть правее своего родителя (допускается вложенность произвольной глубины).

Если место расположения фокуса управления требуется указать с максимальной точностью, можно заштриховать область прямоугольника, соответствующую времени, в течение которого метод действительно работает и не передает управление другому объекту. Крайним слева на диаграмме последовательности изображается объект – инициатор моделируемого процесса взаимодействия (объект a на рисунке 6). Правее – другой объект, который непосредственно взаимодействует с первым. Таким образом, порядок расположения объектов на диаграмме последовательности определяется исключительно соображениями удобства визуализации их взаимодействия друг с другом.

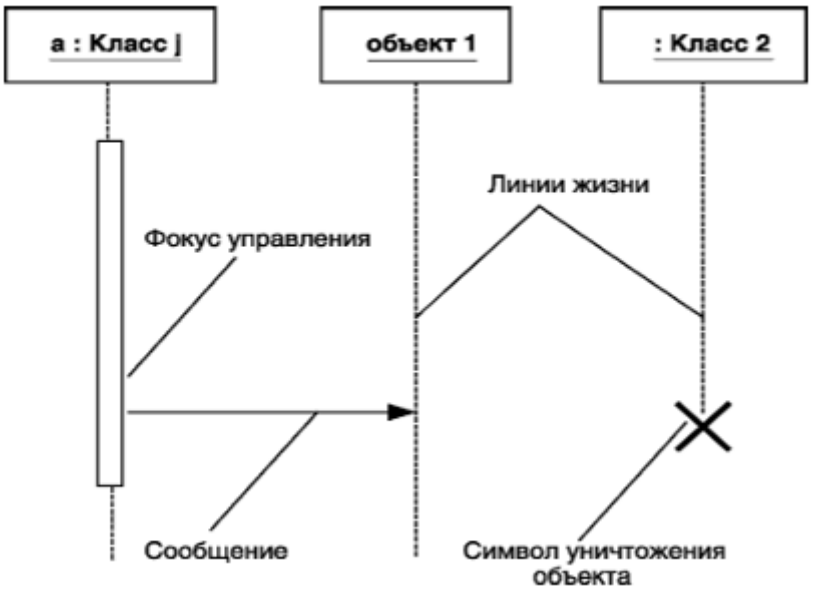

Рисунок 6 – Графические элементы диаграммы последовательности

Диаграмма кооперации

Диаграмма кооперации (Collaborationdiagram) предназначена для описания поведения системы на уровне отдельных объектов, которые обмениваются между собой сообщениями, чтобы достичь нужной цели или реализовать некоторый вариант использования.

С точки зрения аналитика или архитектора системы в проекте важно представить структурные связи отдельных объектов между собой. Такое представление структуры модели как совокупности взаимодействующих объектов и обеспечивает диаграмма кооперации. Диаграммой последовательностей (Sequencediagram) называется диаграмма взаимодействий, акцентирующая внимание на временной упорядоченности сообщений. А кооперативной диа-

граммой (Collaborationdiagram) называется диаграмма взаимодействий, основное внимание в которой уделяется структурной организации объектов, принимающих и отправляющих сообщения. Графически такая диаграмма представляет собой граф из вершин и ребер.

Кооперация (Collaboration) – спецификация множества объектов отдельных классов, совместно взаимодействующих с целью реализации отдельных вариантов использования в общем контексте моделируемой системы. Понятие кооперации – одно из фундаментальных в языке UML. Цель самой кооперации состоит в том, чтобы специфицировать особенности реализации отдельных вариантов использования или наиболее значимых операций в системе. Кооперация определяет структуру поведения системы в терминах взаимодействия участников этой кооперации.

На диаграмме кооперации размещаются объекты, представляющие собой экземпляры классов, связи между ними, которые в свою очередь являются экземплярами ассоциаций и сообщения. Связи дополняются стрелками сообщений, при этом показываются только те объекты, которые участвуют в реализации моделируемой кооперации. Далее, как и на диаграмме классов, показываются структурные отношения между объектами в виде различных соединительных линий. Связи могут дополняться именами ролей, которые играют объекты в данной взаимосвязи. И, наконец, изображаются динамические взаимосвязи – потоки сообщений в форме стрелок с указанием направления рядом с соединительными линиями между объектами, при этом задаются имена сообщений и их порядковые номера в общей последовательности сообщений.

Одна и та же совокупность объектов может участвовать в реализации различных кооперациий. В зависимости от рассматриваемой кооперации, могут изменяться как связи между отдельными объектами, так и поток сообщений между ними. Именно это отличает диаграмму кооперации от диаграммы классов, на которой должны быть указаны все без исключения классы, их атрибуты и операции, а также все ассоциации и другие структурные отношения между элементами модели.

На кооперативных диаграммах линию жизни объекта явным образом не показывают, хотя можно показать сообщения create и destroy. Не показывают там и фокус управления, однако порядковые номера сообщения могут отображать вложенность. В языке UML предусмотрены стандартные действия, выполняемые в ответ на получение соответствующего сообщения. Они могут быть явно указаны на диаграмме кооперации в форме стереотипа перед именем сообщения, к которому они относятся, или выше его. В этом случае они записываются в угловых кавычках. Кооперативная диаграмма акцентирует внимание на организации объектов, принимающих участие во взаимодействии. Для создания кооперативной диаграммы нужно расположить участвующие во взаимодействии объекты в виде вершин графа. Затем связи, соединяющие эти объекты, изображаются в виде дуг этого графа. Наконец, связи дополняются сообщениями, которые объекты принимают и посылают. Это дает пользователю яс-

ное визуальное представление о потоке управления в контексте структурной организации кооперирующихся объектов.

Рекомендуемые источники по теме: [2, с. 66–71]; [3, с. 7–35]; [4, с. 5–20].

Вопросы для самоконтроля

1. Что такое диаграмма последовательности?
2. Какие элементы входят в состав диаграммы последовательности?
3. Каковы различия между диаграммой последовательности и диаграммой классов?
4. Какие типы сообщений используются на диаграммах последовательности?
5. Что означает стрелка сообщения на диаграмме последовательности?
6. Для чего используется диаграмма кооперации?
7. Какие элементы присутствуют на диаграмме кооперации?
8. Чем отличаются диаграммы последовательности и кооперации?
9. Когда лучше использовать диаграмму последовательности, а когда – диаграмму кооперации?

3.6 Раздел 2.Объектно-ориентированный анализ и проектирование ПО. Язык UML

Тема 6. Диаграммы состояний. Диаграммы деятельностей

Вопросы для изучения

1. Состояния и псевдосостояния. Графическое отображение элементов на диаграмме состояний. Рекомендации по построению диаграмм состояний.
2. Рекомендации по построению диаграммы деятельностей. Примеры.

Методические указания

Конечный автомат (statemachine) – модель для спецификации поведения объекта в форме последовательности его состояний, которые описывают реакцию объекта на внешние события, выполнение объектом действий, а также изменение его отдельных свойств.

Диаграмма состояний (statechartdiagram) – диаграмма, которая представляет конечный автомат. Главное назначение – описать возможные последовательности состояний и переходов, которые в совокупности характеризуют поведение моделируемой системы в течение всего ее жизненного цикла.

Состояние (state) – условие или ситуация в ходе жизненного цикла объекта, в течение которого он удовлетворяет логическому условию, выполняет определенную деятельность или ожидает события. Псевдосостояние (pseudo-state) – вершина в конечном автомате, которая имеет форму состояния, но не обладает поведением.

Действие (action) – спецификация выполнимого утверждения, которая образует абстракцию вычислительной процедуры. Входное действие

(entryaction) – действие, которое выполняется в момент перехода в данное состояние. Действие выхода (exitaction) – действие, производимое при выходе из данного состояния. Внутренняя деятельность (doactivity) – выполнение объектом операций или процедур, которые требуют определенного времени.

Переход (transition) – отношение между двумя состояниями, которое указывает на то, что объект в первом состоянии должен выполнить определенные действия и перейти во второе состояние. Если параллельный переход имеет две или более исходящих из него дуг, то его называют разделением (fork). Если же он имеет две или более входящие дуги, то его называют слиянием (join).

Событие (event) – спецификация существенных явлений в поведении системы, которые имеют местоположение во времени и пространстве.

Переход называется триггерным, если его специфицирует событие-триггер, связанное с внешними условиями по отношению к рассматриваемому состоянию. Переход называется нетриггерным, если он происходит по завершении выполнения деятельности в данном состоянии.

Сторожевое условие (guardcondition) – логическое условие, записанное в прямых скобках и представляющее собой булевское выражение.

Основным направлением использования диаграмм деятельности является визуализация особенностей реализации операций классов, когда необходимо представить алгоритмы их выполнения.

Состояние деятельности (activitystate) – состояние в графе деятельности, которое служит для представления процедурной последовательности действий, требующих определенного времени.

Состояние действия (actionstate) – специальный случай состояния. Состояние действия не может иметь внутренних переходов, поскольку оно является элементарным. Состояние под-деятельности (subactivitystate) – состояние в графе деятельности, которое служит для представления неатомарной последовательности шагов процесса.

Графически ветвление на диаграмме деятельности обозначается символом решения (decision), изображаемого в форме небольшого ромба, внутри которого нет никакого текста.

Дорожка (swimlane) – графическая область диаграммы деятельности, содержащая элементы модели, ответственность за выполнение которых принадлежит отдельным подсистемам.

По своему назначению диаграмма состояний не является обязательным представлением в модели и как бы «присоединяется» к тому элементу, который, по замыслу разработчиков, имеет нетривиальное поведение в течение своего жизненного цикла. Наличие у системы нескольких состояний, отличающихся от простой дихотомии «исправен – неисправен», «активен – неактивен», «ожидание – реакция на внешние действия», уже служит признаком необходимости построения диаграммы состояний.

В качестве начального варианта диаграммы состояний, если нет очевидных соображений по поводу состояний объекта, можно воспользоваться подобными состояниями, в качестве составных, уточняя их (детализируя их внутреннюю структуру) по мере рассмотрения логики поведения моделируемой системы или объекта. При выделении состояний и переходов следует помнить, что длительность срабатывания отдельных переходов должна быть существенно меньшей, чем нахождение моделируемых элементов в соответствующих состояниях. Каждое из состояний должно характеризоваться определенной устойчивостью во времени.

Другими словами, из каждого состояния на диаграмме не может быть самопроизвольного перехода в какое бы то ни было другое состояние. Все переходы должны быть явно специфицированы, в противном случае построенная диаграмма состояний является либо неполной (неадекватной), либо ошибочной с точки зрения нотации языка UML (illformed). При разработке диаграммы состояний нужно постоянно следить, чтобы объект в каждый момент мог находиться только в единственном состоянии. Если это не так, то данное обстоятельство может быть, как следствием ошибки, так и неявным признаком наличия параллельности поведения у моделируемого объекта. В последнем случае следует явно специфицировать необходимое число конечных подавтоматов, вложив их в то составное состояние, которое характеризуется нарушением условия одновременности. Следует произвести обязательную проверку, чтобы никакие два перехода из одного состояния не могли сработать одновременно. Другими словами, необходимо выполнить требование отсутствия конфликтов у всех переходов, выходящих из одного и того же состояния. Наличие такого конфликта может служить признаком ошибки, либо параллельности или ветвления рассматриваемого процесса.

Если параллельность по замыслу разработчика отсутствует, то следует ввести дополнительные сторожевые условия либо изменить существующие, чтобы исключить конфликт переходов. При наличии параллельности следует заменить конфликтующие переходы одним параллельным переходом типа ветвления. Использование исторических состояний оправдано в том случае, когда необходимо организовать обработку исключительных ситуаций (прерываний) без потери данных или выполненной работы. При этом применять исторические состояния, особенно глубокие, необходимо с известной долей осторожности. Нужно помнить, что каждый из конечных подавтоматов может иметь только одно историческое состояние. В противном случае возможны ошибки, особенно, когда полуавтоматы изображаются на отдельных диаграммах состояний. Диаграммы деятельностей являются полезными при параллельном программировании. Можно графически изобразить все ветви и показать, когда их необходимо синхронизировать. Такая возможность важна также при моделировании бизнес – процессов, так как среди бизнес–процессов часто встречаются такие, которые не обязаны выполняться последовательно: диаграмма побуждает людей искать возможности делать дела параллельно. Итак, если при описа-

нии поведения системы выявляются параллельные процессы (деятельности), то их необходимо синхронизировать.

Рекомендуемые источники по теме: [4, с. 20–30].

Вопросы для самоконтроля

1. Что такое диаграмма состояний? Каковы её основные элементы?
2. Какие виды состояний существуют на диаграмме состояний?
3. Чем отличаются переходы от действий на диаграмме состояний?
4. Как правильно обозначаются события, вызывающие переход между состояниями?
5. Приведите пример системы, которую удобно моделировать с помощью диаграммы состояний.
6. В чём заключается основное назначение диаграммы состояний?
7. Какое состояние называется начальным? Как оно изображается на диаграмме?
8. Какие особенности имеет конечное состояние на диаграмме состояний?
9. Что представляет собой диаграмма деятельностей? Каковы её ключевые компоненты?
10. Какими символами на диаграмме деятельностей обозначают действия и активности?
11. Для чего используются разветвления и слияния на диаграмме деятельностей?
12. Приведите пример процесса, который удобно моделировать с помощью диаграммы деятельностей.

3.7 Раздел 2. Объектно-ориентированный анализ и проектирование ПО. Язык UML

Тема 7. Компоненты, диаграммы развёртывания

Вопросы для изучения

Компонент, Узел, пакет. Виды компонентов, отношения между компонентами. Отношения между узлами диаграммы.

Методические указания

Физическая система (physicalsystem) – реально существующий прототип модели системы.

Диаграмма компонентов, в отличие от ранее рассмотренных диаграмм, описывает особенности физического представления системы. Диаграмма компонентов обеспечивает согласованный переход от логического представления к конкретной реализации проекта в форме программного кода, позволяет определить архитектуру разрабатываемой системы, установив зависимости между программными компонентами, в роли которых может выступать исходный текст, исполняемый код и др.

Компонент (component) – физически существующая часть системы, которая обеспечивает реализацию классов и отношений, а также функционального поведения моделируемой программной системы.

Модуль (module) – часть программной системы, требующая памяти для своего хранения и процессора для исполнения.

В языке UML для компонентов определены следующие стереотипы:

– <<file>> (файл) – определяет наиболее общую разновидность компонента, который представляется в виде произвольного физического файла.

– <<executable>> (исполнимый) – определяет разновидность компонента-файла, который является исполнимым файлом.

– <<document>> (документ) – определяет разновидность компонента-файла, который представляется в форме документа произвольного содержания, не являющегося исполнимым файлом или файлом с исходным текстом программы.

– <<library>> (библиотека) – определяет разновидность компонента-файла, который представляется в форме динамической или статической библиотеки.

– <<source>> (источник) – определяет разновидность компонента-файла, представляющего собой файл с исходным текстом программы, который после компиляции может быть преобразован в исполнимый файл.

– <<table>> (таблица) – определяет разновидность компонента, который представляется в форме таблицы базы данных.

Диаграмма развертывания (deploymentdiagram) – диаграмма, на которой представлены узлы выполнения программных компонентов реального времени, а также процессов и объектов. Диаграмма развертывания применяется для представления общей конфигурации и топологии распределенной программной системы и содержит изображение размещения компонентов по отдельным узлам системы.

Представляются только те компоненты программы, которые являются исполнимыми файлами или динамическими библиотеками.

Узел (node) представляет собой физически существующий элемент системы, который может обладать вычислительным ресурсом или являться техническим устройством. Ресурсоемкий узел (processor), под которым понимается узел с процессором и памятью, необходимыми для выполнения исполняемых компонентов. Он изображается в форме куба с боковыми гранями, окрашенными в серый цвет. Второй стереотип в форме обычного куба обозначает устройство (device), под которым понимается узел без процессора и памяти (б).

Пакет (package) – общецелевой механизм для организации различных элементов модели в множество, реализующий системный принцип декомпозиции модели сложной системы и допускающий вложенность пакетов друг в друга.

Модель является подклассом пакета и представляет собой абстракцию физической системы, которая предназначена для вполне определенной цели.

Сложные программные системы могут реализовываться в сетевом варианте, на различных вычислительных платформах и технологиях доступа к распределенным базам данных. Наличие локальной корпоративной сети требует решения целого комплекса дополнительных задач рационального размещения компонентов по узлам этой сети, что определяет общую производительность программной системы. Интеграция программной системы с интернетом определяет необходимость решения дополнительных вопросов при проектировании системы, таких как обеспечение безопасности и устойчивости доступа к информации для корпоративных клиентов. Эти аспекты в немалой степени зависят от реализации проекта в форме физически существующих узлов системы, таких как серверы, рабочие станции, брандмауэры, каналы связи и хранилища данных. Технологии доступа и манипулирования данными в рамках общей схемы «клиент–сервер» также требуют размещения больших баз данных в различных сегментах корпоративной сети, их резервного копирования, архивирования, кэширования для обеспечения необходимой производительности системы в целом. С целью спецификации программных и технологических особенностей реализации распределенных архитектур необходимо визуальное представление этих аспектов. Первой из диаграмм физического представления является диаграмма компонентов. Вторая форма физического представления программной системы – это диаграмма развертывания (размещения).

Диаграмма развертывания (Deploymentdiagram) – диаграмма, на которой представлены узлы выполнения программных компонентов реального времени, а также процессов и объектов. Диаграмма развертывания применяется для представления общей конфигурации и топологии распределенной программной системы и содержит изображение размещения компонентов по отдельным узлам системы. Кроме того, диаграмма развертывания показывает наличие физических соединений – маршрутов передачи информации между аппаратными устройствами, задействованными в реализации системы.

Рекомендуемые источники по теме: [2, гл. 2, 3, гл. 2].

Вопросы для самоконтроля

1. Что такое диаграмма компонентов в UML?
   – Опишите основное назначение этой диаграммы.
   – Какие элементы она включает?
2. Какие виды компонентов существуют в UML?
   – Приведите примеры каждого вида компонента.
   – Чем отличаются исполняемые компоненты от библиотечных?
3. Опишите отношения между компонентами в UML.
   – Какие типы отношений между компонентами вы знаете?
   – Что означает отношение зависимости между компонентами?
   – Какое значение имеет отношение композиции?

4. Что такое узел в UML?

– Объясните разницу между узлом и компонентом.

– Какие типы узлов бывают?

5. Каковы отношения между узлами в UML?

– Перечислите возможные типы связей между узлами.

– Что представляет собой физическое развертывание узла?

6. Для чего используется диаграмма пакетов в UML?

– Поясните, какие элементы входят в состав пакета.

– В каких случаях применяется эта диаграмма?

7. Приведите пример использования компонентов в реальной системе.

– Описывая конкретный проект, приведите пример декомпозиции системы на компоненты.

– Какой вид связи будет использоваться между этими компонентами?

8. Объясните связь между узлами и компонентами.

– Может ли один узел содержать несколько компонентов?

– Можно ли считать узлы контейнерами для компонентов?

9. Перечислите преимущества использования диаграмм компонентов и узлов в процессе разработки ПО.

– Почему важно выделять эти элементы в архитектуре системы?

– Как помогает диаграмма компонентов улучшить понимание структуры проекта?

10. Почему важна стандартизация в описании архитектуры системы через UML-диаграммы?

– Как использование унифицированного языка моделирования влияет на эффективность коммуникации внутри команды разработчиков?

Рекомендации по выполнению расчетно-графической работы (РГР)

Учебным планом предусмотрено выполнение одной расчетно-графической работы.

Расчетно-графическая работа (РГР) направлена на закрепление полученных теоретических знаний и приобретение умений и навыков в области проектирования наукоемкого программного обеспечения.

Задание:

1. Разработать программу прикладных вычислений с применением в соответствии с индивидуальным вариантом. Программа должна быть разработана с применением технологии разработки прикладных программных средств.

2. Подготовить набор тестовых данных для проверки работоспособности программы.

3. Протестировать программу.

4. Оформить отчет из следующих проектных документов:

– техническое задание, содержащее сведения о назначении программы, требованиях к программному изделию;

– эскизный проект, содержащий проект структуры программы, разработанный методом нисходящего проектирования, или любым другим методом, применяемым в технологии программирования;

– технический проект, содержащий требования к компонентам, входящим в структуру программы, любым методом разработки спецификаций, применяемым в технологии программирования;

– алгоритмы всех программных компонентов и структуры данных всех информационных компонентов, входящих в структуру программы, разработанные методом пошаговой детализации или любым другим методом, применяемым в технологии программирования;

– рабочий проект, содержащий исходные тексты всех программных компонентов отлаженной программы и описание контрольного примера со всеми тестовыми данными и результатами вычислений для этих тестов.

Типовые темы РГР

1. Вычисление взаимного положения плоскостей, прямых и точек в пространстве.

2. Кодирование и декодирование текста по заданному методу.

3. Поиск различных значений в матрицах произвольного размера.

4. Перевод чисел между произвольными системами счисления.

Краткие сведения по использованию языка Пайтон

Развитие современных информационных технологий напрямую связано с развитием языков программирования. Python – современный мощный высокоуровневый язык программирования с поддержкой объектно-ориентированного подхода.

Легкость в изучении, удобочитаемость, скорость разработки – далеко не полный перечень достоинств, делающих Python таким популярным. Такие крупные компании как Google, Yandex, Intel, HP, IBM, NASA используют в своих разработках Python. В данной теме будут рассмотрены тематические блоки:

– выбор среды разработки и создание программ;

– переменные, типы данных, ввод/вывод информации;

– управляющие конструкции;

– обработка исключительных ситуаций;

– пользовательские функции, пакеты, модули;

– строки и обработка текстовой информации;

– списки, кортежи, словари.

Язык программирования Python – современный мощный высокоуровневый язык программирования, созданный голландским программистом Гвидо Ван Россумом и представленный в 1991 г.

Официальный сайт языка программирования Python – http://python.org. Python можно рассматривать как − интерпретируемый язык: исходный код на Python не компилируется в машинный код, а выполняется непосредственно с

помощью специальной программы-интерпретатора; – интерактивный язык: можно писать код прямо в оболочке интерпретатора и вводить новые команды по мере выполнения предыдущих; – объектно-ориентированный язык: Python поддерживает принципы объектно-ориентированного программирования, которые подразумевают инкапсуляцию кода в особые структуры, именуемые объектами.

Особенности языка Python – легкий для изучения: в Python немного ключевых слов, простая структура и четко определенный синтаксис, что позволяет научиться основам языка за достаточно короткое время; легко читаемый: блоки кода в Python выделяются при помощи отступов, что совместно с ключевыми словами, взятыми из английского языка, значительно облегчает его чтение;

– простота обслуживания кода;

– наличие широкой кросс-платформенной библиотеки;

– наличие интерактивного режима позволяет «на лету» тестировать нужные участки кода;

– портативность: Python без проблем запускается на разных плат формах, сохраняя при этом одинаковый интерфейс;

– расширяемость: при необходимости в Python можно внедрять низкоуровневые модули, написанные на других языках программирования, для наиболее гибкого решения задач;

– работа с базами данных: в стандартной библиотеке Python можно найти модули для работы с большинством баз данных;

– создание GUI (графического интерфейса пользователя): предусмотрена возможность создания GUI приложений с самым широким спектром пользовательских настроек.

Недостатки Python Скорость выполнения программ несколько ниже по сравнению с компилируемыми языками (C или C++), поскольку Python транслирует инструкции исходного программного кода в байт-код, а за тем интерпретирует этот байт-код. Байт-код обеспечивает переносимость программ, поскольку это платформо-независимый формат.

Как результат, имеет место преимущество в скорости разработки с потерей скорости выполнения.

В наше время используют Python:

– Компания Google оплачивает труд создателя Python и использует язык в поисковых системах;

– сервис YouTube;

– компания Yandex в ряде сервисов;

– программа BitTorrent написана на Python;

– веб-фреймворкAppEngine от Google использует Python в качестве прикладного языка программирования;

–Intel, Cisco, HP, IBM используют Python для тестирования аппаратного обеспечения;

– NASA использует язык в научных вычислениях.

4. МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ВЫПОЛНЕНИЮ ЛАБОРАТОРНЫХ РАБОТ

Лабораторные занятия направлены на углубление знаний и закрепление основных понятий и методов, изучаемых в рамках дисциплины. Основная цель занятий – научиться применять теоретические знания для решения прикладных задач.

Рекомендации к подготовке

1. Изучение лекционного материала и конспектов:

– Перед лабораторными занятиями необходимо проработать соответствующие разделы лекционного материала.

– Рекомендуется вести конспект занятий и дополнительно пересмотреть его перед началом лабораторного занятия.

2. Проработка учебной литературы:

– Используйте основные учебники и методические пособия, рекомендованные преподавателем.

– Для глубокого понимания теории рекомендуется обращаться к дополнительной литературе.

3. Подготовка вопросов:

– Составьте список вопросов по материалу, вызвавшему затруднения.

– Обсуждение этих вопросов на лабораторном занятии поможет устранить пробелы в знаниях.

4. Вопросы для самоконтроля:

– Перед каждым занятием рекомендуется проверять себя с помощью вопросов для самоконтроля из методических указаний к лекционным занятиям. Это позволит оценить уровень своей подготовки.

В этом разделе приведены общие рекомендации. Информация, которая касается тематического плана выполнения лабораторных работ приведена в таблице 1 данного документа. Подробная информация по выполнению лабораторных работ приведена в УМП по выполнению лабораторных работ по дисциплине «Проектирование и разработка наукоёмкого программного обеспечения».

5. МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО САМОСТОЯТЕЛЬНОЙ РАБОТЕ

Внеаудиторная самостоятельная работа в рамках данной дисциплины включает в себя:

- подготовку к аудиторным занятиям (лекциям, лабораторным занятиям) и выполнение соответствующих заданий;
- самостоятельную работу над отдельными темами учебной дисциплины в соответствии с тематическим планом;
- подготовку к Экзамену.

Подготовка к лекционным занятиям

При подготовке к лекции рекомендуется повторить ранее изученный материал, что дает возможность получить необходимые разъяснения преподавателя непосредственно в ходе занятия. Рекомендуется вести конспект, главное требование к которому быть систематическим, логически связанным, ясным и кратким. По окончанию занятия обязательно в часы самостоятельной подготовки, по возможности в этот же день, повторить изучаемый материал и доработать конспект.

Подготовка к лабораторным занятиям

Подготовка к лабораторным занятиям предусматривает:

- изучение теоретических положений, лежащих в основе будущих расчетов или методики расчетов;
- детальную проработку учебного материала, рекомендованной литературы и методической разработки на предстоящее занятие;

Самостоятельная работа над отдельными темами учебной дисциплины

При организации самостоятельного изучения ряда тем лекционного курса обучаемый работает в соответствии с указаниями, выданными преподавателем. Указания по изучению теоретического материала курса составляются дифференцированно по каждой теме и включают в себя следующие элементы: название темы; цели и задачи изучения темы; основные вопросы темы; характеристику основных понятий и определений, необходимых обучаемому для усвоения данной темы; список рекомендуемой литературы; наиболее важные фрагменты текстов рекомендуемых источников, в том числе таблицы, рисунки, схемы и т. п.; краткие выводы, ориентирующие обучаемого на определенную совокупность сведений, основных идей, ключевых положений, систему доказательств, которые необходимо усвоить; контрольные вопросы, предназначенные для самопроверки знаний.

Подготовка к выполнению расчетно-графической работы (РГР)

Выполнить на оценку «зачтено» РГР помогает последовательное выполнение заданий в УМП по лабораторным работам.

Выполнение этих заданий из лабораторных работ 1 и 2 поможет при выполнении РГР.

Подготовка к экзамену

При подготовке к экзамену большую роль играют правильно подготовленные заранее записи и конспекты, также вовремя выполненные лабораторные задания.

В ходе самостоятельной подготовки к экзамену при анализе имеющегося теоретического и практического материала студенту также рекомендуется проводить постановку различного рода задач по изучаемой теме, что поможет в дальнейшем выявлять критерии принятия тех или иных решений, причины совершения определенного рода ошибок. При ответе на вопросы, поставленные в ходе самостоятельной подготовки, обучающийся вырабатывает в себе способность логически мыслить, искать в анализе событий причинно-следственные связи.

Вопросы и задачи для самопроверки

1. Что такое наукоемкое программное обеспечение? Какие характеристики отличают его от обычного программного обеспечения?
2. Почему Python часто используется для разработки наукоемких приложений? Перечислите ключевые преимущества языка Python в этой области.
3. Какие особенности синтаксиса Python делают его удобным для научных расчетов и анализа данных?
4. Назовите основные библиотеки Python, используемые для работы с числовыми расчетами и статистическим анализом. Приведите примеры использования каждой из них.
5. Каковы различия между библиотеками NumPy и Pandas? В каких случаях лучше использовать каждую из них?
6. Что такое JupyterNotebook и как он помогает в разработке наукоемкого ПО?
7. Объясните концепцию векторизации в контексте работы с массивами данных. Как она улучшает производительность кода?
8. Что такое параллельные вычисления и как Python поддерживает их реализацию?
9. Опишите процесс интеграции Python-кода с другими языками программирования (например, C/C++). Какие инструменты используются для этого?
10. Как организовать тестирование научного ПО на Python? Какие фреймворки тестирования наиболее популярны среди разработчиков?

11. Расскажите о роли модульного подхода в разработке наукоемкого ПО. Какие принципы SOLID применимы к проектированию Python-приложений?

12. Что такое непрерывная интеграция (CI/CD)? Как эта практика применяется в разработке наукоемкого ПО?

13. Какие подходы к управлению версиями применяются при работе над крупными проектами на Python? В чем заключаются преимущества системы контроля версий Git?

14. Каким образом Python позволяет создавать интерактивные веб-приложения для визуализации данных? Приведите примеры соответствующих библиотек.

15. Какие проблемы безопасности могут возникать при разработке наукоемкого ПО на Python? Какими способами можно минимизировать риски?

16. Как организована работа с большими объемами данных в Python? Какие существуют способы оптимизации производительности в таких ситуациях?

17. Какие методы документирования кода применяются в крупных проектах на Python? Опишите стандартные практики и популярные инструменты.

18. Что такое рефакторинг кода и зачем он необходим в процессе разработки наукоемкого ПО?

19. Какие архитектурные паттерны чаще всего используются при создании сложных научных приложений на Python?

20. Как происходит развертывание и поддержка наукоемкого ПО после завершения разработки?

Практические задания для самостоятельной подготовки

1. Напишите программу на Python, которая генерирует случайную выборку данных из нормального распределения и строит график плотности вероятности.

2. Реализуйте функцию, выполняющую матричное умножение с использованием библиотеки NumPy. Сравните её производительность с циклическим подходом.

3. Разработайте скрипт для анализа временных рядов с использованием библиотеки Pandas. Постройте графики основных метрик (среднее значение, дисперсия).

4. Создайте простое веб-приложение с помощью Flask, которое отображает интерактивный график на основе загружаемых пользователем данных.

5. Напишите тесты для проверки корректности работы функции, реализующей метод наименьших квадратов для линейной регрессии.

6. Разверните локально CI/CD пайплайн для проекта на Python с использованием GitLab CI.

Эти вопросы помогут углубленно изучить тему и подготовиться к практической реализации проектов на Python в научной сфере.

6. КОНТРОЛЬ И АТТЕСТАЦИЯ

К оценочным средствам текущего контроля успеваемости относятся:

– тестовые задания открытого и закрытого типов;

– расчетно-графическая работа.

Контроль РГР происходит в ЭИОС лектором.

Критерий оценки РГР: «зачтено», «не зачтено»

К оценочным средствам для промежуточной аттестации в форме экзамена во втором семестре относятся:

– экзаменационные задания по дисциплине, представленные в виде тестовых заданий закрытого и открытого типов.

Критерии оценки результатов освоения дисциплины Универсальная система оценивания результатов обучения включает в себя системы оценок: 1) «отлично», «хорошо», «удовлетворительно», «неудовлетворительно»; 2) «зачтено», «не зачтено»; 3) 100-балльную/процентную систему и правило перевода оценок в пятибалльную систему (таблица 2).

Таблица 2 – Система оценок и критерии выставления оценки

Система оценок / **Критерий**	**2**	**3**	**4**	**5**
	0–40 %	**41–60 %**	**61–80 %**	**81–100 %**
	«неудовлетворительно»	**«удовлетворительно»**	**«хорошо»**	**«отлично»**
	«незачтено»	**«зачтено»**		
1 Системность и полнота знаний в отношении изучаемых объектов	Обладает частичными и разрозненными знаниями, которые не может научно-корректно связывать между собой (только некоторые из которых может связывать между собой)	Обладает минимальным набором знаний, необходимым для системного взгляда на изучаемый объект	Обладает набором знаний, достаточным для системного взгляда на изучаемый объект	Обладает полнотой знаний и систем ным взглядом на изучаемый объект
2 Работа с информацией	Не в состоянии находить необходимую информацию, либо в состоянии находить отдельны фрагменты информации в рамках поставленной задачи	Может найти необходимую информацию в рамках поставленной задачи	Может найти, интерпретировать и систематизировать необходимую информацию в рамках поставленной задачи	Может найти, систематизировать необходимую информацию, а также выявить новые, дополнительные источники информации в рамках поставленной задачи

Система оценок / Критерий	2	3	4	5
	0–40 %	41–60 %	61–80 %	81–100 %
	«неудовлетворительно»	«удовлетворительно»	«хорошо»	«отлично»
	«незачтено»	«зачтено»		
3 Научное осмысление изучаемого явления, процесса, объекта	Не может делать научно-корректных выводов из имеющихся у него сведений, в состоянии проанализировать только некоторые из имеющихся у него сведений	В состоянии осуществлять научно-корректный анализ предоставленной информации	В состоянии осуществлять систематический и научно-корректный анализ предоставленной информации, вовлекает в исследование новые релевантные задаче данные	В состоянии осуществлять систематический и научно-корректный анализ предоставленной информации, вовлекает в исследование новые релевантные поставленной задаче данные, предлагает новые ракурсы поставленной задачи
4 Освоение стандартных алгоритмов решения профессиональных задач	В состоянии решать только фрагменты поставленной задачи в соответствии с заданным алгоритмом, не освоил предложенный алгоритм, допускает ошибки	В состоянии решать поставленные задачи в соответствии с заданным алгоритмом	В состоянии решать поставленные задачи в соответствии с заданным алгоритмом, понимает основы предложенного алгоритма	Не только владеет алгоритмом и понимает его основы, но и предлагает новые решения в рамках поставленной задачи

Результаты текущего контроля успеваемости оцениваются

Тестовые задания открытого типа оцениваются по системе «зачтено/ не зачтено». Оценивается верность ответа по существу вопроса, при этом не учитывается порядок слов в словосочетании, верность окончаний, падежи.

Оценивание тестовых заданий закрытого типа осуществляется по системе зачтено/не зачтено («зачтено» – 41–100 % правильных ответов; «не зачтено» – менее 40 % правильных ответов) или пятибалльной системе (оценка «неудовлетворительно» – менее 40 % правильных ответов; оценка «удовлетворительно» – от 41 до 60 % правильных ответов; оценка «хорошо» – от 61 до 80% правильных ответов; оценка «отлично» – от 81 до 100 % правильных ответов).

7. СПИСОК ЛИТЕРАТУРЫ

Основные источники

1. Перл, И. А. Введение в методологию программной инженерии: учеб. пособие / И. А. Перл, О. В. Калёнова. – Санкт-Петербург: Университет ИТМО, 2019. – 53 с. – Режим доступа: по подписке. – URL: https://biblioclub.ru/index.php?page=book&id=566776 (дата обращения: 04.06.2024). – Бибиогр. в кн. – Текст : электронный.

2. Доррер, Г. А. Методология программной инженерии: учеб. пособие / Г. А. Доррер. – Красноярск: СибГУ им. академика М. Ф. Решетнёва, 2021. – 190 с. – Текст : электронный // Лань : электронно-библиотечная система. – URL: https://e.lanbook.com/book/195097 (дата обращения: 04.06.2024). – Режим доступа: для авториз. пользователей.

Дополнительные источники

3. Зубкова, Т. М. Технология разработки программного обеспечения: учеб. пособие / Т. М. Зубкова. – Санкт-Петербург: Лань, 2022. – 324 с. – ISBN 978-5-8114-3842-6. – Текст : электронный // Лань : электронно-библиотечная система. – URL:https://e.lanbook.com/book/206882 (дата обращения: 04.06.2024). – Режим доступа: для авториз. пользователей.

4. Дукельский, К. В. Управление качеством программного обеспечения: учеб. пособие / К. В. Дукельский, И. Б. Бондаренко. – Санкт-Петербург: СПбГУТ им. М. А. Бонч-Бруевича, 2021. – 52 с. – Текст : электронный // Лань : электронно-библиотечная система. – URL: https://e.lanbook.com/book/279632 (дата обращения: 04.06.2024). – Режим доступа: для авториз. пользователей.

5. Романов, Е. Л. Программная инженерия: учеб. пособие / Е. Л. Романов. – Новосибирск: НГТУ, 2017. – 395 с. ISBN 978-5-7782-3455-0. – Текст : электронный // Лань : электронно-библиотечная система. – URL: https://e.lanbook.com/book/118221 (дата обращения: 04.06.2024). – Режим доступа: для авториз. пользователей.

Учебно-методические пособия, нормативная литература

6. Рудинский, И. Д. Технология проектирования автоматизированных систем обработки информации и управления: учеб.-метод. пособие по выполнению курсового проекта для студентов, обучающихся в бакалавриате по направлению подгот. «Информатика и вычисл. Техника» / И. Д. Рудинский; Калинингр. гос. техн. ун-т. – Калининград: КГТУ, 2015. – 97 с. – Текст : непосредственный.

Локальный электронный методический материал

Наталья Яронимо Великите

ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА НАУКОЁМКОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Редактор С. Кондрашова
Корректор Т. Звада

Уч.-изд. л. 3,9. Печ. л. 3,2.

Издательство федерального государственного бюджетного
образовательного учреждения высшего образования
«Калининградский государственный технический университет».
236022, Калининград, Советский проспект, 1