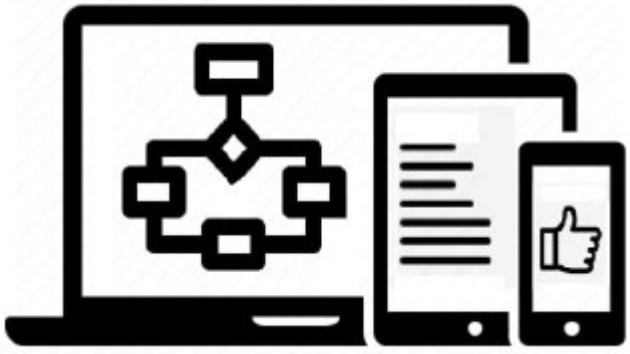
МЕЖГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛОРУССКО-РОССИЙСКИЙ УНИВЕРСИТЕТ»

Кафедра «Программное обеспечение информационных технологий»

ТЕХНОЛОГИИ КОМАНДНОЙ РАЗРАБОТКИ ПРИЛОЖЕНИЙ

Методические рекомендации к лабораторным работам для студентов направления подготовки 09.03.04 «Программная инженерия» очной формы обучения



Могилев 2025

УДК 004.4(07)
ББК 32.973я73
Т38

Рекомендовано к изданию
учебно-методическим отделом
Белорусско-Российского университета

Одобрено кафедрой «Программное обеспечение информационных технологий» «10» сентября 2025 г., протокол № 2

Составители: канд. техн. наук, доц. К. В. Захарченков;
канд. техн. наук, доц. Т. В. Мрочек

Рецензент Ю. С. Романович

Приведены методические рекомендации к лабораторным работам для студентов направления подготовки 09.03.04 «Программная инженерия» по дисциплине «Технологии командной разработки приложений».

Учебное издание

ТЕХНОЛОГИИ КОМАНДНОЙ РАЗРАБОТКИ ПРИЛОЖЕНИЙ

Ответственный за выпуск	В. В. Кутузов
Корректор	А. Т. Червинская
Компьютерная верстка	М. М. Дударева

Подписано в печать 23.12.2025. Формат 60×84/16. Бумага офсетная. Гарнитура Таймс. Печать трафаретная. Усл. печ. л. 1,86. Уч.-изд. л. 1,94. Тираж 21 экз. Заказ № 911.

Издатель и полиграфическое исполнение:
Межгосударственное образовательное учреждение высшего образования
«Белорусско-Российский университет».
Свидетельство о государственной регистрации издателя,
изготовителя, распространителя печатных изданий
№ 1/156 от 07.03.2019.
Пр-т Мира, 43, 212022, г. Могилев.

Содержание

Введение 4
1 Разработка и анализ требований к программной системе 5
2 Спецификации программной системы 10
3 Испытания программных систем 16
4 Использование систем автоматизации разработки программ 18
5 Компонентное программирование 24
Список литературы 30

Введение

Современное состояние науки и техники требует от инженерно-технических и научных работников знания средств вычислительной техники и умения обращения с современными программно-техническими комплексами. Эффективное применение компьютеров для решения инженерных и научных задач невозможно без знаний основных методов составления схем алгоритмов, написания действенного программного обеспечения на языке программирования, использования пакетов программ инженерной графики и математических систем.

Целью изучения дисциплины является освоение системы инженерных принципов проектирования, разработки и тестирования программного обеспечения с заданными характеристиками.

Цели лабораторного цикла работ:

– участие в проектировании компонентов программного продукта в объеме, достаточном для их конструирования в рамках поставленного задания;

– создание компонент программного обеспечения (кодирование, отладка, модульное и интеграционное тестирование);

– выполнение измерений и рефакторинг кода в соответствии с планом;

– участие в интеграции компонент программного продукта;

– разработка и оформление эскизной, технической и рабочей проектной документации.

После выполнения каждой лабораторной работы студент оформляет отчет. Отчет по лабораторной работе должен содержать название и цель работы, краткий порядок действий, ответы на контрольные вопросы.

При выполнении лабораторных работ рекомендуется использовать литературные источники [1–11].

Полученные при изучении дисциплины знания и навыки могут быть востребованы при выполнении курсового проекта и в дальнейшем процессе обучения студента в вузе.

1 Разработка и анализ требований к программной системе

Цель: составить документ описания требований к разрабатываемой программной системе.

Теоретические положения

Документ описания требований. Документ, описывающий требования, является осязаемым результатом этапа установления требований. Большинство организаций вырабатывает документ описания требований в соответствии с заранее определенным шаблоном. Шаблон определяет структуру (содержание) и стиль документа.

Ядро документа описания требований состоит из формулировок (изложения) требований. Требования могут быть сгруппированы в виде формулировок сервисов (зачастую называемых функциональными требованиями) и формулировок ограничений. Формулировки сути сервисов могут быть затем разделены на требования к функциям (function requirements) и требования к данным (data requirements). В литературе термин «функциональные требования» (functional requirements) в широком и в узком смысле используется как взаимозаменяемый. При использовании в узком смысле он соответствует тому, что мы называем требованиями к функциям.

Модели состояний «детализируют» требования к данным. Модели поведения обеспечивают детализированные спецификации для функциональных требований. Модели изменения состояний охватывают два вида требований. Они позволяют объяснить и наглядно представить, каким образом действие функций приводит к изменению данных.

Модели представляются в виде диаграмм на языке визуального моделирования (Visual Modeling Language) – в нашем случае это язык UML (унифицированный язык моделирования (Unified Modeling Language)). Обычно диаграмма служит целям моделирования одной из сторон системы – состояний, поведения или изменения состояний. Заметное исключение составляет диаграмма классов, которая определяет все три аспекта – состояние и поведение объектов и, косвенно, изменения состояний объектов.

Каждая диаграмма дает представление об определенной стороне системы. Взятые вместе диаграммы дают возможность разработчикам и пользователям взглянуть на предлагаемое решение с разных точек зрения, выделяя одни его стороны и игнорируя другие. Ни одна из диаграмм в отдельности не дает полного определения системы. Систему можно понять только через взаимосвязанный набор диаграмм.

Аналогично случаю интерпретации завершенных моделей конструирование диаграмм – это не последовательный процесс построения одной диаграммы за другой.

Диаграммы разрабатываются параллельно, и в результате каждой последующей итерации к ним добавляются новые детали. В то время как разработчи-

ки должны следовать строго определенному процессу разработки, решение о том, какая из моделей должна играть роль «движущей силы» разработки, в значительной мере зависит от личных предпочтений аналитика. Обычно диаграммы прецедентов и модели классов – как наиболее важные типы моделей – конструируются параллельно, взаимно «обогащая» друг друга идеями.

С каждой новой итерацией разработки глубина и степень детализации спецификации возрастает. Многие более глубокие свойства объектов модели выражаются скорее в текстовом, нежели графическом виде. Некоторые свойства определяют замысел объекта модели, а не результат анализа. Некоторые другие свойства могут отражать особенности CASE-средств.

Спецификации состояний. Состояние объекта определяется значениями его атрибутов и ассоциаций.

В типичной ситуации сначала определяются классы-сущности, т. е. классы, которые определяют проблемную область и характеризуются постоянным присутствием в базе данных системы. Подобные классы иногда называются «бизнес-классами». Классы, которые обслуживают системные события (управляющие классы), и классы, которые представляют GUI-интерфейс (классы представления или пограничные классы), не устанавливаются до тех пор, пока не станут известны поведенческие характеристики системы.

Выявление классов. Два разных аналитика, как правило, не могут прийти к идентичным моделям классов для одной и той же проблемной области, и точно так же два разных аналитика не пользуются одним и тем же мыслительным процессом при выделении классов. Литература изобилует подходами, предлагаемыми для выявления классов. Аналитики могут поначалу даже следовать одному из этих подходов, однако последующие итерации, как правило, обязательно приводят к использованию нешаблонных и в чем-то даже случайных механизмов. Ниже перечислены эти подходы.

1 Подход на основе использования именных групп.

2 Подход на основе использования общих шаблонов для классов.

3 Подход на основе использования прецедентов.

4 Подход CRC (class – responsibility – collaborators – класс – обязанности – «сотрудники»).

Некоторые правила выявления классов. Ниже приведен далеко не полный перечень руководящих принципов или правил, которым должен следовать аналитик при выборе потенциальных классов (здесь работа ведется только с классами-сущностями).

1 Для каждого класса должно быть ясно сформулировано его назначение в системе.

2 Каждый класс – это шаблон описания множества объектов. Единичные классы, для которых можно представить существование только одного объекта, весьма маловероятны среди «бизнес-объектов». Подобные классы обычно составляют в приложении «общее знание» и, как правило, жестко запрограммированы в программах приложения. Например, если система спроектирована для единственной организации, существование класса Organization (Организация) может быть не оправдано.

3 Каждый класс (т. е. класс-сущность) должен содержать набор атрибутов. Хорошим приемом является установление идентифицирующих атрибутов (ключей), чтобы помочь нам судить о мощности (cardinality) класса (т. е. ожидаемом количестве объектов данного класса в базе данных). Следует помнить о том, что класс необязательно должен обладать пользовательским ключом.

4 Каждый класс должен отличаться от атрибута. Представляется ли понятие классом или атрибутом зависит от области приложения.

Спецификация классов. После того как перечень потенциальных классов сформирован, необходима их дальнейшая спецификация: классы требуется включить в диаграмму классов и определить их свойства. Некоторые свойства можно ввести и отобразить внутри графических пиктограмм, представляющих классы на диаграмме классов. Многие другие свойства, включенные в спецификацию класса, имеют только текстовое представление. CASE-средства, как правило, обладают возможностями редактирования, позволяющими легко вводить или модифицировать подобную информацию посредством диалоговых окон, снабженных вкладками, или с помощью аналогичных способов.

Выявление и спецификация атрибутов классов.

Графическая пиктограмма, представляющая класс, состоит из трех отделений (имя класса, атрибуты, операции). Спецификация атрибутов классов принадлежит к спецификации состояний и рассматривается в этом разделе.

Выделение атрибутов осуществляется параллельно с выделением классов. Идентификация атрибутов – своего рода «побочный эффект» установления классов. Это не означает, что выявление атрибутов – простая задача. Напротив, это процесс, требующий значительных усилий и многократных итераций.

Исходные модели спецификации определяют только атрибуты, являющиеся существенными для понимания состояний, в которых могут находиться объекты класса. Остальные атрибуты можно до поры до времени игнорировать (однако аналитик должен быть уверен в том, что установленная, но проигнорированная на определенном этапе информация не будет по ошибке утеряна и будет зафиксирована впоследствии). Маловероятно, чтобы все атрибуты класса были приведены в документе описания требований, однако важно не включать в спецификацию те атрибуты, которые не вытекают из требований.

В последующих итерациях можно добавить больше атрибутов.

Для имен атрибутов рекомендуется придерживаться простого соглашения: в именах атрибутов использовать только строчные буквы, а слова в составных именах отделять подчеркиванием.

Выявление ассоциаций. Нахождение основных ассоциаций представляет собой побочный эффект процесса выявления классов. При определении классов аналитик принимает решение об атрибутах классов, и некоторые из этих атрибутов являются ассоциациями с другими классами. Атрибуты могут относиться к элементарным типам данных либо могут вводиться в качестве других классов, устанавливая таким образом отношения с другими классами. Любой атрибут, относящийся к неэлементарным типам данных, должен моделироваться как ассоциация (или агрегация) по классу, представляющему этот тип данных.

Выполнение пробного прогона прецедентов позволяет выявить остающиеся ассоциации. Устанавливаются пути взаимодействия между классами, необходимые для прогона прецедентов. Обычно ассоциации должны поддерживать эти пути взаимодействия.

Каждая тернарная ассоциация должна быть заменена циклом или бинарной ассоциацией. Тернарные ассоциации привносят риск неверного семантического истолкования.

Иногда для того, чтобы полностью выразить базовую семантику, циклы, образуемые ассоциациями, не должны коммутировать (быть замкнутыми). Это значит, что по меньшей мере одна из ассоциаций в цикле может быть производной (derived). Подобная ассоциация является избыточной в семантическом смысле и должна быть исключена (хорошая семантическая модель должна быть лишена избыточности). Вполне допустимо, что многие производные ассоциации все же войдут в проектную модель (например, из соображений эффективности).

Спецификация ассоциаций. Спецификация ассоциаций подразумевает выполнение следующих действий.

1 Присваивание имен ассоциациям.

2 Присваивание имен ассоциативным ролям.

3 Установление кратности ассоциации.

Правила именования ассоциаций должны соответствовать соглашениям по именованию атрибутов – имена ассоциаций состоят из строчных букв, отдельные слова в имени ассоциации разделяются подчеркиванием.

Если два класса связаны только одним ассоциативным отношением, задавать имя ассоциации и ассоциативные ролевые имена между этими классами необязательно. CASE-средства могут внутренне различать каждую ассоциацию через системные идентификационные имена.

Ролевые имена можно использовать для раскрытия более сложных ассоциаций, в частности самоассоциативных отношений (self associations) (рекурсивных ассоциаций, которые связывают объекты одного и того же класса). При задании ролевых имен их следует выбирать с учетом того, что в проектной модели они станут атрибутами классов, расположенных на противоположных концах ассоциативной связи.

Выявление агрегаций и композиций. Поиск агрегаций ведется параллельно с поиском ассоциаций. Если ассоциация проявляет одно или более из четырех семантических свойств, которые рассмотрены выше, то ее можно моделировать как агрегацию.

При объяснении отношения агрегации лакмусовой бумажкой выступают фразы «включает» («has») и «является частью» («is_part_of»). При истолковании отношения сверху-вниз по иерархии классов используется фраза «включает» (например, Книга «включает» Главу). При интерпретации снизу-вверх используется фраза «является частью» (например, Глава «является частью» Книги). Если предложение, описывающее отношение, прочитывается вслух с использованием этих фраз и оно лишено смысла на естественном языке, то это отношение не является агрегацией. Со структурной точки зрения агрегация часто связывает воедино большое количество классов, а ассоциация степени вы-

ше двух бессмысленна. Если требуется связать более двух классов воедино, хорошим вариантом моделирования может быть агрегация типа «Участник».

Спецификация агрегаций и композиций. Язык UML обеспечивает только ограниченную поддержку агрегации. Сильная форма агрегации называется в UML композицией.

В композиции составной объект может физически содержать компонентные объекты (семантически это отношение берется «по значению»). Компонентный объект может принадлежать только одному составному объекту. Отношение композиции языка UML в большей или меньшей степени соответствует нашим агрегациям типа «Безраздельно обладает» и «Обладает».

Слабая форма агрегации в UML называется просто агрегацией. Это отношение семантически берется «по ссылке» – составной объект физически не содержит компонентный объект. Один компонентный объект может обладать несколькими ассоциативными или агрегативными связями в модели. Попросту говоря, агрегация в языке UML соответствует нашим агрегациям типа «Включает» и «Участник».

Многие суперклассы/подклассы аналитик отмечает еще в процессе формирования первоначального перечня классов. Многие другие обобщения можно обнаружить при определении ассоциаций.

Отношение обобщения между классами показывает, что один класс совместно использует структуру или поведение, определенные в одном или более классов. Обобщение представляется в языке UML сплошной линией со стреловидным наконечником, указывающим на суперкласс.

Задание

Выбрать предметную область для создания приложения. Для выбранной предметной области представить результаты анализа требований к приложению. Представить и описать диаграммы классов слоя доступа к данным, бизнес-логики и пользовательского интерфейса в виде UML-диаграмм. Описать связи ассоциаций, агрегаций и композиций, представленные на UML-диаграммах.

Содержание отчета: титульный лист; тема и цель работы; текст индивидуального задания; описание хода выполнения индивидуального задания.

Контрольные вопросы

1 Охарактеризовать состав документа описания требований.
2 Перечислить подходы к выявлению классов.
3 Перечислить правила выявления классов.
4 Как выявить и специфицировать атрибуты классов?
5 Каким образом выявить и специфицировать ассоциации?
6 Как выявить и специфицировать агрегации и композиции?

2 Спецификации программной системы

Цель: получить навыки составления спецификаций к разрабатываемой программной системе.

Теоретические положения

Спецификация требований. Требования необходимо специфицировать (т. е. задать) графически или каким-либо иным формальным способом. Всесторонняя спецификация системы может потребовать использования многих типов моделей. Язык UML изобилует интегрированными методами моделирования, способными помочь бизнес-аналитику справиться с этой задачей. Спецификация – подобно процессу разработки ПО в целом – итеративный процесс с пошаговым наращиванием уровня детализации моделей. Немаловажную роль в успешном моделировании играет использование CASE-средств.

В результате спецификации требований вырабатываются три категории моделей: модели состояний, модели поведения и модели изменения состояния. Для каждой из категорий существует несколько методов работы с ними. Далее объясняются и иллюстрируются на примерах все основные методы моделирования языка UML.

Многие модели разрабатываются параллельно и служат источником взаимного развития. Это особенно справедливо в отношении двух основополагающих типов моделей – моделей классов и моделей прецедентов.

Принципы спецификации требований. Спецификация требований связана с доскональным моделированием требований заказчиков, определенных в процессе установления требований. При этом рассматриваются только услуги, которые стремятся получить от системы заказчики (формулировки сервисов). На этапе спецификации требований формулировки ограничений не подлежат дальнейшей проработке, хотя и могут претерпеть изменения как результат обычного цикла итерации.

В качестве входной информации процесса спецификации требований выступают неформальные требования заказчиков, а результатом этого процесса являются модели спецификации проектных конструкций. Эти модели дают более формальное определение различных сторон (представлений) системы. Обычно требования пользователей в процессе спецификации разделяются на две основные категории: функциональные требования и требования к данным.

В качестве результата этапа спецификации выступает расширенный («детально проработанный») документ описания требований. Новый документ часто называют документом спецификации требований (или просто «спецификацией» на жаргоне разработчиков). Структура исходного документа не изменяется, однако содержание значительно расширяется за счет глав, которые определяют требования заказчиков. Постепенно для целей проектирования и реализации документ спецификации требований заменяет документ описания требо-

ваний (на практике расширенный документ может по-прежнему называться документом описания требований).

Модели спецификаций можно разделить на три группы:

1) модели состояний;

2) модели поведения;

3) модели изменения состояний.

Ф. Брукс следующим образом охарактеризовал роль требований в разработке программного обеспечения. Строжайшее и единственное правило построения систем программного обеспечения (ПО) – решить точно, что же строить. Никакая другая часть концептуальной работы не является такой трудной, как выяснение деталей технических требований, в том числе и взаимодействие с людьми, с механизмами и с иными системами ПО. Никакая другая часть работы так не портит результат, если она выполнена плохо. Ошибки никакого другого этапа работы не исправляются так трудно.

Наука извлечения и формализации качественных (иногда говорят «хороших», «правильных») требований носит во многом эмпирический характер. Однако в практике разработки программных систем накопились определенные представления о том, какими свойствами должны обладать требования к программной системе. К ним относятся:

- полнота;
- ясность;
- корректность;
- согласованность;
- верифицируемость;
- необходимость;
- полезность при эксплуатации;
- осуществимость;
- модифицируемость;
- трассируемость;
- упорядоченность по важности и стабильности;
- наличие количественной метрики.

Рассмотрим указанные выше свойства подробнее.

Полнота. Как известно из теории искусственного интеллекта, неполнота – одно из фундаментальных свойств человеческого знания. При создании программных систем нам приходится иметь дело с характеристиками еще несуществующей системы. Идея о том, что необходимо сформулировать все требования полностью, т. е. исчерпывающим образом, до начала проектирования, а тем более – реализации системы, изжила себя вместе с так называемым каскадным подходом, который поддерживал последовательную модель реализации системы. Спиральный подход, на котором базируется большинство современных методологий, предусматривает поэтапное выделение и детализацию требований на всем протяжении цикла разработки системы.

Тем не менее требование полноты предъявляется к требованиям, формулируемым к системе. Надо понимать, что данное требование – это скорее тенден-

ция, цель, к которой нужно постараться максимально приблизиться на как можно более ранних стадиях проекта.

Требование полноты можно рассматривать в двух аспектах: полнота отдельного требования и полнота системы требований.

Полнота отдельного требования – свойство, означающее, что текст требования не требует дополнительной детализации, т. е. в нем предусмотрены все необходимые нюансы, особенности и детали данного требования.

Полнота системы требований – свойство, означающее, что совокупность артефактов, описывающих требования, исчерпывающим образом описывает все то, что требуется от разрабатываемой системы.

Ясность (недвусмысленность, определенность, однозначность спецификаций). Каждый из совладельцев разрабатываемой системы обладает своим личным опытом восприятия событий внешнего мира. Слово, произнесенное вслух, вызывает индивидуальные ассоциации в семантическом пространстве каждого отдельного воспринимающего субъекта. То, что является ясным, допустим, для кардиохирурга, совсем необязательно будет таковым для специалиста в области программной инженерии.

Соответственно, требование обладает свойством ясности, если оно сходным образом воспринимается всеми совладельцами системы. На практике ясность требований достигается в том числе и в процессе консультаций, в ходе которых происходит «выравнивание тезаурусов» совладельцев системы. Хорошим подспорьем в этом служит согласованный сторонами глоссарий ключевых понятий предметной области.

Еще одной стороной понятия «ясность требования» является его прослеживаемость (см. также понятие трассируемости ниже по тексту). Требование, которое сформулировано ясно, может быть прослежено, начиная от того документа, где оно сформулировано впервые, вплоть до рабочих спецификаций.

Корректность и согласованность (непротиворечивость). Корректность – одно из важнейших свойств требований. Понятие корректности требования вводится через точность описания функциональности. В этом смысле корректность в определенной степени конкурирует с полнотой. Но есть и различие – если свойство полноты носит скорее качественный характер: абсолютная полнота представляет недостижимый идеал, к которому можно приближаться, то свойство корректности носит оценочный характер и задает дихотомию: каждое из требований либо корректно, либо нет. Кроме того, можно рассуждать о взаимной корректности требований или согласованности (непротиворечивости): если два требования вступают в конфликт, значит, как минимум одно из них некорректно. В иерархии требований можно выделить вертикальную и горизонтальную согласованность. То есть требования ниже не должны противоречить, соответственно, требованиям своего уровня иерархии и требованиям «родительского» уровня. Так, требования пользователей не должны противоречить бизнес-требованиям, а функциональные требования – требованиям пользователя.

Верифицируемость (пригодность к проверке). Признаки (свойства) требований, рассматриваемые в данной теме, нельзя считать независимыми. В математической статистике такие признаки называются коррелируемыми.

Так, свойство верифицируемости существенно связано со свойствами ясности и полноты: если требование изложено на языке, понятном и одинаково воспринимаемым участниками процесса создания приложения, причем оно является полным, т. е. ни одна из важных для реализации деталей не упущена, значит это требование можно проверить. При этом в ходе проверки у сторон (принимающей и сдающей работу) не должно возникнуть неразрешимых противоречий в оценках. Так как хорошо сформулированные требования составляют основу успешного создания системы, то роль верифицируемости трудно переоценить. Требования к системе представляют основу контракта между Заказчиком и Исполнителем, и если данные требования нельзя проверить, то и контракт не имеет никакого смысла, следовательно, успех или неудача проекта будут зависеть только от эмоциональных оценок сторон и их способности договориться, а это слишком шаткая основа для осуществления работ.

Необходимость и полезность при эксплуатации. Одни из самых субъективных и трудно проверяемых свойств требований.

Возвращаясь к иерархии требований, нужно отметить, что наиболее бесспорными требованиями следует считать бизнес-требования. Данные требования формулируют первые лица, представляющие Заказчика, и вряд ли кто-нибудь лучше них сможет сказать, каким условиям должно соответствовать создаваемое приложение, чтобы соответствовать бизнес-целям предприятия. Тем не менее, если у представителя исполнителя возникают сомнения в необходимости того или иного бизнес-требования, вызванные интуитивными соображениями либо опытом внедрения приложения на аналогичных предприятиях, он должен проявить инициативу и собрать совместное совещание сторон. Аргументы в пользу отсутствия необходимости требования, несомненно, будут восприняты, особенно если они будут мотивированы в бизнес-терминологии Заказчика и подтверждены выкладками, прогнозирующими соотношение затрат на выполнение требования и ожидаемой от него эффективности.

Необходимость требований пользователя может вытекать из соответствующих бизнес-требований. Кроме того, требования пользователя могут мотивироваться эргономичностью продукта и особенностями функционирования его отдела (подразделения), недостаточно полно раскрытыми на предыдущем уровне иерархии требований.

Большинство функциональных требований вытекают из требований первых двух уровней. Другие функциональные требования могут лежать вне сферы компетенции Заказчика (который, вообще говоря, не обязан быть экспертом в области IT) и их должен сформулировать Исполнитель. Так, например, приложение в процессе его использования может начать снижать свою производительность из-за больших объемов накапливаемых данных. Поэтому целесообразно заложить функции архивирования информации, переключения учетных периодов и т. п., необходимость которых следует не из особенностей бизнеса предприятия внедрения, а из общих принципов построения приложения.

Более слабой, чем «необходимость», формулировкой обладает свойство «полезность при эксплуатации». Разграничение между данными свойствами проводят следующим образом. Необходимыми следует считать свойства, без

выполнения которых невозможно либо затруднено выполнение автоматизированных бизнес-функций пользователей; полезными при эксплуатации следует считать любые свойства, повышающие эргономические качества продукта.

Осуществимость (выполнимость). Является в некоторой степени конкурирующим по введенным выше двум свойствам.

В принципе, никто не мешает сформулировать требование, выполнимость которого ограничивается сегодняшним уровнем развития техники и технологии, хотя многое из того, что было невыполнимо десять лет назад, вполне выполнимо сегодня. Можно сформулировать требование, выполнимость которого ограничена научными представлениями о строении Вселенной, например – требование мгновенной передачи информации с земной станции на Марс.

Выполнимость требования на практике определяется разумным балансом между ценностью (степенью необходимости и полезности) и потребными ресурсами. Так, если стоимость контракта на разработку приложения составляет 10000 д. е., а затраты на выполнение нового требования, возникшего в момент, когда проект выполнен наполовину, оцениваются в 4000 д. е., является ли оно невыполнимым? Скорее всего, да, если Исполнитель докажет Заказчику новизну требования (требование не входило в согласованные спецификации) и сложность его исполнения. Но если требование является критически важным, необходимым, однако выпало из поля зрения при подписании контракта, и Заказчик готов выделить дополнительно финансирование, а Исполнитель – трудовые ресурсы, то требование выполнимо. Таким образом, требование осуществимости в ряде случаев также следует считать субъективным, а критерии его оценки лежат в области договоренностей между Заказчиком и Исполнителем.

Необходимо обеспечить возможность переработки требований, если понадобится, и поддерживать историю изменений для каждого положения. Для этого все они должны быть уникальным образом помечены и обозначены, чтобы можно было ссылаться на них однозначно. Каждое требование должно быть записано в спецификации только единожды. Иначе можно легко получить несогласованность, изменив только одно положение из двух одинаковых. Лучше использовать ссылки на первоначальные утверждения, а не дублировать положения. Выполнение модификации спецификации станет гораздо легче, если составить содержание документа и указатель. Сохранение спецификации в базе данных коммерческого инструмента управления требованиями сделает их пригодными для повторного использования.

Трассируемость. Трассируемость требования определяется возможностью отследить связь между ним и другими артефактами приложения (документами, моделями, текстами программ и пр.). Отдельная трасса представляет собой направленное бинарное отношение, заданное на множестве артефактов приложения, где первый элемент отношения представляет соответствующее требование, а второй – артефакт, зависимый от данного требования. На практике трассировки анализируются при посредстве графовых либо табличных моделей.

Процесс трассировки позволяет, с одной стороны, выявить уже на стадии проектирования системы проектные артефакты, к которым не ведет связь ни от одного из артефактов, описывающих требования, с другой – артефакты, описы-

вающие требования, не связанные с проектными артефактами. В первом из случаев целесообразно убедиться в том, что проектный артефакт действительно имеет право на существование, а не является избыточным. Во втором случае необходимо проанализировать полезность выявленных требований: либо эти требования несут недостаточную полезную нагрузку и могут быть игнорированы, либо имеют место ошибки проектирования: пропущены соответствующие артефакты. Другая цель трассировки – повысить управляемость проектом: при изменении отдельно взятого требования становится понятно, какие из проектных, рабочих и других артефактов подлежат изменению.

Упорядоченность по важности и стабильности. Приоритет требования представляет собой количественную оценку степени значимости (важности) требования. Приоритеты требований обычно назначает представитель Заказчика. Разработчик, отталкиваясь от приоритетности требований, управляет процессом реализации приложения. Стабильность требования характеризует прогнозную оценку неизменности требований во времени.

Наличие количественной метрики. Количественные метрики играют важную роль в верификации и аттестации приложения. В первую очередь это относится к нефункциональным требованиям, которые, как правило, должны иметь под собой количественную основу (запрос должен отрабатываться не более, чем *n* секунд; средняя наработка на отказ должна составлять не менее, чем *m* часов). Функциональные требования также могут расширяться количественными мерами при помощи так называемых аспектов применимости.

Задание

Сформировать таблицу, в которой для каждого функционального требования, представленного в разд. 1, будут указаны объекты приложения, участвующие в реализации требования; исполнители; сущности, в которых представлены исходные данные для реализации требования. Для разрабатываемого приложения сформировать спецификации требований в виде таблиц с указанием входных и выходных данных, предварительных условий, последовательности действий по реализации требования, события, при котором реализуется требование, результата для каждого функционального требования, представленного в разд. 1. Описать, каким образом реализуются свойства полноты, ясности, корректности, согласованности, верифицируемости, необходимости, полезность при эксплуатации, осуществимости, модифицируемости, трассируемости, упорядоченности по важности и стабильности при формировании требований к приложению.

Содержание отчета: титульный лист; тема и цель работы; текст индивидуального задания; описание хода выполнения индивидуального задания.

Контрольные вопросы

1 Перечислить принципы спецификации требований к приложению.
2 Перечислить и охарактеризовать группы моделей спецификаций.

3 Перечислить свойства требований к приложению.

4 Охарактеризовать свойства полноты, ясности, корректности, согласованности, верифицируемости, необходимости, полезность при эксплуатации, осуществимости, модифицируемости, трассируемости, упорядоченности по важности и стабильности программной системы.

3 Испытания программных систем

Цель: составить план тестирования к разрабатываемой программной системе.

Теоретические положения

Если учесть, что программная система – это не только используемые в ее составе программные компоненты, но и аппаратное и организационное обеспечение, то и в результатах ее испытаний должны быть отражены показатели выбранных серверов, рабочих станций, сетевого оборудования (их надежность и производительность), а также эффективность разработанного регламента эксплуатации системы. Все *виды испытаний программных систем* можно разделить на функциональные и нефункциональные тесты.

Функциональное тестирование призвано показать (доказать), что автоматизированные рабочие места предоставляют пользователям ровно ту функциональность, которую они от нее ожидают. Система выполняет свои функции корректно в соответствии со спецификациями требований.

Нефункциональное тестирование подтверждает или опровергает соответствие таких свойств приложения, как производительность, надежность, эргономичность и т. д. заданным на этапе ее проектирования параметрам. Система выполняет свои функции в срок, в должном объеме и с приемлемым качеством, и пользоваться ею удобно.

Виды функционального тестирования.

Компонентное тестирование – испытание отдельных программных компонентов приложения, в ходе которых подтверждается корректность проводимых этими компонентами вычислений.

Интеграционное тестирование – испытания, направленные на выявление проблем взаимодействия отдельных компонентов системы. Если программная архитектура приложения довольно сложная, то в ней выделяются подсистемы, для каждой из которых проводят последовательно компонентное и интеграционное тестирование. В завершении проводят интеграционное тестирование всех выделенных подсистем как компонентов единой системы.

Тестирование прототипа представляет собой испытания приложения на первых этапах ее разработки, когда готовы не все ее функциональные блоки. Отсутствующие компоненты заменяются функциональными заглушками, имитирующими их будущую работу. Приложение на указанном этапе представляет собой прототип целевого программного продукта.

Виды нефункционального тестирования.

Нагрузочное тестирование (load testing) – испытание приложения в условиях прогнозируемой нормальной нагрузки. Под величиной нагрузки понимается количество запросов к системе, которое она должна успевать обрабатывать, не превышая определенное исходными требованиями время отклика.

Стрессовое тестирование (stress testing) – испытание приложения в условиях минимальных аппаратных ресурсах и максимально допустимой нагрузки. Цель стрессового тестирования, как понятно из названия, – проверить работоспособность системы в стрессовых ситуациях.

Объемное тестирование (volume testing) – испытания приложения в условиях максимальных (предельно допустимых) объемов информации в базе данных. Основным объектом тестирования в данном случае является зависимость времени отклика и прочих аспектов производительности системы от объемов контролируемых данных.

Тестирование стабильности (stability testing) – проверка, может ли испытываемое приложение длительное время нормально функционировать в условиях, близких к нормальным условиям (средняя нагрузка, средние объемы данных, рекомендуемые аппаратное ресурсы и т. д.).

Тестирование надежности (reliability testing) – гибрид всех перечисленных ранее видов тестирования, направленный на то, чтобы проверить способность системы возвращаться к нормальному режиму работы после коротких периодов максимальной нагрузки, стрессов, предельных объемов данных и т. д.

Тестирование эргономики решений – испытания пользовательского интерфейса на предмет удобства и безопасности эксплуатации приложения.

Испытания программной системы на этапах подготовки к эксплуатации.

После завершения этапа реализации информационной системы Разработчик, совместно с Заказчиком, может проводить следующие виды испытаний.

Тестирование процесса установки (installation testing) – проверка корректности развертывания программных компонентов системы в различных ее конфигурациях, предусмотренных исходными требованиями.

Тестирование на различных конфигурациях (configuration testing) – проверка работоспособности системы при развертывании отдельных ее компонентов (серверной части, клиентских рабочих мест) в условиях всех возможных (предусмотренных исходными требованиями) вариантах операционных систем и конфигурациях аппаратных и программных ресурсов.

Приемочное тестирование (acceptance testing) – комплексное испытание программной системы, выполняемое представителями Заказчика по специально разработанной Исполнителем программе и методике испытаний (ПМИ). Цель приемочного испытания – показать, что разработанная и развернутая на территории Заказчика приложения делает ровно то, что от нее требуется, и делает это с заданными параметрами производительности. В программу приемочных испытаний, помимо функциональных тестов, могут входить и тестирование процесса установки системы, и тестирование ее работы на различных конфигурациях, а также все виды нефункционального тестирования.

Более подробно теоретические сведения и методики изложены в [1].

Задание

Используя задание на курсовое проектирование, разработать план функционального и нефункционального тестирования разрабатываемой программной системы.

Содержание отчета: титульный лист; тема и цель работы; текст индивидуального задания; описание хода выполнения индивидуального задания.

Контрольные вопросы

1 Перечислить и охарактеризовать основные виды функционального тестирования программной системы.

2 Перечислить и охарактеризовать виды нефункционального тестирования программной системы.

3 Перечислить и охарактеризовать виды испытаний программной системы на этапах подготовки к эксплуатации.

4 Использование систем автоматизации разработки программ

Цель: приобретение практических навыков работы с системами автоматизации разработки программ.

Теоретические положения

Средства автоматизации разработки программ (*CASE-средства*). Средства автоматизации разработки программ – инструментарий для системных аналитиков, разработчиков и программистов, позволяющий автоматизировать процесс проектирования и разработки программного обеспечения. Первоначально под CASE-средствами понимались средства, применяемые на ранних процессах жизненного цикла. В первую очередь – на наиболее трудоемких процессах анализа и проектирования. Международный стандарт [ISO/IEC 14102:2008] определяет CASE-средства более широко – как программное средство, поддерживающее процессы жизненного цикла программного обеспечения. CASE-средства характеризуются наличием мощных средств визуального моделирования.

Особенности средств автоматизации разработки программ:

– поддерживают единственную методологию;

– ориентируются на определенную технологию;

– предназначаются для команд, работающих над единственным проектом (так сложилось исторически);

– используются для разработки программной системы;

– разрабатываются одной компанией. Возможность интеграции инструментов других компаний отсутствует.

Примеры CASE-средств:

– Oracle Designer (компании Oracle (http://www.oracle.com/));

– ERwin (компании Computer Associates International, Inc. (http://www.cai.com/));

– Enterprise Architect (компании Sparx Systems (https://sparxsystems.com/));

– Rational Rose (компании Rational Software Corporation (http://www.rational.com/)).

Интегрированные среды.

Интегрированная среда – совокупность программных инструментов, поддерживающая все процессы жизненного цикла программного обеспечения в рамках определенной технологии. Компонентами интегрированных сред являются:

– инструменты управления процессами;

– инструменты управления проектом;

– инструменты конфигурационного управления;

– инструменты верификации;

– инструменты поддержки разработки документации.

Выделяют три уровня интеграции инструментов в интегрированных средах.

Уровень 1. Интеграция инструментов очень слабая. Как правило, обмен информацией между ними происходит через интерфейсы экспорта и импорта.

Уровень 2. Интеграция инструментов одной компании осуществляется на основе единого репозитория. Интеграция инструментов других компаний с первыми инструментами происходит по образцу уровня 1.

Уровень 3. Интеграция всех инструментов осуществляется с помощью общего репозитория. При этом любой инструмент любой компании может осуществлять взаимодействие через службы взаимодействия с репозиторием.

Тенденции развития современных информационных технологий приводят к постоянному возрастанию сложности приложений, создаваемых в различных областях экономики. Современные крупные проекты программных систем характеризуются, как правило, следующими особенностями:

– сложность описания (достаточно большое количество функций, процессов, элементов данных и сложные взаимосвязи между ними), требующая тщательного моделирования и анализа данных и процессов;

– наличие совокупности тесно взаимодействующих компонентов (подсистем), имеющих свои локальные задачи и цели функционирования (например, традиционных приложений, связанных с обработкой транзакций и решением регламентных задач, и приложений аналитической обработки (поддержки принятия решений), использующих нерегламентированные запросы к данным большого объема);

– отсутствие прямых аналогов, ограничивающее возможность использования каких-либо типовых проектных решений и прикладных систем;

– необходимость интеграции существующих и вновь разрабатываемых приложений;

– функционирование в неоднородной среде на нескольких аппаратных платформах;

– разобщенность и разнородность отдельных групп разработчиков по уровню квалификации и сложившимся традициям использования тех или иных инструментальных средств;

– существенная временная протяженность проекта, обусловленная, с одной стороны, ограниченными возможностями коллектива разработчиков, и, с другой стороны, масштабами организации-заказчика и различной степенью готовности отдельных ее подразделений к внедрению приложения.

Для успешной реализации проекта объект проектирования (приложение) должен быть прежде всего адекватно описан, должны быть построены полные и непротиворечивые функциональные и информационные модели приложения. Накопленный опыт проектирования приложений показывает, что это логически сложная, трудоемкая и длительная по времени работа, требующая высокой квалификации участвующих в ней специалистов. Однако до недавнего времени проектирование приложений выполнялось в основном на интуитивном уровне с применением неформализованных методов, основанных на искусстве, практическом опыте, экспертных оценках и дорогостоящих экспериментальных проверках качества функционирования приложения. Кроме того, в процессе создания и функционирования приложения информационные потребности пользователей могут изменяться или уточняться, что еще более усложняет разработку и сопровождение таких систем.

В 70-х и 80-х гг. XX в. при разработке приложений достаточно широко применялась структурная методология, предоставляющая в распоряжение разработчиков строгие формализованные методы описания приложений и принимаемых технических решений. Она основана на наглядной графической технике: для описания различного рода моделей приложения используются схемы и диаграммы. Наглядность и строгость средств структурного анализа позволяла разработчикам и будущим пользователям системы с самого начала неформально участвовать в ее создании, обсуждать и закреплять понимание основных технических решений. Однако широкое применение этой методологии и следование ее рекомендациям при разработке конкретных приложений встречалось достаточно редко, поскольку при неавтоматизированной (ручной) разработке это практически невозможно. Действительно, вручную очень трудно разработать и графически представить строгие формальные спецификации системы, проверить их на полноту и непротиворечивость и тем более изменить. Если все же удается создать строгую систему проектных документов, то ее переработка при появлении серьезных изменений практически неосуществима. Ручная разработка обычно порождала следующие проблемы:

– неадекватная спецификация требований;
– неспособность обнаруживать ошибки в проектных решениях;
– низкое качество документации, снижающее эксплуатационные качества;
– затяжной цикл и неудовлетворительные результаты тестирования.

С другой стороны, разработчики приложений исторически всегда стояли последними в ряду тех, кто использовал компьютерные технологии для повы-

шения качества, надежности и производительности в собственной работе (феномен «сапожника без сапог»).

Перечисленные факторы способствовали появлению программно-технологических средств специального класса – CASE-средств, реализующих CASE-технологию создания и сопровождения приложений. Термин CASE (Computer Aided Software Engineering) используется в настоящее время в весьма широком смысле. Первоначальное значение термина CASE, ограниченное вопросами автоматизации разработки только лишь программного обеспечения (ПО), в настоящее время приобрело новый смысл, охватывающий процесс разработки сложных приложений в целом. Теперь под термином CASE-средства понимаются программные средства, поддерживающие процессы создания и сопровождения приложений, включая анализ и формулировку требований, проектирование прикладного ПО (приложений) и баз данных, генерацию кода, тестирование, документирование, обеспечение качества, конфигурационное управление и управление проектом, а также другие процессы. CASE-средства вместе с системным ПО и техническими средствами образуют полную среду разработки приложений.

Появлению CASE-технологии и CASE-средств предшествовали исследования в области методологии программирования. Программирование обрело черты системного подхода с разработкой и внедрением языков высокого уровня, методов структурного и модульного программирования, языков проектирования и средств их поддержки, формальных и неформальных языков описаний системных требований и спецификаций и т. д. Кроме того, появлению CASE-технологии способствовали и такие факторы, как:

– подготовка аналитиков и программистов, восприимчивых к концепциям модульного и структурного программирования;

– широкое внедрение и постоянный рост производительности компьютеров, позволившие использовать эффективные графические средства и автоматизировать большинство этапов проектирования;

– внедрение сетевой технологии, предоставившей возможность объединения усилий отдельных исполнителей в единый процесс проектирования путем использования разделяемой базы данных, содержащей необходимую информацию о проекте.

CASE-технология представляет собой методологию проектирования приложений, а также набор инструментальных средств, позволяющих в наглядной форме моделировать предметную область, анализировать эту модель на всех этапах разработки и сопровождения приложений и разрабатывать приложения в соответствии с информационными потребностями пользователей. Большинство существующих CASE-средств основано на методологиях структурного (в основном) или объектно-ориентированного анализа и проектирования, использующих спецификации в виде диаграмм или текстов для описания внешних требований, связей между моделями системы, динамики поведения системы и архитектуры программных средств.

Согласно обзору передовых технологий (Survey of Advanced Technology), составленному фирмой Systems Development Inc. в 1996 г. по результатам анке-

тирования более 1000 американских фирм, CASE-технология в настоящее время попала в разряд наиболее стабильных информационных технологий (ее использовала половина всех опрошенных пользователей более чем в трети своих проектов, из них 85 % завершились успешно). Однако, несмотря на все потенциальные возможности CASE-средств, существует множество примеров их неудачного внедрения, в результате которых CASE-средства становятся «полочным» ПО (shelfware). В связи с этим необходимо отметить следующее:

– CASE-средства необязательно дают немедленный эффект; он может быть получен только спустя какое-то время;

– реальные затраты на внедрение CASE-средств обычно намного превышают затраты на их приобретение;

– CASE-средства обеспечивают возможности для получения существенной выгоды только после успешного завершения процесса их внедрения.

Ввиду разнообразной природы CASE-средств было бы ошибочно делать какие-либо безоговорочные утверждения относительно реального удовлетворения тех или иных ожиданий от их внедрения. Можно перечислить следующие факторы, которые усложняют определение возможного эффекта от использования CASE-средств:

– широкое разнообразие качества и возможностей CASE-средств;

– относительно небольшое время использования CASE-средств в различных организациях и недостаток опыта их применения;

– широкое разнообразие в практике внедрения различных организаций;

– отсутствие детальных метрик и данных для уже выполненных и текущих проектов;

– широкий диапазон предметных областей проектов;

– различная степень интеграции CASE-средств в различных проектах.

Вследствие этих сложностей доступная информация о реальных внедрениях крайне ограничена и противоречива. Она зависит от типа средств, характеристик проектов, уровня сопровождения и опыта пользователей. Некоторые аналитики полагают, что реальная выгода от использования некоторых типов CASE-средств может быть получена только после одно- или двухлетнего опыта. Другие полагают, что воздействие может реально проявиться в фазе эксплуатации жизненного цикла приложения, когда технологические улучшения могут привести к снижению эксплуатационных затрат.

Для успешного внедрения CASE-средств организация должна обладать следующими качествами:

– *технология*. Понимание ограниченности существующих возможностей и способность принять новую технологию;

– *культура*. Готовность к внедрению новых процессов и взаимоотношений между разработчиками и пользователями;

– *управление*. Четкое руководство и организованность по отношению к наиболее важным этапам и процессам внедрения.

Если организация не обладает хотя бы одним из перечисленных качеств, то внедрение CASE-средств может закончиться неудачей независимо от степени тщательности следования различным рекомендациям по внедрению.

Для того чтобы принять взвешенное решение относительно инвестиций в CASE-технологию, пользователи вынуждены производить оценку отдельных CASE-средств, опираясь на неполные и противоречивые данные. Эта проблема зачастую усугубляется недостаточным знанием всех возможных «подводных камней» использования CASE-средств. Среди наиболее важных проблем выделяются следующие:

– достоверная оценка отдачи от инвестиций в CASE-средства затруднительна ввиду отсутствия приемлемых метрик и данных по проектам и процессам разработки ПО;

– внедрение CASE-средств может представлять собой достаточно длительный процесс и может не принести немедленной отдачи. Возможно даже краткосрочное снижение продуктивности в результате усилий, затрачиваемых на внедрение. Вследствие этого руководство организации-пользователя может утратить интерес к CASE-средствам и прекратить поддержку их внедрения;

– отсутствие полного соответствия между теми процессами и методами, которые поддерживаются CASE-средствами, и теми, которые используются в данной организации, может привести к дополнительным трудностям;

– CASE-средства зачастую трудно использовать в комплексе с другими подобными средствами. Это объясняется как различными парадигмами, поддерживаемыми различными средствами, так и проблемами передачи данных и управления от одного средства к другому;

– некоторые CASE-средства требуют слишком много усилий для того, чтобы оправдать их использование в небольшом проекте, при этом тем не менее можно извлечь выгоду из той дисциплины, к которой обязывает их применение;

– негативное отношение персонала к внедрению новой CASE-технологии может быть главной причиной провала проекта.

Пользователи CASE-средств должны быть готовы к необходимости долгосрочных затрат на эксплуатацию, частому появлению новых версий и возможному быстрому моральному старению средств, а также постоянным затратам на обучение и повышение квалификации персонала.

Несмотря на все высказанные предостережения и некоторый пессимизм, грамотный и разумный подход к использованию CASE-средств может преодолеть все перечисленные трудности. Успешное внедрение CASE-средств должно обеспечить такие выгоды, как:

– высокий уровень технологической поддержки процессов разработки и сопровождения ПО;

– положительное воздействие на некоторые или все из перечисленных факторов: производительность, качество продукции, соблюдение стандартов, документирование;

– приемлемый уровень отдачи от инвестиций в CASE-средства.

Задание

Для разрабатываемого приложения выбрать средства автоматизации разработки программ. Обосновать выбор CASE-средств, описать порядок их применения при реализации разрабатываемого приложения. Используя выбранные средства, сгенерировать автоматически программный код основных компонентов разрабатываемого приложения на основании диаграмм классов, представленных в разд. 1.

Содержание отчета: титульный лист; тема и цель работы; текст индивидуального задания; описание хода выполнения индивидуального задания.

Контрольные вопросы

1 Перечислить основные сложности современных крупных проектов по созданию программных систем.

2 Перечислить основные проблемы ручной разработки приложений.

3 Какие факторы способствовали появлению CASE-технологии разработки программного обеспечения?

4 Для чего предназначены CASE-средства разработки программ?

5 Какими качествами должна обладать организация, чтобы в ней можно было эффективно использовать CASE-средства?

5 Перечислить и охарактеризовать уровни интеграции инструментов в интегрированных средах.

6 Что такое CASE-технология?

5 Компонентное программирование

Цель: разделить разрабатываемую программную систему на компоненты.

Теоретические положения

Компонентно-ориентированное программирование (КОП). Эта парадигма программирования направлена прежде всего на повышение надежности коммерческих бизнес-систем. Суть компонентно-ориентированного программирования (КОП) сводится к возможности контролировать взаимодействие проектируемых и выполняемых модулей на предмет согласованности информационных структур. Идея является относительно новой. Частично идеи КОП воплощены в такие языки, как Java, Ada, C#.

Отличительные черты КОП. Несмотря на свою относительную молодость КОП имеет свои особенности, которые регулируют не только особенности языка, но и всей экосистемы КОП. К таким отличительным чертам относят:

– четко выраженную ориентированность на модули. Модуль является основной структурной единицей;

– раздельную компиляцию модулей. Это приводит к сбережению вычислительных и временных ресурсов;

– строгую типизацию как внутри модуля, так и между модулями. Обеспечивает надежную работу компонентов в целом;

– неизбежность динамической сборки мусора. Для компилируемых языков это важный и необычный механизм;

– строгое разделение частей модулей, предназначенных для взаимодействия с другими модулями, и скрытые части только для работы внутри модуля.

Компонентное ПО – это хорошая идея, но без компонентов она не будет работать. Компоненты сначала нужно создать, а это требует подходящих средств разработки. И даже если компоненты уже есть, должны быть инструменты для их сборки. Некоторые инструменты могут предназначаться для сборки компонентов, другие – для их конструирования, а некоторые подходят для того и другого. Небольшое число продуктов хорошо подходит для разработки и развития целых компонентных каркасов.

В основном большинство языков сегодня выглядят похоже на разновидность C или Pascal, например, C++, C# и Java подобны C, в то время как Ada, Component Pascal и даже Visual Basic и Eiffel подобны Pascal. Lisp, Smalltalk и некоторые другие «экзотические» языки программирования составляют сравнительно небольшое меньшинство.

Это означает, что есть достаточная свобода в выборе языка для решения конкретной задачи.

На практике языково-независимые объектные модели значительно увеличили эту свободу в последнее время. Большинство языков сегодня дают доступ к языково-независимым объектным моделям, например, COM. Объектные модели добавляют динамические возможности, которых может недоставать языку. Языково-независимые объектные модели делают языковые решения менее стратегическими, чем они обычно были: выбор одного языка для одного компонента не мешает использовать второй язык для другого компонента. Использование конкретного языка больше не создает острова. Следовательно, больше нет причин не использовать лучший язык для данной задачи.

Создает ли язык различия вообще, если кодирование является только малой частью общей стоимости проекта? Фактически мелкие синтаксические нюансы, например, как выглядят конструкции циклов, не дают никаких заметных отличий в стоимости проекта (предполагая, что запретный GOTO сейчас более или менее «умер»). Следовательно, сравнение языков касательно «программирования в малом» не имеет большого значения. Но современный язык программирования – это нечто большее, чем только нотация для реализации мелких алгоритмов. Хорошо спроектированный язык программирования также поддерживает программирование в большом. Чтобы оставаться управляемыми, большие программы должны разбиваться на компоненты, которые взаимодействуют только через определенные интерфейсы. Хороший язык программирования может использоваться не только как язык реализации, но и как язык описания и спецификации интерфейса. Интерфейсы определяют архитектуру системы: те части, которым разработчик может доверять; статические свойства

системы, которые не могут быть нарушены; структурную суть компонентов. Язык, который позволяет выражать явно («статически») большую часть архитектуры системы в своей нотации для интерфейсов, делает возможным написание средств, которые помогут обеспечить согласованность реализации и спецификации. Компилятор может дать сигнал о нарушениях интерфейса еще во время компиляции, когда исправление ошибки обходится недорого. Проверки во время выполнения дают возможность выявить другие нарушения интерфейсов как можно раньше, во время тестирования.

Статическая выразительность языка и инструментальная поддержка, задействованная им, становятся еще более важными, когда интерфейсы изменяются, что часто случается на этапе проектирования и создания прототипа, или позже, на этапе эксплуатации ПО (который отнимает около 80 % от общих затрат разработки). Фактически архитектура большой системы неизменно ухудшается со временем, когда выполняются изменения и расширения. Улучшение архитектуры системы, при котором отбрасывается старый багаж и придается гибкость некоторому подмножеству интерфейсов и компонентов, называется рефакторингом. Сегодня этой стороной разработки пренебрегают более всего. Но компонентно-ориентированные языки являются мощными инструментами рефакторинга, которые уменьшают время, стоимость и риск, связанные с изменением частей существующей системы.

Это значит, что состоятельный язык программирования может привнести отличия в большинство фаз жизненного цикла программных компонентов и в жизненный цикл компонентной программной системы (который может быть намного длиннее жизненного цикла любого из ее компонентов). Следовательно, распространенное мнение, что выбор языка программирования не имеет значения, основан на слишком плоской аргументации, которая опускает измерение «программирование в большом».

Современные языки программирования поддерживают явно формулируемые интерфейсные конструкции. Их главное преимущество в том, что они могут быть применены к очень большому числу очень больших задач и они эффективны. Языки, которые избегают статических конструкций, чтобы получать предельную гибкость с наименьшими усилиями, называются «динамическими» или «четвертого поколения» языками. Динамические языки поддерживают инкрементную загрузку кода, сборку мусора и тесно связаны с поддержкой среды разработки. Их превосходство – в быстрой разработке маленьких частей низкотехнологичного кода, например, сценариев для сборки компонентов. Их основное достоинство – что «все проходит», т. е. в них нет жесткой системы типов, лишних секций объявлений или других подобных ограничивающих статических конструкций. Даже с самыми агрессивными технологиями оптимизации они обычно медленнее, чем статические языки. Тесная интеграция со средой означает, что их использование удобно и что объектная модель может быть вставлена непосредственно в язык, так что взаимодействие между компонентами становится намного более легким, чем при использовании языковонезависимой объектной модели.

Относительно компонентного ПО интересно отметить, что различия между статическими и динамическими языками – это причина того, что OLE и OpenDoc пришли к двум уровням программируемости: на уровне объектной модели (статический) и на уровне автоматизации (динамический).

Более современные языки, такие как C# и Java, доказали, что различия между статическими и динамическими языками могут быть преодолены, т. е. что язык может собрать в себе большинство преимуществ обоих сторон. Такой гибридный язык позволяет гибкую разработку или модификацию реализаций компонентов. С другой стороны, он позволяет жестко определять интерфейсы, так что соответствие этим интерфейсам может быть проверено автоматически. Мы называем такие языки, подобные C# и Java, компонентно-ориентированными. Пока что мы не говорили об объектной ориентированности, которая появилась и в тех, и в других языках. Давайте взглянем на ООП и на то, как компонентная ориентированность перерастает ООП.

Нет всеобщего согласия относительно того, что такое ООП и чем оно должно быть, но большинство потребует от ООП-языка следующих характеристик: объекты, классы, полиморфизм, позднее связывание и наследование. Объект инкапсулирует состояние и поведение. Поведение объекта доступно через процедуры, связанные с типом объекта, так называемые методы. Класс является планом, чертежом для реализации объектов данного типа. Во время выполнения может быть создано произвольное число экземпляров класса, т. е. объектов.

Полиморфизм означает, что достаточно похожие объекты могут подменять друг друга, т. е. во время выполнения переменной могут быть присвоены объекты различных типов. Объекты являются «достаточно похожими», если они реализуют одинаковый интерфейс, т. е. следуют одному и тому же контракту. Например, механизм хранилищ должен принимать любой объект, который является хранимым, т. е. который реализует интерфейс хранилища.

Позднее связывание значит, что поведение объекта может быть различным в зависимости от того, какой динамический тип он имеет. Название «позднее связывание» происходит из факта, что из-за полиморфизма во время компиляции неизвестно, будет ли сохраняемый объект треугольником, текстом или чем-то еще. Следовательно, решение о том, какого типа объект сохраняется, должно приниматься позже, а именно – во время выполнения.

Сокрытие информации означает, что интерфейс и реализация объекта различаются между собой. Внешние взаимодействия происходят только через интерфейс, а реализация остается скрытой. Это позволяет позднее изменить скрытые детали реализации, без нарушений в работе клиентов.

Полиморфизм, позднее связывание и сокрытие информации работают совместно, чтобы сделать возможным четкое разделение интерфейса и реализации и, следовательно, обеспечить поддержку для компонентного ПО. Однако наследование реализации дает нечто другое. Это значит, что объект может «наследовать» некоторое поведение от другого объекта, «перегрузить» часть его и добавить к нему свое собственное новое поведение. Это удобная форма повторного использования кода. Она хорошо работает, если наследующий объект жестко придерживается контракта того объекта, от которого он наследует,

т. к. тогда он может использоваться вместо него без нарушения контракта с клиентами.

К сожалению, если используется наследование, очень тяжело предупредить повторный вход в объект, т. е. само-рекурсия может вести к непредсказуемым изменениям состояния унаследованного объекта. Например, поток управления в пределах объекта может перескакивать вверх и вниз между базовым классом и подклассом. Если некоторая деталь реализации базового класса изменяется в новой версии, подкласс может перестать работать, поскольку его предположения больше неверны.

Сокрытие информации делает возможным задать контракт между подклассом и базовым классом однозначно (так называемый интерфейс специализации). Наследование реализации – это такая тесная связь между объектами, которая на практике требует, чтобы исходный код наследуемого объекта был открыт, т. е. сокрытие информации отбрасывается и реализация должна рассматриваться как интерфейс («наследование разрушает инкапсуляцию»). Мы ранее уже встречались с этой проблемой под названием «семантическая хрупкость базового класса». В будущем кто-нибудь предложит практические правила для ограниченного вида наследования реализации, который не ведет к проблеме семантической хрупкости.

Хорошим высказыванием является то, что наследование реализации – это GOTO девяностых. Подобно GOTO шестидесятых, наследование очень удобно, программисты его используют, не всегда очевидно, как без него можно обойтись, решение без него может сделать программу длиннее, и сам вопрос может вызвать дискуссию. Но с фундаментальной точки зрения наследование имеет сходство с GOTO в том, что приводит к неконтролируемым передачам управления, которые затрудняют понимание программы и делают рискованным ее применение.

Наследование вредно, если оно используется через границы компонентов. Пока наследование используется внутри компонента, оно не опасно. Так как компонент является черным ящиком, он может быть реализован с использованием наследования реализации, функционального программирования, ассемблера и чего угодно, лишь бы это годилось для конкретного компонента. Единственное, что имеет значение, – это верная реализация интерфейса, т. е. соблюдение контракта с окружающим миром. Внутри компонента разработчик имеет полный контроль над всеми своими исходными кодами и может свободно менять внутренние интерфейсы, как ему будет угодно.

Компонентно-ориентированные языки помогают создавать более надежное компонентные программные системы быстрее, т. к. такие языки предоставляют «компонентно-ориентированные» возможности в дополнение к ООП-возможностям (полиморфизм, позднее связывание и сокрытие информации).

Безопасность – одна из таких возможностей. Безопасность означает, что язык гарантирует некоторые базовые правила для компонента, которые не приходится помещать заново в контракт каждого компонента. В частности, безопасный язык программирования гарантирует целостность памяти, т. е. один компонент не может разрушить память других компонентов. Это упрощает до-

говорные обязательства каждого объекта, т. к. правильное управление памятью – а его отсутствие является причиной более половины всех ошибок программирования – может просто считаться само собой разумеющимся. Это достигается предоставлением службы сборки мусора, т. е. память возвращается автоматически, когда она больше не используется. Сборка мусора невидима и освобождает программиста от ручной работы.

Безопасные языки дают тот тип защиты, который желателен в программной среде, состоящей и тесно взаимодействующих компонентов, особенно если учитывать, что в этом случае традиционные механизмы аппаратной защиты применять нельзя.

В будущем все больше и больше заказчиков будут требовать использования безопасных языков для создания компонентов, поскольку это может значительно уменьшить количество загадочных сбоев и, следовательно, недоверие к компоненту. Как только компоненты станут широко распространенными, вопросы качества обязательно окажутся на главном месте.

Компонентно-ориентированный язык также помогает достичь безопасности на более высоком уровне, нежели просто целостность памяти. Сокрытие информации – часть ответа, поскольку оно позволяет прятать, а значит защищать детали реализации. Большинство ООП-языков ограничивают сокрытие информации отдельными классами. Это очень ограничено, т. к. обычно несколько классов могут кооперироваться для предоставления некоторой услуги. Такие классы должны иметь возможность тесно работать вместе, при этом их кооперация должна быть защищена от внешних воздействий. Это значит, что такие классы должны иметь их собственный закрытый интерфейс, который не будет делиться больше ни с кем. Чтобы гарантировать согласованность закрытых контрактов такого рода, язык программирования должен поддерживать сокрытие информации вокруг нескольких классов. Чтобы сделать это, язык должен предоставлять конструкцию «модуль» или «пакет», подобно языкам C# или Java.

Сокрытие информации над отдельными классами является необходимым требованием для компонентно-ориентированного языка, поскольку это позволяет программному архитектору создавать заказные безопасные свойства (т. е. инварианты) в компонентной программной системе.

Компонентно-ориентированный язык подразумевает, что реализация предоставляет объектную модель, которая поддерживает динамическую загрузку новых компонентов. Обычно это библиотечная служба, которая позволяет явно загружать компонент по его имени или иному подходящему идентификатору. Такое средство называется поддержкой метапрограммирования, которая позволяет одной программе манипулировать (в данном случае загружать) другой программой. Это требует обширной информации о типах во время выполнения (RTTI), которая идет гораздо дальше минимальной информации, поддерживаемой ООП-языками, такими как C++.

Задание

Разбить разрабатываемое приложение на компоненты. Реализовать и протестировать компоненты разрабатываемого приложения. Представить результаты тестирования компонентов бизнес-логики и пользовательского интерфейса.

Содержание отчета: титульный лист; тема и цель работы; текст индивидуального задания; описание хода выполнения индивидуального задания.

Контрольные вопросы

1 Что такое компонентно-ориентированное программирование?

2 Перечислить отличительные особенности, преимущества и недостатки компонентно-ориентированного программирования.

3 Охарактеризовать преимущества и недостатки применения полиморфизма в компонентно-ориентированных языках программирования.

4 Охарактеризовать преимущества и недостатки применения наследования в компонентно-ориентированных языках программирования.

5 Охарактеризовать преимущества и недостатки применения позднего связывания в компонентно-ориентированных языках программирования.

6 В чем состоит свойство безопасности компонентно-ориентированных языков программирования?

Список литературы

1 **Гагарина, Л. Г.** Технология разработки программного обеспечения : учеб. пособие / Л. Г. Гагарина, Е. В. Кокорева, Б. Д. Сидорова-Виснадул ; под ред. Л. Г. Гагариной. – М. : ФОРУМ ; ИНФРА-М, 2025. – 400 с.

2 **Макконнелл, С.** Совершенный код. Мастер-класс = Code Complete. Second Edition : пер. с англ. / С. Макконнелл. – СПб. : БХВ, 2020. – 896 с. : ил.

3 **Фримен, Э.** Паттерны проектирования = Head First Design Patterns / Э. Фримен ; пер. с англ. Е. Матвеева. – СПб. : Питер, 2016. – 656 с. : ил.

4 **Dennis, A.** System Analysis & Design. An Object-Oriented Approach with UML = Системный анализ и проектирование на универсальном языке моделирования / A. Dennis, B. Wixom, D. Tegarden. – 5th ed. – New York : John Wiley & Sons, 2015. – 276 p.

5 **Макаровских, Т. А.** Документирование программного обеспечения. В помощь техническому писателю : учеб. пособие / Т. А. Макаровских. – 2-е изд. – М. : ЛЕНАНД, 2015. – 266 с.

6 **Арлоу, Д.** UML 2 и унифицированный процесс. Практический объектно-ориентированный анализ и проектирование / Д. Арлоу. – М. : Символ-Плюс, 2015. – 624 с.

7 **Орлов, С. А.** Программная инженерия / С. А. Орлов. – СПб. : Питер, 2016. – 640 с.

8 **Буч, Г.** Введение в UML от создателей языка : практ. рук. / Г. Буч, Дж. Рамбо, И. Якобсон ; пер. с англ. Н. Мухина. – 3-е изд. – М. : ДМК Пресс, 2023. – 495 с.

9 **Белладжио, Д.** Разработка программного обеспечения: управление изменениями : практ. рук. / Д. Белладжио, Т. Миллиган ; пер. с англ. Н. А. Мухина. – 2-е изд. – М. : ДМК Пресс, 2023. – 385 с.

10 **Гэртнер, М.** ATDD – разработка программного обеспечения через приемочные тесты : практ. рук. / М. Гэртнер ; пер. с англ. А. А. Слинкина. – 2-е изд. – М. : ДМК Пресс, 2023. – 233 с.

11 **Чернышев, С. А.** Принципы, паттерны и методологии разработки программного обеспечения : учебник для вузов / С. А. Чернышев. – М. : Юрайт, 2025. – 176 с.