МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования «Национальный исследовательский Нижегородский
государственный университет им. Н.И. Лобачевского»
Институт информационных технологий, математики и механики

П.Ю. Белокрылов

ПРАКТИЧЕСКИЙ АСПЕКТ ПРОГРАММИРОВАНИЯ СИСТЕМ ИНЖЕНЕРНОГО АНАЛИЗА

Учебно-методическое пособие

Рекомендовано методической комиссией ИИТММ
для студентов ННГУ, обучающихся по направлению подготовки
09.03.03 «Прикладная информатика»

Нижний Новгород
2017

УДК 004.413, 004.414, 004.415, 004.416
ББК 32.973
Б 43

Б43 П.Ю. Белокрылов ПРАКТИЧЕСКИЙ АСПЕКТ ПРОГРАММИРОВАНИЯ СИСТЕМ ИНЖЕНЕРНОГО АНАЛИЗА. Учебно-методическое пособие. – Нижний Новгород: Нижегородский госуниверситет, 2017. – 83 с.

Рецензент: д.ф.-м.н., профессор Д.Т. Чекмарев

В учебно-методическом пособии излагаются основные этапы разработки прикладного программного обеспечения — систем инженерного анализа. Специфика прикладной области данного программного обеспечения определяет особенности основных процессов его жизненного цикла. Излагаются рекомендации по организации процессов разработки такого класса систем, учитывающие особенности его жизненного цикла в общем контексте программной инженерии.

Пособие предназначено для бакалавров направления подготовки «Прикладная информатика».

УДК 004.413, 004.414, 004.415, 004.416
ББК 32.973

Оглавление

Введение

На сегодняшний день в индустрии разработки программного обеспечения (ПО) отсутствуют универсальные и общепринятые методологии, которые детально регламентировали бы все процессы разработки ПО. Также отсутствуют готовые стандартные компоненты для реализации функциональности разрабатываемого ПО. Все это является постоянным источником различного рода проблем при разработке и эксплуатации программ и самая главная из них — *преодоление сложности — одной из самых важных характеристик большинства программных систем*.

Лишь немногие программные системы настолько просты, что спецификацию для них может разработать один человек, реализовать их по данной спецификации и использовать для задуманной цели, а также сопровождать, внося изменения при появлении новых требований или обнаружении ошибок. Подобные простые системы достаточно ограничены по своему применению и живут относительно недолго. Инвестиции (время/деньги) в такое ПО невелики и их нетрудно при необходимости переписать заново. Но большинство программ имеют совсем другие характеристики. Это сложные системы, которые не под силу реализовать одному человеку. В их создании участвует целый коллектив и разработчики должны координировать свою деятельность. Так как стоимость разработки таких программ велико, то подобные системы должны быть *сопровождаемыми*, а их программный код — *изменяемым*. Изменения, внесенные в одном месте программы, нередко влияют на другие ее части, то есть влекут за собой очередные изменения. Если такие зависимости не отмечены (а иногда просто пропущены), то система будет работать некорректно. Ситуацию еще больше усугубилась после появления языков высокого уровня, что позволило создавать программы непрофессиональным программистам.

Дополнительно следует отметить, что для крупных программных проектов известной проблемой является взаимопонимание между заказчиками и разработчиками. Это касается как выработки спецификаций (в которых могут участвовать специалисты различных областей), так и оценки соответствия данным спецификациям (в данном процессе нередко участвует несколько человек). Несогласованность и пробелы в спецификациях, определяющих, что должна делать система (и оценках, насколько хорошо она это делает), добавляют проблемы в процесс разработки ПО.

Предлагаемые в данной работе рекомендации разработчикам — это, прежде всего подходы, позволяющие контролировать проблемы разработки ПО. Рассматриваемые решения направлены на сокращение издержек основных процессов жизненного цикла программного продукта: выработка требований (первичных и специфицирующих последующие изменения), разработка архитектуры, конструирование, сопровождение (расширение существующего функционала).

Следует отметить, что все эти рекомендации предназначены в основном для систем целевого назначения, а именно — систем инженерного анализа. Данное ПО характеризуется относительно простой архитектурой и достаточно ясными требованиями к функционированию системы, что в некоторой степени переносит основную сложность процесса создания данного ПО на этап конструирования.

1. Основные этапы разработки ПО

Разработка ПО — сложный процесс, который может включать множество этапов:

- анализ проблемной области;
- выработка первичных требований;
- создание плана конструирования;
- разработка архитектуры ПО, или высокоуровневое проектирование;
- детальное проектирование;
- кодирование и отладка;
- блочное тестирование;
- интеграционное тестирование;
- интеграция компонент системы;
- тестирование системы в целом;
- корректирующее сопровождение.

Процессы «детальное проектирование» и «кодирование и отладка» объединяют общим понятием «конструирование», которое и будет использоваться в дальнейшем. Среди множества компонент разработки ПО будут рассмотрены только «разработка архитектуры ПО, или высокоуровневое проектирование» и конструирование, так как именно на эти этапы проецируется проблема сложности рассматриваемого целевого ПО.

2. Разработка архитектуры

2.1 Определение

Архитектура — это высокоуровневая часть проекта приложения, каркас, состоящий из деталей проекта [1, 2, 3]. Архитектуру также называют «архитектурой системы», «высокоуровневым проектом» и «проектом высокого уровня». Разработка архитектуры всегда предшествует этапу конструирования, так как качество архитектуры определяет концептуальную целостность системы, которая в свою очередь определяет итоговое качество системы. Продуманная архитектура предоставляет такую структурную модель системы, которая нужна ей для поддержания концептуальной целостности. Для этапа конструирования архитектура естественным образом укажет те части системы, над которыми разработчик может трудиться независимо, при этом, не потеряв из вида все взаимосвязи между ними.

Внесение изменений в архитектуру на этапе конструирования обходится весьма недешево. Время, необходимое для исправления ошибки в архитектуре ПО, существенно превышает временные затраты на исправление ошибки в коде [4, 5], а в некоторых случаях может повлечь повторный процесс разработки (ремануфактуринг).

2.2 Компоненты архитектуры

Приведем компоненты архитектуры, на которые следует обратить внимание.

2.2.1 Организация программы

В первую очередь архитектура должна включать общее описание системы, то есть определять ее основные компоненты. В зависимости от размера программы ее компонентами могут быть отдельные классы (модули) или подсистемы, состоящие из нескольких классов. Каждый компонент является классом или набором классов/методов, которые в совокупности реализуют высокоуровневые функции программы, такие как взаимодействие с пользователем, реализации алгоритмов или доступ к данным.

Архитектура должна четко определять ответственность каждого компонента. Компонент должен иметь одну область ответственности и как можно меньше знать об областях ответственности других компонентов. Сведя к минимуму объем сведений, известных одним компонентом о других, можно локализовать информацию о проекте в отдельных частях связанной структуры.

С другой стороны, архитектура должна ясно определять правила коммуникации для каждого компонента. Она должна описывать, как данный компонент может непосредственно использовать другие, а какие вообще не должен использовать.

2.2.2 Основные классы/модули

Архитектура должна определять основные классы/модули приложения, их области ответственности и механизмы взаимодействия друг с другом. Она должна описывать иерархии классов/модулей, а также изменения состояний и время существования объектов. Если система достаточно велика, архитектура должна описывать организацию классов/модулей в подсистемы.

2.2.3 Организация данных

Архитектура должна описывать основные виды формата файлов, таблиц и должна определять высокоуровневую организацию и содержание всех используемых баз данных. Прямой доступ к данным обычно следует предоставлять только одной подсистеме или классу; за исключением того случая, когда используются классы или методы доступа, обеспечивающие доступ к данным, контролируемым абстрактным образом.

2.2.4 Особенности работы расчетных алгоритмов

Архитектура, зависимая от специфических расчетных алгоритмов, должна описывать их влияние на проект системы. Например, длительное время работы расчетного алгоритма приводит к требованию размещения его в отдельном рабочем потоке, который может определенным образом взаимодействовать с другими потоками в системе, в частности, интерфейсным. Все эти особенности должны найти свое отражение в спецификации архитектуры.

2.2.5 Пользовательский интерфейс

Пользовательский интерфейс (UI) может проектироваться на этапе разработки архитектуры. Архитектура должна описывать главные элементы UI (например, для GUI — окна, элементы управления, форматы Web-страниц) или интерфейс командной строки. Удобство интерфейса в конечном итоге может определить популярность или провал программы. Архитектура должна быть модульной, чтобы UI можно было изменить, не затронув расчетных алгоритмов и модулей

программы, отвечающих за вывод данных. Например, архитектура должна обеспечивать возможность сравнительно легкой замены группы классов интерфейса командной строки на группу классов интерактивного (оконного) интерфейса.

2.2.6 Управление ресурсами

Архитектура должна включать план управления ограниченными ресурсами, такими как соединения с базой данных, потоки и дескрипторы. При разработке приложений, которые будут работать в условиях ограниченной памяти (оперативной и дисковой), архитектура должна определять способ управления данным ограничением (менеджеры памяти). Архитектура должна включать оценку ресурсов, используемых в номинальном режиме и при экстремальной нагрузке. В простейшем случае эти оценки должны подтвердить, что предполагаемая среда использования приложения (целевая вычислительная система пользователя) будет располагать нужными ресурсами.

2.2.7 Безопасность

Архитектура должна определять подход к безопасности на уровне проекта приложения и на уровне кода. О безопасности нужно помнить и при разработке принципов кодирования, в том числе методик обработки буферов и ненадежных данных (данных, вводимых пользователями, файлов «cookie», конфигурационных данных и данных других внешних интерфейсов), подходов к шифрованию, уровню подробности сообщений об ошибках, защите секретных данных, находящихся в памяти.

2.2.8 Масштабируемость

Масштабируемостью называют возможность системы адаптироваться к росту требований. Хорошее архитектурное решение предусматривает возможность модификации системы «по горизонтали»: рост числа пользователей, серверов, сетевых узлов, записей в базу данных, транзакций и так далее.

2.2.9 Интернационализация/локализация

«Интернационализацией» называют реализацию в программе поддержки региональных стандартов. «Локализацией» называют перевод интерфейса программы и реализацию в ней поддержки конкретного языка. Вопросы интернационализации заслуживают особого внимания при разработке архитектуры интерактивной системы, так как такая система включает множество подсказок, индикаторов состояния, вспомогательных сообщений, сообщений об ошибках. Прежде всего, следует оценить объем ресурсов, используемых для поддержки интерактивности и рассмотреть типичные вопросы, связанные со строками и наборами символов, такие как выбор набора символов (ASCII, DBCS, EBCDIC, MBCS, Unicode, ISO 8859 и т.д.) и тип строк (строки **C**, строки **Visual Basic** и так далее), а также предусмотреть такие способы изменения строк, которые не требовали бы изменения кода и оказывали минимальное влияние на расчетный код и пользовательский интерфейс. Предпочтительным решением в этом случае является встраивание строк непосредственно в код программы, инкапсуляция их в специальный класс/модуль, и использование их в дальнейшем только через четко регламентируемые методы/функции или же сохранить эти строки в отдельном

файле ресурсов. Такое решение должно быть принято на этапе проектирования архитектуры системы.

2.2.10 Ввод-вывод

Решение об организации ввода-вывода принимается на этапе проектирования архитектуры. Именно на этом этапе определяется схема чтения данных: упреждающее чтение, чтение с задержкой или по требованию. Кроме того, архитектура четко определяет уровень, на котором будут определяться ошибки ввода-вывода: на уровне компонентов UI (полей ввода) или при реализации конкретного техническое представления (записи БД, формирование потоков данных или файлов).

2.2.11 Обработка ошибок

Обработка ошибок — одна из самых сложных проблем современной информатики. Так по некоторым оценкам [6], код на 90% состоит из блоков обработки исключительных ситуаций, в том числе и ошибок, из чего следует, что только 10% кода отвечают за номинальный режим работы программы. Так как на обработку исключительных ситуаций приходится такая большая часть кода, стратегия их согласованной обработки должна быть выражена в архитектуре. Тем не менее, обработку ошибок часто рассматривают на уровне конвенции кодирования, однако она оказывает влияние на всю систему, поэтому лучше всего рассматривать ее на уровне архитектуры. Приведем некоторые аспекты обработки ошибок, которые должны найти свое отражение на уровне разработки архитектуры:

- тип поведения системы при обнаружении ошибки: активное или пассивное. Система может активно предвосхищать ошибки (например, проверяя корректность данных, введенных пользователем) или пассивно реагировать на них только в том случае, если избежать их не удалось (например, когда введенные пользователем данные привели к численному переполнению). В том или ином случае сделанный выбор повлияет на дизайн пользовательского интерфейса. Если в спецификации архитектуры не определена единственная согласованная стратегия, пользовательский интерфейс окажется непонятной комбинацией разных решений, относящихся к разным частям программы. Так же, при проектировании архитектуры нужно определить соглашения вывода сообщений об ошибках;
- стратегия обработки исключения. Архитектура должна определять, когда код может генерировать исключения, где они будут перехватываться, как документироваться. Архитектура определяет на каком уровне программы обрабатываются ошибки: обработка может осуществляться в точке их обнаружения, передаваться классу обработки ошибок или возвращаться по цепи вызовов;
- уровень ответственности каждого класса/модуля за проверку получаемых данных: каждый класс отвечает за проверку собственных данных или существует группа классов, проверяющих данные для всей системы;
- реализация механизма обработки ошибок: предоставляемого средой разработки, или создание собственного, комбинация того и другого.

2.2.12 Стратегия изменений

Разработка программы, как правило, редко происходит по точным и исчерпывающим спецификациям, поэтому в этот период приложение претерпевает самые активные изменения своей структуры. И в дальнейшем, при сопровождении продукта, могут добавляться новые возможности, изменяться существующие (типы данных, форматы файлов и тому подобное). Поэтому архитектура должна быть достаточно гибкой, чтобы в систему можно было легко внести вероятные изменения. Иными словами, архитектура должна предвосхищать будущие потенциально возможные изменения.

2.3 Модель архитектуры систем инженерного анализа

Подведем итог по списку компонентов архитектуры, учитывающий специфику разработки расчетного ПО.

Прежде всего, отметим, что архитектура подобного типа ПО должна реализовывать многослоевую модель. В приложении, строящимся с применением многослоевой модели, разнообразие функциональных возможностей распределяется между несколькими слоями по некоторым четко формализуемым признакам. В общем случае, для расчетных программ, может быть реализована трехуровневая модель, включающая уровень представления (препроцессор данных, постпроцессор результатов решения), уровень рабочей логики (или уровень решателя) и уровень доступа и валидации данных. В этом случае выстраивается вертикальная иерархия уровней архитектуры, где уровень представления располагается сверху, уровень алгоритмического представления задачи посередине и уровень доступа к данным внизу. Уровень доступа к данным решает задачи инкапсуляции деталей хранения, реализации различных технических представлений и проверки данных.

При использовании объектно-ориентированного подхода к разработке, создаваемые для каждого из уровней классы должны ориентироваться на цели и задачи только своего уровня. При этом следует тщательно избегать проникновения задач одного уровня на другой, особенно если это касается проникновения конкретных деталей реализации с более высокого уровня на нижележащий. В равной мере, подобное проникновение недопустимо и в обратном направлении, так как изоляция уровней осуществляется в обоих направлениях. В качестве характерной иллюстрации такого «просачивания» выступает непосредственное использование классов рабочей логики (уровень решателя) в «элементах управления», относящихся к уровню представления.

В конечном итоге, использование многоуровневой архитектуры программы позволит избежать переработки всего приложения при внесении изменений в требования, реализуемые в одном из уровней. Следует отметить, что доступная степень изоляции слоев зависит, прежде всего, от самой задачи, решаемой конкретным приложением, тем не менее, большинство программ имеют тенденцию к росту сложности сверх запланированной при запуске проекта в разработку. Причем это усложнение выражается как раз в необходимости введения дополнительной функциональности (средств контроля информации, потоков данных и правил обработки). Все эти средства будут естественным образом располагаться на уровне рабочей логики, а если изначально этот уровень в архитектуре не предусмотрен, то придется выбирать между встраиванием этого кода в уровень представления или уровень доступа к данным (при этом все

преимущества четкого разделения уровней теряются). Добавление кода и новых компонентов в существующую структуру по мере развития приложения является совершенно естественным, в то время как разрушение существующей архитектуры или изменение архитектурных решений в ходе разработки будут иметь крайне негативные последствия: полный беспорядок в коде, появление непрогнозируемых побочных эффектов, усложнение сопровождения (зачастую делая его практически невозможным).

Управление ресурсами, интернационализация/локализация, ввод-вывод определяются на уровне доступа к данным. Все эти компоненты архитектуры должны быть выражены конкретными группами классов/модулей со строго разграниченными сферами ответственности. Именно такая инкапсуляция качеств позволит оперативно решать возникающие проблемы сопровождения ПО, связанные с переходом к новым актуальным версиям операционной системы и масштабируемостью. Иными словами, вся функциональность, связанная с динамическим или статическим выделением памяти (если работа с памятью не поддерживается автоматически средой исполнения), определением различных строковых ресурсов, ссылающихся на ресурсы операционной системы, определением форматов файлов ввода-вывода, реализацией взаимодействия с базами данных должны быть представлены отдельными классами/модулями на уровне доступа к данным с подробными комментариями.

3. Процесс конструирования

3.1 Подготовка процесса конструирования

После того, как завершен этап высокоуровневого планирования (выработано приемлемое архитектурное решение), основное внимание уделяется решениям вопросов подготовки этапа конструирования.

3.1.1 Выбор технологической платформы

Грамотный выбор технологической платформы во многом определяет жизненный цикл программного продукта и перспективы его развития. Естественно, такой выбор осуществляется не случайно, а исходя из определенных критериев, при том, что некоторые из них определяются интуитивно. Основной критерий выбора является следствием назначения программного средства (диктуется требуемой функциональностью). Такое положение дел в большей степени справедливо для специализированных программных продуктов, к которым и относятся системы инженерного анализа. В частности, высокопроизводительные коды научного и инженерного анализа ориентируются на технологии кластерных и серверных (MIMD архитектуры) вычислений, в то время как код, в той или иной степени автоматизирующий некоторую специфическую инженерную методику расчета, не так требователен к производительности, как к удобству эксплуатации и широтой презентации обрабатываемой информации.

При выборе технологической платформы следует учесть и такой немаловажный критерий, как зрелость ее технической среды поддержки. Такие среды предоставляют широкий выбор языков программирования, мощные средства поиска ошибок, эффективные инструменты отладки и надежные автоматизированные средства оптимизации производительности приложений. Компиляторы учитывают особенности технологической платформы и не содержат

критических ошибок. Все инструменты интегрированы (благодаря чему можно разрабатывать пользовательский интерфейс, модули работы с базами данных, составлять различные формы отчетов и реализовать расчетную логику в одной среде) и хорошо документированы производителями (имеется достаточно методического материала). Кроме того, доступны разнообразные консалтинговые услуги и программы обучения.

3.1.2 Выбор языка программирования

Выбор языка программирования напрямую определяет возможность и эффективность решения поставленной задачи. Исследования показывают, что выбор языка программирования различными способами влияет на производительность труда программистов и качество создаваемого ими кода. В частности, высокоуровневые языки выразительнее низкоуровневых и, как следствие, более продуктивны [7, 8, 9]. Так же, каждый язык в той или иной мере выражает определенную концепцию (парадигму или вычислительную модель) программирования. Качественный выбор парадигмы, наилучшим образом соответствующей поставленной проблеме, определяет в конечном итоге успешность ее решения. Для разработки расчетного программного обеспечения наилучшим выбором будут являться структурная парадигма, и отчасти, объектно-ориентированная парадигма.

Приведем частичное описание языков указанных моделей вычислений, которые находят применение в процессе разработки расчетного ПО.

Fortran — язык высокого уровня со строгой статической типизацией. Язык проектировался для широкого использования в научных и инженерных вычислениях. Одним из его преимуществ — большое количество написанных на нём программ и библиотек подпрограмм. Большинство таких библиотек доступны в исходных кодах, хорошо документированы, отлажены и весьма эффективны [10]. Современные реализации (**Fortran**-95 и **Fortran**-2003) приобрёл черты, необходимые для эффективного программирования на новых вычислительных архитектурах, позволяет применять современные технологии программирования, в частности, ООП.

С момента первоначальной разработки языка, его компиляторы производит IBM. В настоящее время IBM поставляется оптимизирующий компилятор *VS Fortran* для мэйнфреймов IBM System z, компилятор *XL Fortran* для платформ на базе архитектуры PowerPC — AIX, Linux для суперкомпьютера Blue Gene. До 1997 крупным производителем компилятора была Microsoft, отказавшаяся в дальнейшем от его поддержки. В дальнейшем, этим компилятором занималась DEC, поглощенная в 1998 году Compaq, которая в свою очередь, вошла в 2002 году в HP. Данная компания поставляет компилятор, интегрированный в среду разработки Digital Visual Fortran, основанную на Microsoft Visual Studio. Наиболее известными продуктами этой линейки являются *FPS 4.0* (*Microsoft Fortran Power Station*), *DVF 5.0* и *DVF 6.0*. С некоторых пор, разработкой и поддержкой версии компилятора для Windows стала заниматься фирма Intel (*Intel Fortran Compiler*), который позволяет оптимизировать код под платформы Intel IA-32, x86_64 и IA-64. Долгое время лучшим компилятором **Fortran** для PC считался компилятор фирмы Watcom, который был выделен в отдельный проект Open Watcom, развивающий компилятор на открытой основе. Среди других бесплатных компиляторов языка можно отметить компилятор от Oracle, входящий в состав Sun Studio, который генерирует

нативный код под SPARC, x86 и x86-64 и доступен для ОС Solaris, OpenSolaris и GNU/Linux. Фонд свободного программного обеспечения GNU поддерживает открытый компилятор *GFortran*, в котором реализованы практически все конструкции стандарта **Fortran**-95 и многие конструкции стандартов **Fortran**-2003, **Fortran**-2008.

В качестве положительных аспектов использования языка, как средства решения прикладных задач, отметим следующие. Жёсткая стандартизация языка позволяет относительно легко переносить код на различные платформы. Новые стандарты языка в значительной мере сохраняют преемственность с более старыми, что позволяет использовать коды ранее написанных программ [11].

Язык имеет большой набор встроенных математических функций, поддерживает работу с целыми, вещественными и комплексными числами высокой точности. Выразительные средства языка изначально были весьма бедны, поскольку он был одним из первых языков высокого уровня. В дальнейшем были добавлены многие лексические конструкции, характерные для структурного и даже объектно-ориентированного программирования. Структура программ изначально была ориентирована на ввод с перфокарт и имела ряд удобных именно для этого случая свойств. Когда использование перфокарт было прекращено, эти достоинства превратились в серьёзные неудобства. Именно поэтому в стандарт языка, начиная с **Fortran**-90, в добавление к фиксированному формату исходного текста появился свободный формат, который не регламентирует позиции строки, а также позволяет записывать более одного оператора на строку. Введение свободного формата позволило создавать код, читаемость и ясность которого не уступает коду, созданному при помощи других современных языков программирования, таких как **C** или **Java**. Отличительной чертой первых версий языка является большое количество меток, которые использовались как в операторах безусловного перехода, так и в операторах циклов, и в операторах описания форматного ввода-вывода, что делает программы трудными для понимания. Именно этот негативный опыт стал причиной, по которой в ряде современных языков программирования метки и связанные с ними операторы безусловного перехода сильно видоизменены. Современные версии языка избавлены от этого недостатка за счёт введения таких новых операторов цикла и условного перехода. Вычисляемый оператор безусловного перехода, а также конструкция множественного входа в процедуры, были исключены. Также к положительным чертам современной версии языка можно отнести большое количество встроенных операций с массивами.

Многие системы программирования позволяют компоновать полученные в результате трансляции **Fortran** программы объектные файлы с объектными файлами, полученными от компиляторов с других языков, что позволяет создавать более гибкие и многофункциональные приложения. Для языка **Fortran** также доступно большое количество библиотек, содержащих как подпрограммы решения классических вычислительных задач (LAPACK, IMSL, BLAS), задач организации распределённых вычислений (MPI, pvm), так и задач построения графических интерфейсов (Quickwin, FORTRAN/TK, Photran 8.1 для Eclipse 4.3 (Juno) и CDT 8.2.) или доступа к СУБД (Oracle).

В заключении отметим, что, несмотря на все выше отмеченные положительные качества языка, имеется существенное ограничение для его предпочтительного использования в качестве средства разработки сложного прикладного ПО — это строгая статическая система типов. Даже новые стандарты

языка, позволяющие использовать ООП, не могут обеспечить оптимальное разрешение проблемы сложности, возникающие при проектировании и реализации систем с нетривиальным поведением. Существенные ограничения, накладываемые типизацией, приводят к отсутствию возможностей использования, например, порождающего метапрограммирования, функционального программирования. Иными словами, уровень абстрагирования при решении проблемы сложности слишком низок и, как следствие, возрастают затраты на реализацию и поддержку проекта на актуальном уровне (уровне современного развития информационных технологий).

Delphi — императивный, структурированный, объектно-ориентированный язык программирования со статической сильной типизацией, диалект **Object Pascal**. При создании языка, не ставилась задача обеспечить максимальную производительность исполняемого кода или его лаконичность при работе с памятью. Изначально, язык предназначался для обучения дисциплине программирования и преследовал цели высокой читаемости кода. В дальнейшем, как по мере роста аппаратных мощностей, так и в результате появления новых парадигм, язык расширялся новыми конструкциями.

На сегодняшний день существует множество компиляторов языка **Delphi** для различных платформ (Windows 32/64, Apple Mac OS X, iOS, Google Android):

- *Embarcadero Delphi* — наиболее популярная интегрированная среда разработки ПО для языка **Delphi**, которая позволяет осуществлять создание прикладного ПО под операционные системы Windows, Mac OS X, iOS и Android;
- *Free Pascal* (FPC) — свободный компилятор **Object Pascal**, который поддерживает различные диалекты языка, включая **Turbo Pascal**, **Delphi** и собственные диалекты. Предназначается для платформ x86, x86-64, PowerPC, SPARC и процессоров ARM, и различных операционных систем (Windows, Linux, FreeBSD, Mac OS). Существует несколько сред разработки программного обеспечения для FPC (один из известных представителей — Lazarus);
- **Oxygene** — язык программирования со сходным **Object Pascal** синтаксисом, поддерживающий интеграцию с Microsoft Visual Studio, использующий .NET и совместимые с ней платформы. В настоящий момент продаётся под брендом Embarcadero Delphi Prism;
- **MIDletPascal** — язык программирования с **Delphi**-подобным синтаксисом, и одноимённый компилятор, преобразующий исходный код в байт-код **Java**.

В качестве достоинств языка следует, прежде всего, отметить низкий порог вхождения (затраты на освоение), обусловленный тем фактом, что язык создавался с целью обучения программированию. Так же язык позволяет использовать различные стили написания программ: императивный, ООП, обобщенное программирование (с некоторыми оговорками). Хотя возможности типизации языка не так широки, как, например, в языках **C++**, **C#** (что заметно снижает разнообразие решений сложных задач), язык успешно используется для разработки небольших приложений с native кодом, поскольку системные вызовы в Windows (как и в POSIX-системах наподобие Linux, Mac OS) формально необъектны и взаимодействие ОО кода с ними затруднено (даже без учета разной парадигмы управления временем жизни переменных в памяти).

Недостатки языка проистекают из целей его создания — язык для обучения дисциплине программирования. В виду этого, он имеет довольно сдержанные возможности типизации, управления памятью, типов и структур данных.

C++ — компилируемый статический слабо типизированный язык программирования общего назначения. Поддерживает такие парадигмы программирования как структурное программирование, объектно-ориентированное программирование, обобщённое программирование. Язык обеспечивает модульность, раздельную компиляцию, обработку исключений, абстракцию данных, объявление типов (классов) объектов, виртуальные функции. Стандартная библиотека включает, в том числе, общеупотребительные контейнеры и алгоритмы. **C++** сочетает свойства как высокоуровневых, так и низкоуровневых языков [12]. Язык широко используется для разработки программного обеспечения, и область его применения включает создание операционных систем, разнообразных прикладных программ, драйверов устройств, приложений для встраиваемых систем, высокопроизводительных серверов. В целом язык спроектирован и развивается как мультипарадигменный, впитывающий в себя различные методы и технологии программирования, и реализующий их на платформе, обеспечивающей высокую техническую эффективность. Существует множество реализаций языка для различных платформ, как бесплатных, так и коммерческих. Например, на платформе x86 это *GCC*, *Visual C++*, *Intel C++ Compiler*, *Embarcadero C++ Builder* и другие. Стандарт языка состоит из двух основных частей: описание ядра языка и описание стандартной библиотеки. Стандартная библиотека включает в себя набор средств, которые должны быть доступны для любой реализации языка, чтобы обеспечить программистам удобное пользование языковыми средствами и создать базу для разработки, как прикладных приложений самого широкого спектра, так и специализированных инструментов. Библиотека содержит следующие разделы: поддержка языка, стандартные контейнеры, основные утилиты, итераторы, алгоритмы, строки, ввод-вывод, локализация, диагностика, числа. Кроме того, существует огромное количество библиотек, не входящих в стандарт, а также имеется возможность использовать многие библиотеки **C**.

В качестве основного достоинства языка можно отметить широкий спектр решаемых с помощью него задач, а именно, **C++** содержит средства разработки программ контролируемой эффективности от низкоуровневых утилит и драйверов до весьма сложных программных комплексов. Такая универсальность достигается за счет следующих особенностей:

- вычислительная производительность. Язык позволяет контролировать все аспекты структуры и порядка выполнения программы, также имеется возможность работы с памятью на низком уровне;
- поддержка различных стилей программирования: традиционное императивное программирование (структурное, объектно-ориентированное), функциональное программирование, порождающее метапрограммирование;
- шаблоны **C++** дают возможность построения обобщённых контейнеров и алгоритмов для разных типов данных. Использование шаблонов расширяет возможности языка для использования парадигм, которые не поддерживаются компиляторами напрямую. Например, библиотека Boost.Bind позволяет связывать аргументы функций. Используя

шаблоны и множественное наследование, можно имитировать классы-примеси и комбинаторную параметризацию библиотек;

- возможность встраивания предметно-ориентированных языков программирования в основной код. Такой подход использует, например библиотека Boost.Spirit, позволяющая задавать EBNF-грамматику парсеров прямо в коде **C++**;
- низкий порог вхождения (до определенного уровня). Для изучения языка существует большое количество учебной литературы.

Широкие возможности языка позволяют его сторонникам позиционировать его как «универсально применимый» — вплоть до отождествления «применимости» с Тьюринг-полнотой (что является ошибкой) и одновременно с оптимальностью, то есть обоснованностью выбора его в качестве инструмента для данной конкретной задачи. При этом ни одной конкретной задачи не обозначается, а наоборот, делается утверждение, что язык подходит для любой задачи (что теоретически невозможно). Однако **C++** не отвечает многим требованиям качества программирования (которые не предъявляются к **C**, в силу узкой специфики данного языка), важными для широкого спектра задач прикладного программирования. Синергетика таких требований находит свое выражение в понятии «результативность». Для заказчиков, выбирающих язык для реализации задуманных ими проектов, важны соотношение факторов результативности с затратами на разработку и общая дисциплина и культура программирования. С этой точки зрения, выбор языка **C++** для реализации крупных проектов прикладного ПО не является оптимальным по следующим критериям:

- ручное управление памятью. Как отмечается в исследовании [13], программисты на **C** тратят 30% — 40% общего времени разработки только на управление памятью, что вполне оправдано в низкоуровневых задачах (для которых и разработан **C**). Однако, в прикладных задачах широкого спектра (на которые претендует **C++**) это не только является напрасным, но и чревато ошибками. Хотя для **C++** существуют библиотечные средства автоматического управления памятью, они применяются далеко не всегда и, кроме того, их эффективность ограничена;
- менеджмент проектов. Сложность менеджмента проектов на **C++** является одной из самых высоких в индустрии разработки ПО [14];
- идеология компонентности. Объектная модель **C++**, унаследованная от **Simula 67** и дополненная двумя формами множественного наследования (простой и виртуальной), имеет не только объективные проблемы, но и опасную идеологию [15] и, как следствие, показатель повторного использования кода оказывается крайне низким, показатели ясности, модифицируемости и тестируемости — слабыми. В силу того, что реализация указателей на методы классов не стандартизирована, и их размер в различных компиляторах варьируется в диапазоне от 4 до 20 байт, значительно снижается портируемость программ с их использованием [16]. Методология декомпозиции задач, принятая в сообществе **C++**, приводит к проектным решениям, не доказуемым математически и не адекватным предметной области, что является причиной неоправданных затрат при сопровождении проекта (низкая

культура программирования). Так, традиционно в математике и, соответственно, в более строгих языках программирования, понятие «класс» отождествляется с понятием «множества», понятие «наследования классов» означает создание «подмножеств» или «подтипов» (по Хаскеллу Карри, тип определяется как множество значений). В **C++** эта традиция не соблюдается и наследование часто используется вместо вложения — то есть определение нового типа на основании более простых взаимно-ортогональных типов осуществляется посредством создания не совершенно нового типа, инкапсулирующего эти типы (и ортогонального им), а «их общего подтипа» (несмотря на то, что ортогональность означает отсутствие точек соприкосновения) [17]. Такой же идеологии придерживаются некоторые библиотеки, опираясь на возможность приведения типов вверх и вниз по иерархии классов, подтверждают тем самым, что типобезопасность не входит в традиции языка. Множественное наследование еще больше осложняет ситуацию. Ошибочность проектных решений, принятых в соответствии с этой идеологией, может обнаруживаться на поздних этапах разработки и, из-за высокой связности, потребовать повторной разработки значительных частей проекта [18];

- качество и культура программирования. Тогда как другие языки обычно предлагают чётко очерченный баланс между порогом вхождения и результативностью программиста (чем выше порог вхождения, тем выше результативность), **C++** (имея низкий порог вхождения) не накладывает ограничение на программиста в вопросе использования «плохих» языковых средств, даже если для этой же цели имеются «хорошие» решения. Другими словами, язык предоставляет программисту возможность выбора средств решения проблемы в соответствии с его личными предпочтениями и уровнем знаний. В связи с этим, появляется проблема, заключающаяся в условии правильного понимания программистом целей, для которых были предусмотрены те или иные возможности языка. Возможность **C++** «не навязывать «хороший» стиль программирования» лишь снижает порог вхождения и, с точки зрения качества ПО, является недостатком. При высоких требованиях качества (не только в аспекте надёжности) предпочтительным выбором являются языки с семантикой, сводящей к минимуму риск влияния человеческого фактора, то есть именно навязывающие «хороший» стиль программирования (**Ada**, **StandardML**, **Haskell**) — но такие языки имеют более высокий порог вхождения. Следует отметить, что элементы семантики **C++**, отсутствующие в **C**, были заимствованы из других языков, и очевидно, что для адекватного и аккуратного применения взаимно противоречивых возможностей и избегания провоцируемых ими ошибок, программист на **C++** должен изначально знать непосредственно эти языки и лежащую в их основе формальную базу. Этого, однако, не наблюдается: практика показывает, что преимущественно положительная оценка и предпочтение использования **C++** характерны лишь для тех, кто не владеет ни одним

языком, не являющимся потомком **Algol** (не считая ограниченного набора узко специализированных языков, таких как **SQL**, **HTML**, **Perl** и других — зачастую даже не полных по Тьюрингу). Программисты же более высокого уровня обычно отзываются о **C++** не так оптимистично и стараются его избегать, если нет нужды поддерживать существующий код [19];

- ненадёжность продукта. Наличие побочных эффектов (даже в таких простых операциях как индексация массива), в сочетании с отсутствием контроля со стороны системы времени исполнения и слабой системой типов, делает программы на **C++** традиционно нестабильными, что исключает применение языка при высоких требованиях к отказоустойчивости кода. Преодоление указанных негативных явлений увеличивает длительность процесса разработки, в тоже время, например, наличие контроля типов заметно его сокращает [20]. В частности, семантика языка **Forth** обеспечивает практически гарантированное выявление подобного рода ошибок на этапе разработки, а **Eiffel**, **Smalltalk** и **Erlang** предоставляют простые способы обработки любых возможных ошибок самой программой без обрушения.

Java — объектно-ориентированный язык программирования со строгой полиморфной статической системой типов, разработанный компанией Sun Microsystems (в последующем приобретённой компанией Oracle). Особенностью языка является трансляция исходного кода в байт-код, выполняемый виртуальной машиной Java (JVM) — программой, обрабатывающей байтовый код и передающей инструкции оборудованию как интерпретатор.

Существуют несколько основных семейств технологий **Java**:

- Java SE — Java Standard Edition, основное издание **Java**, содержит компиляторы, API, Java Runtime Environment; подходит для создания пользовательских приложений, в первую очередь — для настольных систем;
- Java EE — Java Enterprise Edition, представляет собой набор спецификаций для создания программного обеспечения уровня предприятия;
- Java ME — Java Micro Edition, создана для использования в устройствах, ограниченных по вычислительной мощности, например в мобильных телефонах, КПК, встроенных системах;
- JavaFX — технология, являющаяся Rich Client Platform и предназначена для создания графических интерфейсов корпоративных приложений и бизнеса;
- Java Card — технология предоставляет безопасную среду для приложений, работающих на смарт-картах и других устройствах с очень ограниченным объёмом памяти и возможностями обработки.

Для платформы **Java** существуют следующие средства разработки ПО:

- JDK — содержит наборы библиотек для платформ Java SE и Java EE, компилятор командной строки javac и набор утилит, также работающих в режиме командной строки;

- NetBeans IDE — свободная интегрированная среда разработки для всех платформ Java — Java ME, Java SE и Java EE. Oracle продвигала NetBeans IDE как базовое средство для разработки ПО на языке **Java** и других языках (**C**, **C++**, **PHP**, **Fortran**);
- Eclipse IDE — свободная интегрированная среда разработки модульных кроссплатформенных приложений, в том числе и для Java SE, Java EE и Java ME [21]. Пропагандируется Eclipse Foundation как базовое средство для разработки ПО на языке **Java** и других языках (**C**, **C++**, **Ruby**, **Fortran**);
- IntelliJ IDEA — среда разработки для платформ Java SE, Java EE и Java ME. Разработчик — JetBrains. Среда распространяется в двух версиях: свободной бесплатной (Community Edition) и коммерческой проприетарной (Ultimate Edition);
- JDeveloper — бесплатная интегрированная среда разработки для платформ Java SE, Java EE и Java ME от компании Oracle.

Безусловным достоинством языка **Java** является высокий показатель портируемости кода [22]. Такие возможности достигаются за счет полной независимость байт-кода от операционной системы и оборудования, что позволяет выполнять Java-приложения на любом устройстве, для которого существует соответствующая виртуальная машина. Другим достоинством технологии является гибкая система безопасности, обеспечивающаяся полным контролем виртуальной машины над выполнением программы. Любые операции, которые превышают установленные полномочия программы (например, попытка несанкционированного доступа к данным или сетевого соединения) вызывают немедленное прерывание.

К недостаткам концепции виртуальной машины относится тот факт, что исполнение байт-кода может снижать производительность программ и алгоритмов. В частности, по некоторым данным, для разных задач время выполнения **Java** кода составляет в среднем в полтора-два раза больше, чем для **C/C++** [23] и потребление памяти Java-машиной было в 10-30 раз больше, чем программой на **C/C++**. Также существует исследование, проведённое компанией Google, согласно которому отмечается существенно более низкая производительность и бóльшее потребление памяти в тестовых примерах на **Java** в сравнении с аналогичными программами на **C++** [24].

Для преодоления указанных недостатков, в последнее время был внесен ряд усовершенствований, которые несколько увеличили скорость выполнения программ на **Java**:

- применение технологии трансляции байт-кода в машинный код непосредственно во время работы программы (JIT-технология) с возможностью сохранения версий класса в машинном коде;
- широкое использование платформенно-ориентированного кода (native-код) в стандартных библиотеках;
- аппаратные средства, обеспечивающие ускоренную обработку байт-кода (например, технология Jazelle, поддерживаемая некоторыми процессорами фирмы ARM).

C# — объектно-ориентированный язык программирования со статической, динамической, строгой, «утиной», системой вывода типов типизацией. Разработан Microsoft как язык разработки приложений для платформы Microsoft .NET

Framework и впоследствии был стандартизирован как ECMA-334 и ISO/IEC 23270. Возможности языка, прежде всего, определяются возможностями CLR (Common Language Runtime) — виртуальной машины, интерпретирующей и исполняющей код на языке **CIL**, в который компилируются программы, написанные на .NET-совместимых языках программирования. Это касается, прежде всего, системы типов **C#**. Иными словами, присутствие или отсутствие тех или иных выразительных особенностей языка диктуется тем, может ли конкретная языковая особенность быть транслирована в соответствующие конструкции **CLR**.

Среди достоинств языка можно отметить поддержку множества парадигм программирования, в частности, обобщенное программирование. Поддержка обобщённости интегрирована в саму виртуальную среду выполнения, а язык стал внешним интерфейсом для доступа к этой возможности. Широкие возможности системы типов языка, отражающая BCL (Base Class Library — стандартная библиотека классов платформы .NET Framework), позволяют реализовывать прикладные проекты высокой сложности. Отдельно следует отметить возможность использования языка для разработки приложений на технологической платформе Windows Presentation Foundation. Эта система предназначается для построения клиентских приложений Windows с широкими возможностями построения сложных графических интерфейсов. Основу WPF составляет векторная система визуализации, не зависящая от разрешения устройства вывода и созданная с учётом возможностей современного графического оборудования. Платформа предоставляет средства для создания визуального интерфейса, включая язык **XAML** (Extensible Application Markup Language), элементы управления, привязку данных, макеты, двухмерную и трёхмерную графику, анимацию, стили, шаблоны, документы, текст, мультимедиа и оформление [25]. Графической технологией, лежащая в основе WPF, является DirectX, а не Windows Forms, где используется GDI/GDI+. За счет этого достигается высокая производительность приложений, интенсивно использующих графику в своей работе.

Следует отметить, что коду, созданному с использованием языка **C#**, будут сопутствовать известные проблемы производительности, характерные для всех платформ разработки, основывающиеся на использовании подхода виртуальной машины. Ситуация усугубляется еще и тем фактом, что такой код будет иметь низкий показатель портируемости, так как платформа .NET Framework является закрытой для сторонних разработчиков. В качестве решения последней указанной проблемы выступает проект по созданию полноценного воплощения системы .NET Framework на базе свободного программного обеспечения — Mono [26]. Реализации Mono существуют для следующих операционных систем: Windows, Linux, BSD (FreeBSD, OpenBSD, NetBSD), Solaris, Mac OS X, Apple iOS, Wii. Поддерживаются аппаратные платформы: s390, SPARC, PowerPC, x86/x86-64, IA64, ARM, Alpha, MIPS, HPPA. Еще одним проектом по созданию аналога платформы Microsoft .NET на базе свободного программного обеспечения является DotGNU организации Free Software Foundation (FSF).

Python — высокоуровневый язык программирования общего назначения со строгой динамической типизацией, ориентированный на повышение производительности разработчика и читаемости кода. Стандарт языка состоит из двух частей: ядро языка (с минималистичным синтаксисом) и стандартная библиотека, включающая большой объём полезных функций. **Python** позволяет использовать несколько парадигм программирования: структурное, объектно-

ориентированное, функциональное, императивное и аспектно-ориентированное. Язык имеет механизмы автоматического управления памятью, полной интроспекции, обработки исключений, поддержки многопоточных вычислений.

Эталоном реализации языка является интерпретатор *CPython*, поддерживающий большинство активно используемых платформ [27] и распространяющийся под свободной лицензией Python Software Foundation License. Есть реализации интерпретаторов для JVM (с возможностью компиляции — *Jython*), MCIL (с возможностью компиляции — **IronPython**, **Python.Net**), LLVM. Язык широко портирован под различные ОС: Windows, практически все варианты UNIX (включая FreeBSD и Linux), Plan 9, Mac OS и Mac OS X, iPhone OS 2.0 и выше, Palm OS, OS/2, Amiga, HaikuOS, Windows Mobile, Symbian и Android. При этом, для всех основных платформ **Python** имеет поддержку характерных для данной платформы технологий (например, Microsoft COM/DCOM).

Существенную роль в привлекательности языка, как средства разработки, играет его стандартная библиотека, предлагающая разработчику широкий набор модулей для работы с сетевой инфраструктурой (множество сетевых протоколов и форматов Internet), администрированием ОС (позволяют создавать кросс-платформенные приложения) и множество других возможностей (регулярные выражения, текстовые кодировки, мультимедийные форматы, криптографические протоколы, архивы, сериализация данных, поддержка юнит-тестирования и др.). Помимо стандартной библиотеки существует множество прикладных библиотек, существенно расширяющих спектр применимости языка (веб, базы данных, обработка изображений, обработка текста, численные методы, приложения операционной системы и т. д.).

Таким образом, среди достоинств языка **Python** как платформы разработки пользовательского ПО можно отметить следующие:

- система типов языка позволяет реализовать различные парадигмы при написании программ. Помимо императивной и хорошо продуманной ООП, имеется возможность использования функционального программирования;
- высокая культура программирования, поддерживаемая на уровне языка;
- широкие возможности стандартной библиотеки и огромный выбор сторонних прикладных библиотек;
- широкий спектр портирования языка (множество ОС и вычислительных архитектур);
- немаловажным является тот факт, что **Python** и подавляющее большинство библиотек к нему бесплатны и поставляются в исходных кодах. Более того, лицензия никак не ограничивает использование языка в коммерческих разработках и не налагает никаких обязательств, кроме указания авторских прав (в отличие от многих других открытых систем).

Из недостатков отметим следующее — это низкое быстродействие. Как и многие другие интерпретируемые языки, классический **Python**, не применяющий, в частности, JIT-компиляторы, имеет существенный недостаток — невысокую скорость выполнения программ [28]. Для устранения проблемы предлагаются следующие подходы: сохранение байт-кода (файлы .pyc и .pyo), что позволяет интерпретатору не тратить время на перекомпиляцию кода модулей при каждом

запуске; специальный JIT-компилятор Psyco [29], позволяющий ускорить выполнение программ (только x86 совместимый код), но приводящая к увеличению потребления оперативной памяти; реализации языка, вводящие высокопроизводительные виртуальные машины в качестве back-end компилятора, например, PyPy (до 2017 г.).

3.1.3 Основные модели разработки программного обеспечения

3.1.3.1 Процесс проектирования

Процесс разработки программного обеспечения регламентируется различными моделями проектирования. Проектирование в данном контексте является отдельной специфической IT-дисциплиной. Однако, в зависимости от обстоятельств (размер проекта и средств, выделяемых на его создание) процесс проектирования может быть реализован на разных этапах разработки. Так, для небольших проектов проектирование является сопутствующим процессом при конструировании. С другой стороны, и в некоторых достаточно серьезных проектах формальная архитектура дает ответы только на вопросы системного уровня, оставляя значительную часть проектирования опять на этап конструирования. И лишь в довольно редких случаях, когда проектирование проведено в достаточном объеме на этапах выработки требований и создания плана конструирования, сам процесс кодирования становится почти механическим. Так или иначе, от тщательного выполнения процесса проектирования выигрывают проекты любого масштаба.

Рассмотрим наиболее популярные модели проектирования, которые могут использоваться при конструировании.

3.1.3.2 Итеративная модель проектирования

Итеративная модель в разработке программного обеспечения — это выполнение работ параллельно с непрерывным анализом полученных результатов и корректировкой предыдущих этапов работы. Применительно к конструированию, логически замкнутый фрагмент кода при использовании такой модели в каждой фазе развития проходит повторяющийся цикл: Планирование — Реализация — Проверка — Оценка.

Преимуществами применения итеративной модели являются:

- снижение воздействия серьёзных ошибок на ранних стадиях кодирования, что ведет к минимизации затрат на их устранение;
- организация эффективной обратной связи программиста с заказчиком, и как следствие, создание продукта, реально отвечающего его потребностям;
- концентрация усилий на наиболее критичных участках кода;
- непрерывное итеративное тестирование фрагмента кода, позволяющее в перспективе, оценить успешность всего проекта в целом;
- раннее обнаружение конфликтов между требованиями, моделями и реализацией проекта;
- реальная оценка текущего состояния проекта и, как следствие, большая уверенность заказчиков и непосредственных участников в его успешном завершении.

Пример методологии разработки, использующей данную модель — Rational Unified Process [30].

3.1.3.3 Спиральная модель проектирования

Спиральная модель представляет собой процесс разработки программного обеспечения, сочетающий в себе как проектирование, так и постадийное прототипирование (быстрая «черновая» реализация базовой функциональности для анализа работы системы в целом) с целью сочетания преимуществ восходящей и нисходящей концепции проектирования. Отличительной особенностью этой модели, применительно к конструированию, является концентрирование внимания на потенциальных проблемах, влияющих на организацию жизненного цикла фрагмента кода. В модели выделяются следующие наиболее распространённые риски (по приоритетам):

- дефицит специалистов;
- нереалистичные сроки и бюджет;
- реализация несоответствующей функциональности;
- разработка неадекватного пользовательского интерфейса;
- перфекционизм (ненужная оптимизация и оттачивание деталей);
- непрекращающийся поток изменений в требованиях;
- нехватка информации о внешних компонентах, влияющих на фрагмент кода или вовлечённых в интеграцию с ним;
- недостатки в работах, выполняемых аутсорсерами;
- недостаточная производительность получаемой функциональности фрагмента кода.

Как можно видеть, большая часть этих рисков связана с организационными аспектами взаимодействия специалистов в команде разработчиков.

Согласно модели, на каждом витке спирали создается фрагмент или версия программного обеспечения, на котором уточняются цели и характеристики проекта в целом, определяется его качество, планируются работы для следующего витка спирали. Таким образом, углубляются и последовательно конкретизируются все фрагменты проекта, что в результате приводит к выбору наиболее обоснованного варианта программы, который и доводится до окончательной реализации. Каждый виток разбит на четыре сектора: оценка и разрешение рисков, определение целей, разработка и тестирование, выработка планов дальнейших работ.

На каждом витке спирали могут применяться различные модели процесса разработки ПО. Сама модель, таким образом, сочетает в себе возможности прототипирования (этап жизненного цикла фрагмента кода) и каскадной модели. Прототипирование позволяет переходить на следующий этап, не дожидаясь полного завершения работы на текущем. Главная цель, достигаемая при применении такой модели — как можно быстрее показать пользователям (заказчикам) системы работоспособный продукт, активизируя тем самым процесс уточнения и дополнения требований.

Основная проблема спиральной модели — определение момента перехода на следующий виток. Для ее решения вводятся временные ограничения пребывания на каждом витке спирали. Переход осуществляется в соответствии с планом, даже если не вся работа на текущем витке была завершена.

Примером методологии на основе данной модели является RAD (Rapid Application Development). Методология применяется при наличии следующих факторов (возможностей): небольшая команда разработчиков (от 2 до 10 человек); имеется сжатый, но тщательно проработанный производственный график (от 2 до 6 месяцев); существует возможность реализации повторяющегося цикла процесса разработки, при котором разработчики, по мере того, как приложение начинает обретать форму, запрашивают и реализуют в продукте требования, полученные через взаимодействие с заказчиком.

3.1.3.4 Каскадная модель проектирования

Каскадная модель — модель процесса разработки программного обеспечения, в которой процесс разработки выглядит как поток, последовательно проходящий фазы анализа требований, проектирования, реализации, тестирования, интеграции и поддержки.

В оригинальной каскадной модели фазы разработки чередуются в следующем порядке:

- определение требований;
- проектирование;
- конструирование (или «реализация» либо «кодирование»);
- интеграция;
- тестирование и отладка (или «верификация»);
- инсталляция (внедрение);
- поддержка.

Переход от одной фазы к другой происходит строго последовательно и только после полного и успешного завершения предыдущей. Так на первом этапе «определение требований» получается полный и исчерпывающий список требований к ПО (фрагменту кода). Только после этого происходит переход к проектированию, в ходе которого создаются документы, подробно описывающие для программистов способ и план реализации указанных требований. Далее выполняется реализация полученного проекта. На следующей стадии процесса происходит интеграция отдельных компонентов, разрабатываемых различными командами программистов и после того как реализация и интеграция завершены, производится тестирование и отладка продукта (на этой стадии устраняются все недочёты, появившиеся на предыдущих стадиях разработки). На завершающих фазах программный продукт внедряется (инсталлируется) и обеспечивается его поддержка — внесение новой функциональности и устранение ошибок.

Рассматриваемую модель часто критикуют за недостаточную гибкость и объявление самоцелью формальное управление проектом в ущерб срокам, стоимости и качеству [31]. Тем не менее, при управлении сложными проектами формализация часто является бесценной, так как может кардинально снизить многие риски проекта и сделать его более прозрачным. Поэтому даже в PMBOK (Project Management Body of Knowledge) 3-ей версии формально был закреплен только подход «каскадной модели» и не были предложены альтернативные варианты, например итеративная модель ведение проектов (закреплена в PMBOK 4-ой версии в 2009 году).

3.1.3.4 V-Model проектирования

V-Model (или VEE модель) — модель процесса разработки информационных систем, базирующаяся на принципе, заключающимся в том, что детализация проекта возрастает при движении слева направо, одновременно с течением времени. Итерации в проекте производятся по горизонтали, между левой и правой сторонами буквы «V». При разработке информационных систем V-Model является вариацией каскадной модели, в которой задачи (этапы) разработки идут сверху вниз по левой стороне буквы «V», а задачи тестирования — вверх по правой стороне буквы «V». Внутри буквы проводятся горизонтальные линии, показывающие, как результаты каждой из фаз разработки влияют на развитие системы тестирования на каждой из фаз тестирования. Таким образом, приемо-сдаточные испытания основываются, прежде всего, на требованиях, системное тестирование — на требованиях и архитектуре, а компонентное тестирование (тестирование модулей) — на требованиях, архитектуре, интерфейсах и алгоритмах [95].

Преимущества использования данной модели заключаются в следующем:

- особое значение придается планированию, направленному на верификацию и аттестацию разрабатываемого продукта (модуля) на ранних стадиях его разработки. Фаза модульного тестирования сразу подтверждает правильность детализированного проектирования. Фаза тестирования в процессе интеграции подтверждает архитектурное проектирование. Фаза тестирования в процессе введения в эксплуатацию (приемо-сдаточные испытания) подтверждает правильность выполнения этапа требований к продукту и его спецификации;
- предусмотрены аттестация и верификация всех внешних и внутренних полученных данных, а не только самого программного продукта [33];
- частично реализованы преимущества каскадной модели: определение требований выполняется перед разработкой проекта системы; имеется возможность проектирования целого перед разработкой компонентов [32];
- менеджеры проекта могут контролировать процесс разработки, так как в данном случае можно воспользоваться временной шкалой, а завершение каждой фазы является удобной контрольной точкой [32].

Как следовало ожидать, недостатки модели являются следствием ее достоинств. А именно: модель не предусматривает внесение динамических изменений на разных этапах жизненного цикла программы [33]; тестирование требований происходит слишком поздно, вследствие чего невозможно внести изменения, не повлияв при этом на график выполнения проекта [33]; модель не имеет возможности анализа рисков.

3.2 Объектно-ориентированное конструирование

3.2.1 Основной прием решения задач проектирования сложных систем

Характерная черта коммерческого программирования — высокий уровень сложности решаемых задач, иными словами, сложность присуща всем большим программных системам и превышает возможности человеческого интеллекта.

Сложность программного обеспечения — не случайное его свойство, а вызвано тремя основными причинами: сложностью реальной предметной области решаемой задачи; необходимостью обеспечить достаточную гибкость жизненного цикла программы; неудовлетворительными математическими моделями описания поведения больших дискретных систем (конечные автоматы не обладают полнотой по Тьюрингу).

Основным приемом решения проблемы сложности является декомпозиция — деление пространства состояний системы на части, причем такие, что состояния выделенной части могут контролироваться одним человеком.

Рассмотрим два типа декомпозиции: алгоритмическая и объектно-ориентированная, использующиеся при решении вычислительных задач.

3.2.1.1 Алгоритмическая декомпозиция.

В данном случае система представляется в виде совокупности слабо связанных алгоритмов, каждый из которых инкапсулирован в отдельном модуле, участвующим в одном из этапов общего процесса решения. Результаты работы отдельных алгоритмов могут являться исходными данными результирующего общего алгоритма решения поставленной задачи. Как правило, в данном случае используется каскадная и V-Model модели проектирования систем. Данный тип декомпозиции применяется при решении сравнительно несложных, строго формализованных задач, имеющих исчерпывающую математическую модель вычисления.

3.2.1.2 Объектно-ориентированная декомпозиция.

В том случае, когда формализация задачи затруднена, решение может быть найдено с использованием объектно-ориентированной декомпозиции. Рассмотрим некоторые предпосылки (проблемы), побуждающие к использованию декомпозиции данного типа:

- построение простых моделей объектов реального мира. В данном случае поведение объекта описывается отдельным классом, а взаимодействие между объектами декларируется интерфейсами;
- моделирование абстракций некоторого множества объектов реального мира. В данном случае класс формализует характерные признаки или поведенческие шаблоны множества схожих по этим критериям объектов. Здесь следует учесть, что процесс извлечения абстракций из разнообразия сущностей реального мира не детерминирован, и формирование абстракций может быть основано на разных (иногда не совсем удачных) критериях, что в конечном итоге приводит к потере всех преимуществ объектно-ориентированной декомпозиции. Таким образом, нахождение адекватных абстрактных моделей — одна из главных проблем объектно-ориентированного проектирования;
- снижение сложности посредством ее изоляции и сокрытия деталей реализации. Инкапсуляция сложности — хороший побудительный мотив для создания класса. Изолировав сложный алгоритм/код от других алгоритмов/частей программы, можно создать удобные интерфейсы управления (обмена данными) и, в дальнейшем, использовать класс везде, где это необходимо, не зная о его внутренней

работе. Тем самым достигается ряд положительных следствий: минимизация объема кода, снижение числа ошибок, облегчение сопровождения программы. Поиск и исправление ошибок будут проще, если сложность изолирована в классе, а не распределена по всему коду. Последствия исправления ошибок не повлияют на остальной код. Более того, при обнаружении более эффективного алгоритма, произвести модернизацию программы будет проще, заменив/адаптировав всего один класс, в котором этот алгоритм был инкапсулирован. Сокрытие деталей реализации позволяет абстрагироваться от сложных технических решений (доступ к БД, протоколы коммуникации, работа с памятью и т.д.) и сосредоточить свое внимание на поиск эффективных вариантов проектирования;

- создание точек централизованного управления. Такие точки содержат реализации конкретных механизмов преобразования информации и решают задачи экспорта/импорта представлений (работа с базами данных и системными ресурсами, сериализация, интерфейсное представление данных);
- облегчение повторного использование кода. Любая потенциальная возможность повторного использования кода влечет за собой его классовую декомпозицию. Код инкапсулируется в класс и окружается интерфейсом для его использования. Такая процедура в дальнейшем существенно уменьшит накладные расходы по сопровождению ПО;
- упаковка набора операций по критерию. В данном случае, операции (функции), обобщенные по некоторому признаку, инкапсулируются в одно пространство имен, например, имя класса.

3.2.2 Концепция абстрактных типов данных

Абстрактный тип данных (АТД) — это набор, включающий данные и выполняемые над ними операции. Операции описывают данные для остальной части программы и позволяют их изменять. Слово «данные» используется в АТД условно и предметно-зависимо, а операции описывают сценарии поведения клиентов при работе с этими данными. Понимание концепции АТД позволяет избежать распространенной ошибки при объектно-ориентированной декомпозиции сложности, когда класс является лишь выражением некоторого контейнера, содержащего наборы плохо согласующихся друг с другом данных и методов.

С применением АТД становятся доступны следующие возможности:

- возможность сокрытия деталей реализации. Сокрытие информации о типах данных объекта подразумевает, что при необходимости их изменений, такую замену можно осуществить в одном месте, не внося изменений по всему коду;
- более высокая информативность интерфейса объекта. В АТД все однотипные операции объединяются и выражаются через предметно-зависимый интерфейс;
- легкость оптимизации и проверки кода за счет локализации функциональности и высокой информативности интерфейса;
- многократно повышается удобочитаемость и понятность кода;

- гибкое управление областью использования данных, и, как следствие — понижение уровня их потенциальной глобализации;
- возможность работы с сущностями повышенной абстрактности. В этом контексте целесообразно представлять в форме АТД следующие примитивы: распространенные низкоуровневые типы данных (повышать уровень абстрактности в ущерб универсальности), простейшее поведение (облегчает понимание и изменение кода, ограничивает потенциальные следствия изменений в этих примитивах). Следует тщательно выбирать метафору при абстрагировании, полагаясь, прежде всего, на предметно-зависимые характеристики и особенности поведения.

Абстрактные типы данных лежат в основе концепции объектно-ориентированной декомпозиции сложности. В языках, поддерживающих ООП, каждый АТД можно реализовать как отдельный класс (АТД, поддерживающий наследование и полиморфизм).

3.2.3 Интерфейсы классов

Разработку высококачественного класса следует начинать с создания адекватного интерфейса, что подразумевает представление интерфейса как хорошей абстракции, скрывающей детали реализации класса. Иными словами, интерфейс класса — это абстракция реализации класса, скрытой за интерфейсом. Только в этом случае интерфейс будет предоставлять группу методов, четко согласующихся друг с другом (имеющих высокую связность).

Рассмотрим принципы, руководствуясь которыми, можно выстраивать качественные интерфейсы:

- интерфейс класса должен выражать согласованный уровень абстракции. Каждый класс должен являться механизмом реализации некоторого АТД и, причем, только одного. Если класс реализует более одного АТД — его следует реорганизовать. Смешанные абстракции часто возникают, когда класс реализуется при помощи класса-контейнера или других библиотечных классов и этот факт не скрывается. Информация об использовании класса-контейнера не может быть частью хорошей абстракции. Обычно она является деталью реализации, которую следует скрывать. Также встречаются ситуации, когда одна группа методов работает с одной половиной данных, а другая — с другой. То есть, возникает ситуация, когда несколько классов подмешиваются в один. Следует разделить их на более четкие абстракции. По мере внесения изменений в код, смешанные уровни абстракции делают его все менее и менее понятным;
- при построении эффективных интерфейсов большое значение имеет достаточное понимание сути абстракции, которую реализует класс. Некоторые классы реализуют близкие, но, естественно, не тождественные абстракции, поэтому при разработке класса нужно понимать, какую абстракцию должен представлять его интерфейс;
- при построении интерфейса следует отдавать предпочтение его полной программной реализации, которая может быть проверена компилятором, исключая из него семантические (в понятии общей

семантики) элементы (предположения об использовании данного интерфейса в некотором контексте программы). Если от семантической части отказаться не удается, следует снабдить ее подробной документацией, описывающие все условия, при которых данный интерфейс будет работать корректно. Любой аспект интерфейса, не контролирующийся компилятором, является потенциальным источником ошибок;

- признаком хорошо выстроенной абстракции класса является высокий уровень связности (cohesion) его интерфейса, когда для достижения конкретной цели функции выполняют с каждым вычислительным объектом одну или несколько близко соотносящихся друг с другом задач. Таким образом, интерфейс низкой связности указывает на необходимость пересмотра согласованности абстракции, которую реализует класс.

3.2.4 Инкапсуляция

Инкапсуляция является более строгой концепцией, чем абстракция. Основное назначение абстракции — предоставлять модели, позволяющие *игнорировать* детали реализации и тем самым управлять сложностью задачи. Инкапсуляция же *не позволяет* получать сведения о деталях реализации. Эти две концепции связаны: без инкапсуляции абстракция обычно разрушается. Рассмотрим некоторые правила, поддерживающие указанное качество концепции инкапсуляции:

- следует выбирать такой модификатор доступности для класса и его членов, который лучше всего сохраняет целостность абстракции интерфейса. Если предоставление доступа к методу согласуется с абстракцией, можно сделать его открытым, если сделать выбор однозначно не удается, следует установить самый строгий уровень защиты из имеющихся [34, 35];
- при проектировании и реализации класса следует исключить аспекты его взаимодействия с клиентами. Класс выражает свои требования в интерфейсе, и не делает предположений о том, как этот интерфейс будет использоваться. Не следует повышать удобство использования класса за счет легкости чтения его интерфейса;
- следует избегать семантических нарушений инкапсуляции. Реализовать инкапсуляцию синтаксически относительно просто: достаточно объявить внутренние методы и данные класса закрытыми. Достигнуть семантической инкапсуляции гораздо сложнее. Довольно часто встречается ситуация, когда клиентский код зависит от закрытой реализации класса, а не от его открытого интерфейса. Такое положение дел проистекает, прежде всего, от ситуационного удобства проектирования. Таким образом, когда из интерфейса класса не понятно, как его использовать и требуется изучать его реализацию (программирование осуществляется не в соответствии с интерфейсом, а сквозь интерфейс в соответствии с реализацией), разрушается инкапсуляция, а вслед за ней и абстракция. Изучение реализации класса с целью понять, как он работает (когда исключительно по документации интерфейса разобраться с его использованием не удается) — хорошая инициатива, но плохое решение;

- следует избегать жесткого сопряжения (coupling) между классами. Сопряжение характеризует меру взаимодействия классов или функций, причем не на уровне синтаксических единиц (например, числа параметров), а на уровне информационных потоков данных. Сопряжение бывает *неявным* (функции взаимодействуют через глобальные переменные, а классы через открытые поля) или *явным* (функции взаимодействуют через параметры, классы — только через интерфейсный протокол). Неявный характер взаимодействия увеличивает сопряжение и вынуждает разработчика и сопровождающего приложение программиста изучать большие сегменты кода, чтобы понять поток данных в программе.

3.2.5 Проектирование и реализация классов

Прежде всего, рассмотрим некоторые канонические отношения классов, так как хорошее проектное решение класса должно показывать концепцию его использования в системе.

3.2.5.1 Включение (containment) (отношение «содержит»).

Метафора отношения состоит в том, что класс содержит примитивный элемент данных или другой класс. Включение — один из мощных инструментов объектно-ориентированного программирования, хотя в литературе ему уделяют гораздо меньше внимания, чем наследованию. Там, где взаимодействие информационных сущностей характеризуется качеством системной структурности с выраженной синергичностью, использование включения будет являться хорошим проектным решением.

3.2.5.2 Наследование (отношение «является»).

Метафора отношения выражается в том, что один класс специализирует другой. В так называемом «базовом» классе идентифицируются общие элементы (интерфейсы методов, их реализация, данные-члены или типы данных), которые в той или иной мере используются в классах «наследниках» с целью избежать дублирования кода и данных. Применение такого проектного решения оправдано, когда информационное взаимодействие выстраивается как системное с явным акцентом на иерархичность. При выстраивании отношения следует разрешить ряд непростых вопросов доступности, инициализации и переопределения реализаций по вертикали и горизонтали иерархии. Следующие рекомендации позволяют избежать некоторых распространенных ошибок наследования.

Отношение «является» реализовывается при помощи открытого наследования. Другими словами, новый класс «является» более специализированной версией существующего класса. При этом базовый класс формулирует ожидания и ограничения, которым должен будет соответствовать производный класс [34]. Если, по тем или иным причинам, производный класс не собирается полностью придерживаться контракта, определенного интерфейсом базового класса, наследование не применяется. В таком случае применяется включение или вносится изменения на более высоком уровне иерархии наследования. Хорошим критерием уместности применения наследования в каждом конкретном случае является принцип LSP (Liskov Substitution Principle), который выражается в том,

что наследование стоит использовать, только если производный класс действительно «является» более специализированной версией базового класса [36]. Или в другой формулировке: клиенты должны иметь возможность использования подклассов через интерфейс базового класса, не замечая никаких различий [37]. Таким образом, все методы базового класса должны иметь в каждом производном классе то же семантическое значение. В этом случае наследование — мощное средство снижения сложности задачи, если же приходится постоянно помнить о семантических различиях реализаций в подклассах, наследование только повышает сложность.

При проектировании класса следует сразу учитывать возможности наследования. Если такое отношение не планируется, следует сразу запретить его. Выстраивая отношение между классами, раздельно управляйте наследованием интерфейсов и реализаций. Наследуемые методы относятся к одному из трех типов:

- абстрактный переопределяемый метод: производный класс наследует только интерфейс метода;
- переопределяемый метод: производный класс наследует и интерфейс метода, и его реализацию по умолчанию, а также может переопределить эту реализацию;
- непереопределяемый метод: производный класс наследует интерфейс метода, его реализацию по умолчанию, переопределить которую он не может.

При проектировании нового класса с помощью наследования, определите тип наследования каждого метода-члена. Не наследуйте реализацию только потому, что наследуется интерфейс, и не наследуйте интерфейс только для того, чтобы унаследовать реализацию. В последнем случае целесообразно заменить отношение наследования на включение.

Общие данные, интерфейсы и формы поведения следует переместить как можно на более высокий уровень иерархии наследования. При перемещении метода на более высокий уровень следите за целостностью абстракции соответствующего класса.

Классы, которые переопределяют метод, оставляя его пустым, указывают на ошибку проектирования базового класса. Как правило, такая ситуация возникает, когда базовый класс имеет слишком оптимистичное (обобщенное) представление о наследуемых классах. В результате, вместо использования универсального поведения, описываемого интерфейсом базового класса, классы-потомки вынуждены конкретизировать поведение своих объектов с помощью узкоспециальных частных интерфейсов (которые, в свою очередь, могут наследоваться классами нижележащего уровня иерархии), что приводит к бесконтрольному разрастанию дерева иерархии.

Избегайте многоуровневых иерархий наследования. ООП предлагает широкий спектр способов управления сложностью, однако некоторые возможности часто повышают сложность вместо того, чтобы снижать ее. Создание многоуровневых иерархий наследования значительно повышает сложность и, как следствие, число ошибок [38], что вообще говоря, противоположно цели наследования (избежать дублирования кода и минимизировать сложность).

Избегайте множественного наследования. Также как и в случае с многоуровневыми иерархиями, множественное наследование — мощный, но

опасный инструмент управления сложностью. Ему можно найти применение главным образом только при создании «миксинов» (mix in) — простых классов, позволяющих добавить ряд поведенческих свойств в другой класс. Как правило, «миксины» являются абстрактными и не поддерживают создания экземпляров. Именно поэтому разработчики таких языков как **Java**, **C#** и **Visual Basic**, понимая ценность «миксинов», разрешили множественное наследование интерфейсов, но только единичное наследование классов.

Вследствие того, что наследование часто противоречит главному техническому императиву программирования — управлению сложностью, рекомендуется применять данное отношение с осторожностью. Выбор типа отношения осуществляется из следующих соображений:

- если взаимодействие классов предполагается осуществить на основании общих данных, но не форм поведения, создайте объект, содержащий общие данные, и включите его во все эти классы;
- если несколько классов имеют общие формы поведения, но не данные, сделайте эти классы производными от общего базового класса, определяющего общие методы;
- если несколько классов имеют общие данные и формы поведения, так же сделайте эти классы производными от общего базового класса, теперь уже определяющего и общие данные, и общие методы;
- для предотвращения изменений в интерфейсе взаимодействующих классов, определите его в базовом классе и используйте наследование, для получения же полного контроля над интерфейсом — используйте включение.

3.2.5.3 Методы-члены и данные-члены класса.

При проектировании методов-членов и использовании данных в реализации класса, рекомендуется придерживаться следующих правил:

- включайте в класс как можно меньше методов и минимизируйте число разных методов, вызываемых классом. Согласно исследованию [38], число дефектов в коде класса статистически коррелирует с общим числом методов, вызываемых классом, а так же, большему числу методов в расчете на один класс соответствует большее число изъянов в программном обеспечении. Такое положение дел объясняется, прежде всего, сильным сопряжением между используемыми классами и повышением «коэффициента ветвления по выходу» (fan out);
- избегайте опосредованных вызовов методов других классов, так как такие непосредственные связи плохо поддаются контролю при модификации кода. Для контроля ситуации, рекомендуется применять так называемое «Правило Деметры (Law of Demeter)» [39] — Объект *A* может вызывать любые из собственных методов, если же он создает Объект *B*, то Объект *A* может вызывать любые методы Объекта *B*, но не следует вызывать методы объектов, возвращаемых Объектом *B*.

В целом, рекомендуется ослаблять сопряжением между классами, сводя следующие показатели к минимуму:

- число видов создаваемых объектов;
- число непосредственно вызываемых методов созданных объектов;

- число вызовов методов, принадлежащих объектам, возвращенным другими созданными объектами.

3.2.5.4 Конструкторы.

При работе с конструкторами объектов рекомендуется придерживаться следующих правил:

- по мере возможности, инициализируйте все данные-члены во всех конструкторах (один из приемов защитного программирования);
- в случае если нет веских обоснований обратного, полное копирование объектов предпочтительнее ограниченному. Одним из важных аспектов работы со сложными объектами является выбор типа их копирования: полного (deep copy — почленное копирование данных-членов объекта) или ограниченного (shallow copy) — указание или предоставление ссылки на исходный объект. Как правило, создание ограниченных копий повышает быстродействие кода, однако это становится заметным лишь при достаточном количестве копируемых объектов. Полное копирование может снижать быстродействие, но, как правило, не является «узким местом» программы в рассматриваемом контексте излагаемого материала. Таким образом, сомнительное улучшение быстродействия кода не является достаточным обоснованием закономерного повышения сложности, так как при ограниченном копировании требуется реализовать надежную инфраструктуру процесса (код подсчета ссылок, безопасного сравнения, уничтожения объектов и т. д.), которая сама по себе является источником ошибок. Иными словами, без веской на то причины, использовать ограниченное копирование не рекомендуется.

3.3 Процедурное конструирование

Под процедурой (функцией, методом) понимается контекстно-зависимый (выделенный на определенном уровне абстракции) фрагмент кода, решающий специализированную задачу.

Рассмотрим подходящие причины оформления некоторой части кода в виде метода. Прежде всего, такой причиной является **снижение сложности**. Посредством инкапсуляции (в некоторых случаях и сокрытия), формируются абстракции, которые разделяют поле сложности задачи на подобласти, информационная нагрузка в которых поддается осмыслению и управлению. Следствиями такой декомпозиции является общая устойчивость кода к ошибкам, минимизация его объема и облегчение его последующего сопровождения. Инкапсуляция кода в функцию с именем, наиболее точно передающим назначение функции, позволяет формировать в программе **промежуточные абстракции**, позволяющие свертывать второстепенные аспекты общей сложности разрабатываемого ПО, осуществлять **опосредованное документирование** кода. Другой популярной причиной реализации метода является предотвращение дублирования кода. Тем самым повышается компактность и гибкость архитектуры программы.

3.3.1 Проектирование и реализация функций

Качественной мерой проектирования на уровне функций является критерий «связности» (cohesion) [40]. Обобщенно, связность характеризует соответствие выполняемых в функции операций одной единой цели. Связность, как концепция, формируется на разных уровнях, характеризующих ее степень и фактор влияния на архитектуру ПО.

Покажем те виды (концепции) связности, которые могут быть полезны.

Функциональная связность — самый сильный и лучший вид связности; она имеет место, когда метод выполняет одну и только одну операцию. В таком случае требуется, чтобы метод соответствовал своему имени, в противном случае о связности нельзя сказать ничего определенного.

Последовательная связность (sequential cohesion) наблюдается в том случае, когда в методе последовательно выполняются разнообразные операции в определенном порядке, используя данные предыдущих этапов, и не формируют в целом единую функциональную связность. Чтобы сделать такой метод функционально связным, каждую операцию инкапсулируют в отдельный метод, которые в дальнейшем вызываются в определенном порядке. Обмен данными в этом случае производится через параметры, формируемые последовательно на каждом этапе.

Коммуникационная связность (communicational cohesion) имеет место, когда выполняемые в методе операции используют одни и те же данные и не связаны между собой иным образом. Иными словами, если каждую из операций инкапсулировать в отдельный метод функциональной связности, то такие функции могут быть вызваны в разных местах программы и общим у них будет только набор входных параметров.

Временная связность (temporal cohesion) наблюдается, когда операции объединяются в метод на том основании, что все они выполняются в момент наступления определенного события. Временную связность применяется как вынужденная мера в исключительных случаях, так как функция содержит слишком много разнообразного кода. Для устранения этой проблемы ищутся способы организации другой последовательности событий, не вынуждающей к созданию данного вида связности. Если этого достичь не возможно, требуется осуществить функциональную декомпозицию последовательности действий в методе с временной связностью.

Остальные виды связности не так полезны, однако о них следует упомянуть во избежание их использования.

Логическая связность с управлением (logical cohesion) имеет место, когда функция включает несколько операций, а выбор выполняемой операции осуществляется на основе передаваемого в метод управляющего флага. Этот вид связности называется логическим на основании лишь наличия у функции управляющей логики. Какой-нибудь другой по-настоящему объединяющей связи (подчинению одной простой цели) между операциями нет. Наряду с этим, следует отметить, что логически связный метод вполне приемлем, если его код состоит исключительно из ряда управляющих операторов и вызовов других методов. Если единственная роль метода — координация выполнения различных команд, а сам он не выполняет действий, это обычно удачное проектное решение. Такие методы называют «обработчиками событий».

Процедурная связность (procedural cohesion) имеет место, когда множество алгоритмов инкапсулировано в одну функцию из-за желания упростить архитектурное решение программы.

3.3.2 Проектирование интерфейсов функций

Согласно [41], интерфейсы между методами довольно часто являются источниками ошибок. Рассмотрим некоторые рекомендации по предотвращению подобных проблем.

Передавайте параметры в порядке выполняемых в методе операций: ввод данных, их изменение и возврат результата (входные значения — изменяемые значения — выходные значения). Хорошим подспорьем для этого может служить выработка конвенции для именования каждой группы параметров.

Если несколько методов имеют схожий набор параметров, то последовательность их передачи должна иметь согласованный порядок.

Используйте все параметры. Если параметр заявлен в интерфейсе функции, то он должен быть задействован в ее коде явным образом. Интерфейсы, содержащие параметры-заглушки, обычно являются плохим проектным решением (в частности, и для чисто абстрактных методов класса).

Передавайте переменные диагностики/статуса или кода ошибки в последнюю очередь. Они второстепенны по отношению к главной цели функции и являются исключительно выходными параметрами.

Не используйте параметры функции, отмеченные в интерфейсе как входные значения, в качестве рабочих переменных. Для этой цели используются локальные переменные с явным присвоением значения. Естественно, это требование становится обязательным при работе со ссылочными типами.

Документируйте предположения о параметрах в случае, если они должны иметь определенные характеристики. Хорошим проектным решением будет применение утверждения (assertions), позволяющие встроить предположения в код. Следует документировать следующие предположения о параметрах:

- вид параметров (входные, изменяемые или исключительно выходные);
- единицы измерения;
- смысл кодов диагностики/статуса и ошибок, если для их представления не используются специальные типы;
- диапазоны допустимых значений.

Ограничивайте число параметров метода. Одной из качественных характеристик интерфейса функции является мера сцепления (coupling).

Данная мера описывает поток данных между вызываемой функцией (функцией-сервером) и вызывающей функцией (функцией-клиентом). Сцепление может быть *неявным* (когда функции взаимодействуют через глобальные переменные) или *явным* (через параметры). Явное сцепление дает более сильную зависимость между функцией-клиентом и функцией-сервером и является предпочтительнее. При коммуникациях через параметры, функции легче понять, повторно использовать и модифицировать. Общее число параметров в интерфейсе выражает *меру сцепления*.

Таким образом, сцепление показывает, сколько усилий и времени потребуется для понимания потоков данных между функциями.

Важно отметить следующее. Для понимания потоков данных в программе, компоненты потока данных должны быть каким-либо образом описаны: как

глобальные переменные, либо определяться в области действия клиентской функции и передаваться серверной функции. В индустрии ПО много лет идут дискуссии по поводу использования глобальных переменных, однако многие участники до сих пор не уяснили суть проблемы. Считается, что потенциально любая функция может случайно (или умышленно) изменить значение глобальной переменной и в результате очень трудно будет найти источник ошибки. К этому добавляют следующее — вовсе не очевидно, какие именно функции обращаются к данной глобальной переменной и потенциально проблема может возникнуть в любом месте программы. Все сказанное верно, но суть проблемы в том, что *основной ущерб от применения глобальных переменных — это неявное сцепление. И использование глобальных переменных вынуждает изучать большие сегменты кода, чтобы понять поток данных в программе в процессе разработки и дальнейшего сопровождения.* Несмотря на все недостатки использования глобальных переменных, их использование оправдано, по крайней мере, в двух случаях. Во-первых, это производительность программы. Функции, использующие параметры, тратят время на организацию работы со стеком и копирование значений самих параметров или их адресов. Если предполагается использовать глобальные параметры для данной цели, следует продумать следующий момент: программа в самом деле сталкивается с проблемой производительности и применение глобальных переменных для этой цели действительно устранит проблему. Во-вторых, причиной применения глобальных переменных является повышение производительности разработчика. Интерфейсы потенциально являются источниками ошибок, поэтому, безопаснее и быстрее в некоторых случаях написать серверную функцию, использующую глобальные переменные, а не параметры.

Таким образом, везде, где это представляется возможным, следует *минимизировать сцепление*, уменьшив число элементов в интерфейсе функции. В процессе функциональной декомпозиции сложности задачи, очень важно не разделять операции, которые могут быть выполнены вместе. В случае неоправданного разделения почти неизбежны избыточные коммуникации между компонентами декомпозиции и дополнительное сцепление.

Для объектно-ориентированных языков, решением проблемы минимизации сцепления может казаться использование сложных типов данных в качестве параметров. В некоторых случаях это оправдано и количество параметров при этом действительно уменьшается, однако это не всегда ведет к уменьшению сцепления. В данном случае важно учитывать информационную нагрузку всего потока данных, а не формальное количество синтаксических оболочек информационного фрейма. Разделение вычислений вместо объединения их в одной функции обычно требует организовать дополнительное взаимодействие между частями кода и если такие вычисления реализуются в разных функциях, хотя могли бы совмещаться в одной, получается избыточный поток данных.

Объединение в одной функции опрометчиво разделенных операций исключает лишние коммуникации между функциями и понижает сцепление. С другой стороны, объединение в функции самых разных операций, обрабатывающих многочисленные сложные пакеты данных ведет к ухудшению качества связности. Эти две характеристики являются ортогональными и служат хорошими регуляторами качества функциональной декомпозиции задачи. Принятие решения

в пользу одной из них без учета влияния другой — пример неудачного подхода к конструированию.

Передавайте в метод те переменные или объекты, которые нужны ему для поддержания абстракции интерфейса. Для языков, поддерживающих сложные типы данных, существует два конкурирующих подхода к передаче данных в методы. Предположим, существует объект (экземпляр класса), предоставляющий доступ к данным посредством своих методов доступа, а вызываемой функции нужны не все элементы данных объекта. Следуя первому подходу, в функцию нужно передать только те параметры, которые ей нужны. Тем самым достигается минимальное сопряжение между функциями с одной стороны, с другой стороны сохраняется принцип инкапсуляции, не позволяя вызванной функции использовать все методов доступа к данным объекта в случае его передачи в качестве параметра. Согласно второму подходу, утверждается, что следует передать весь объект. В этом случае вызываемая функция получит доступ к дополнительным членам объекта, что позволит сохранить стабильность ее интерфейса. Утверждается, что именно передача конкретных элементов нарушает инкапсуляцию, так как в этом случае нарушается объектно-ориентированная направленность кода.

Ни тот ни другой подход, на самом деле, не отвечает на самый важный вопрос: какую именно абстракцию формирует интерфейс метода? Если абстракция подразумевает, что метод ожидает совокупность конкретных элементов данных, которые по независящим от функциональной декомпозиции обстоятельствам, принадлежат подмножеству данных некоторого объекта, то следует передать эти элементы по отдельности. Если же абстракция состоит в том, что элементы данных всегда принадлежат объекту конкретного класса, над которым метод должен выполнять ту или иную операцию, тогда, следует передавать эти параметры в связи с объектом. Наличие кода, «подготавливающего» данные перед вызовом функции («собирающего объект») или «разбирающего» объект после вызова, — признак неудачного проектирования метода. С другой стороны, если сигнатура интерфейса постоянно изменяется в процессе проектирования и при этом параметры относятся к одному и тому же объекту, в метод следует передавать весь объект, а не конкретные элементы.

3.4 Использование переменных

Под переменными будут пониматься именованные области памяти, хранящие фрагменты информации, которые участвуют в создании конечного результата (а так же сам результат) процесса вычисления. Под вычислением понимается процесс преобразования информации из одного представления в другое. Ограничения и возможности использования переменных в процессе вычисления обуславливаются типизацией, которая, в свою очередь, является областью ответственности конкретного языка программирования или исполнительной среды.

3.4.1 Система типов в языках программирования

Под системой типов в языках программирования понимается совокупность правил, с помощью которых различным языковым конструкциям, составляющим программу (переменные, выражения, функции и т.д.), назначается свойство, именуемое типом.

Образно говоря, система типов декларирует интерфейс обмена данными между различными компонентами программы и позволяет осуществить проверку

согласованности взаимодействия этих компонент. Такая проверка может осуществляться на стадии компиляции (статическая), во время выполнения (динамическая), а также быть комбинацией обоих видов. Таким образом, система типов назначает тип каждому вычисленному значению и затем, отслеживая последовательность этих вычислений, предпринимает попытку доказать отсутствие ошибок согласования типов. В зависимости от своей реализации, она может определять, какие именно конкретные высказывания языка приводят к ошибкам согласования типов, но главной ее целью является предотвращение ситуации, когда операции, предполагающие определённые свойства своих параметров, получили параметры, для которых эти операции не имеют смысла.

Система типов является частью реализации языков программирования и проверка ошибок согласования типов встраивается в их трансляторы. Система типов языка может быть расширена с помощью специальных инструментов, осуществляющих дополнительные проверки на основе исходного синтаксиса типизации в языке. Компилятор может оптимизировать исполняемый код на основании системы статической типизации как с точки зрения производительности (путем использования специальных инструкций микропроцессора для вычислений с определенными типами данных), так и по операциям с памятью (использование стека или общей динамической памяти процесса). Количество налагаемых ограничений на использование типов и алгоритмов их определений характеризуют типизацию языка (и в конечном итоге определяют возможности языка, а так же потенциал его развития). Конкретная система типов языка программирования строится на основании особенностей вычислительной архитектуры (как в hard, так и soft исполнении).

Операция назначения свойства последовательности бит, называемая типизацией, превращает программируемое аппаратное обеспечение в символьную систему, состоящую из аппаратного обеспечения и исполняемой программы. При *статической типизации* языковая конструкция связывается с типом в момент объявления и такая связь не может быть изменена позже. Примерами статически типизированных языков являются потомки **Algol 68**: **Pascal** — **Ada**, **C++/C#**, **Java**, а так же **ML**. Преимуществами статической типизации являются: самый простой машинный код (поэтому она удобна для языков, имеющих трансляторы, производящие исполняемые файлы для ОС или JIT-компилируемые промежуточные коды), исключение множества ошибок уже на стадии компиляции. Именно поэтому языки со статической типизацией хороши для написания сложного, и отчасти, быстрого кода. В качестве недостатков статической типизации обычно отмечают многословность языка (каждый раз надо указывать тип переменной или возвращаемого значения), неудобство работы с данными переменных типов (например, в реляционных СУБД). Хотя в некоторых языках есть автоматическое выведение типа, оно может привести к трудноуловимым ошибкам. При *динамической типизации* языковая конструкция связывается с типом в момент присваивания значения, а не в момент объявления переменной. Таким образом, в различных участках программы одна и та же переменная может принимать значения разных типов. Примеры языков с динамической типизацией — **Smalltalk**, **Python**, **Objective-C**, **Ruby**, **PHP**, **Perl**, **JavaScript**, **Lisp**, **xBase**, **Erlang**. В качестве преимуществ динамической типизации отмечают сравнительную простоту работы прикладного программиста с СУБД, которые принципиально возвращают информацию в «динамически типизированном» виде. Также

динамические языки зарекомендовали себя с лучшей стороны при программировании веб-служб и создании скриптов. К основными недостатками динамической типизации относят: сложности поиска ошибок (для данной типизации требуется, как минимум, выполнить инспектируемый участок кода), дополнительные накладные расходы при сопровождении сложных систем (развитая статическая система типов, например, по Хиндли-Милнеру [42] занимает существенное место в самодокументировании программы, в то время как динамическая типизация по определению не проявляет этого свойства), низкая скорость работы кода и большие расходы памяти (связанные с динамической проверкой типа), непредсказуемость результатов операций (в языках с динамической перегрузкой операций в зависимости от типа возможны неожиданные результаты, если вдруг операнд окажется не того типа). К тому же большинство языков с динамической типизацией интерпретируемые, а не компилируемые.

Статические системы типов могут требовать провозглашения типа до использования переменной, а могут и не требовать. Такие системы (развитые *системы типов Хиндли — Милнера*, применяемые в языках **ML**, **Standard ML**, **OCaml**, **Scala**, **Haskell**, **F#**, **Fortress** и **Boo**), осуществляют выведение типов: компилятор выстраивает заключение о типах переменных с помощью алгоритма Хиндли — Милнера вывода типов выражений на основании того, как в коде используются эти переменные. В дальнейшем эта возможность появилась в качестве front-end и в объектно-ориентированных языках программирования: **C#**, **D**, **Visual Basic.NET**, **C++11**, **Vala**, **Go**. Здесь она ограничивается синтаксической возможностью опустить тип идентификатора в определении с инициализацией. Вообще говоря, построение алгоритма выведения типов возможно, если эта проблема принципиально разрешима в данной теории типов. С практической точки зрения, создатели языка программирования принимают некоторые компромиссные решения для выбранной системы типов, чтобы построение такого алгоритма было возможно. Например, система типов языка **Haskell**, являющаяся разновидностью системы типов Хиндли — Милнера, представляет собой ограничение системы Fω для полиморфных типов первого ранга, на которых выведение типов разрешимо.

Еще одной характеристикой системы типов языка является *сильная или слабая типизированность* (англ. strongly and weakly typed). Если спецификация языка требует, чтобы правила типизации исполнялись строго (допускаются лишь те автоматические преобразования типов, которые не теряют информацию), то такой язык называется *сильно/строго типизированным*, в противном случае — *слабо типизированным*. В [43], система типов называется «сильной», если она исключает возможность возникновения ошибки согласования типов времени выполнения. Сильная или слабая типизация является следствием множества компромиссов, принятых при разработке системы типов языка наряду с такими ее характеристиками, как: наличием или отсутствием типобезопасности (англ. type safety), безопасностью памяти и проверки согласования типов, статической или динамической природы типизации. Примерами слабых систем типов являются системы, принятые для языков **C, C++**. Их характеризуют понятия приведения типов и «каламбуры» типизации. Все эти механизмы поддерживаются на уровне компилятора и часто используются неявно. Так, например, в **C++** используются операции, которые могут представить элемент данных любого типа как принадлежащий любому другому типу при условии равенства длины их битового

представления. Таким образом, имеется возможность изменить состояние данных таким способом, который был бы недопустимым для исходного типа. В противоположность этому, в языках, типизированных по Хиндли - Милнеру, понятие о приведении типов отсутствует в принципе. Единственным способом «преобразования» типа является возможность алгоритмически построить значение требуемого типа на основе значения исходного типа. Тем не менее, система типов Хиндли - Милнера обеспечивает чрезвычайно высокий показатель повторного использования кода за счёт параметрического полиморфизма. В то время как сильная, но не полиморфная система типов может затруднить решение многих алгоритмических задач, как это было справедливо отмечено в отношении языка **Pascal** [44]. В теории программирования сильная типизация выступает непременным элементом обеспечения надёжности разрабатываемых программных систем. При правильном применении (когда в программе объявляются и используются отдельные типы данных для логически несовместимых значений) она защищает код от простых, но труднодиагностируемых ошибок, связанных с совместным использованием логически несовместимых значений. Подобные ошибки выявляются компилятором, тогда как при возможности неявного приведения практически любых типов друг к другу эти ошибки можно выявить только при тестировании.

Вообще говоря, язык программирования может и не использовать какую-то ни было систему типов, то есть являться бестиповым. В этом случае имеется возможность осуществлять любую операцию над любыми данными, которые представляются цепочками бит произвольной длины [45] (в противоположность типизированным, где спецификация каждой операции определяет типы данных, к которым эта операция может применяться, подразумевая её неприменимость к иным типам [45]). Бестиповыми является большинство языков ассемблера. Имеются и примеры высокоуровневых языков: **BCPL**, **BLISS**, **Forth**, **Рефал**. Подобное обращение с данными возможно и для типизированных языков, а именно, в особом случае — однотиповых языках (англ. single-type language). Обычно это языки сценариев или разметки, такие как **REXX** и **SGML**, где единственным типом данных является символьная строка, используемая для представления как символьных, так и числовых данных.

Строго говоря, лишь некоторые языки могут считаться типизированными с точки зрения теории типов (разрешая или отвергая все операции), большинство коммерческих языков предлагают лишь некий компромисс типизированности [45]. Из них многие имеют возможности обойти или нарушить свою систему типов, поступаясь типобезопасностью ради более человекозависимого контроля над исполнением программы.

3.4.1.1 Типобезопасность

В информатике типобезопасность характеризует степень, с которой язык программирования предотвращает ошибки согласования типов (ошибочное или нежелательное поведение программы по причине несогласованности типов компонентов программы).

Формальное определение типобезопасности в теории типов намного строже, чем то, что под этим понимается в практическом программировании. Так, ведущие эксперты по теориям и системам типов (Хоар, Карри, Хиндли, Милнер и др.) используют понятие типобезопасности только в том случае, когда высказывания

языка в принципе не допускают возможности такого поведения программы, которое могло бы привести к нарушению целостности данных, повреждению runtime-системы языка и аварийному завершению. Как правило, разработчики языков менее требовательны, и обычно называют «безопасными» любые операции, непосредственно не приводящие к краху программы, сужают понятия типобезопасности до «безопасности типов в отношении доступа к памяти» (memory type safety), означающее, что компоненты объектов одного агрегатного типа не могут обращаться к ячейкам памяти, выделенным под объекты другого типа. Иногда, с чисто практической точки зрения, типобезопасность рассматривается как свойство конкретной программы, а не языка, на котором она написана, так как некоторые языки предоставляют типобезопасные возможности, но разрешают их нарушать. Итак, в наиболее строгом смысле, язык классифицируется как типобезопасный, если он допускает лишь те операции над данными, что разрешены типами этих данных [46]. Или (в контексте денотационной семантики) [Принятие термина «денотационный» предполагает использование приема описания объектов при котором, всем правильным конструкциям языка даются обозначения, называемые «денотатами». Денотаты рассматриваются как идеализированные математические объекты для моделирования других объектов.] типобезопасность означает, что любое выражение, прошедшее проверку типов, и имеющее некий тип τ, является истинным элементом некоторого множества всех значений τ.

С практической точки зрения, определение типобезопасности представляется как соблюдение двух свойств семантики языка программирования [47]:

- сохраняемость типов («хорошая типизируемость»): типизируемость (прохождение проверки согласования типов) программы инвариантно сохраняется в рамках правил преобразования языка (стратегии вычисления или редукции типа);
- продолжаемость («незастреваемость»): прошедшая проверку типов программа никогда не войдёт в неопределённое состояние, при котором никакие дальнейшие операции не возможны.

В этом же контексте, термин «строгая типизация» применяется для обозначения наиболее очевидных (необходимых) аспектов типобезопасности. Поэтому языки со статической проверкой согласования типов часто описываются как «строго типизированные», так как статически предотвращают преобразования между значениями несовместимых типов. Даже **C++** идентифицирует себя как «строго типизированный», несмотря на сложную (и всячески поощряемую к применению) систему правил неявного (автоматического) приведения типов, что, естественно, не может классифицироваться как типобезопасность. Языки, осуществляющие динамическую проверку согласования типов, могут так же классифицироваться как «строго типизированные», поскольку программа, пытающаяся преобразовать значение к несовместимому типу, породит исключение (свойство продолжаемости).

Для статических (времени компиляции) систем типов, типобезопасность гарантирует, что вычисленное значение любого выражения будет допустимо статическим типом этого выражения. Более того, при разветвленной статической системе типов предотвращаются логические ошибки, проистекающие из семантики различных типов. Например, и миллиметры, и дюймы могут представляться

целыми числами, но будет ошибкой вычитать дюймы из миллиметров, и развитая система типов не допустит этого.

Безопасность по типам тесно связана с безопасностью доступа к памяти — предотвращение возможности копирования произвольной цепочки бит из одной области памяти в другую. Если язык предусматривает некоторый тип, имеющий ограниченный спектр допустимых значений, и предоставляет возможность копирования нетипизированных данных в переменную этого типа, то этот язык не является типобезопасным. Например, запись произвольного целого значения в переменную, имеющую тип «указатель». Также предотвращение возможности переполнения буфера памяти — записи в ячейки памяти, не относящиеся к данному контексту и хранящие значения других объектов.

3.4.1.2 Примеры типобезопасных языков

Ada (семейство **Pascal**) ориентирован на задачи системного программирования. Считается самым типобезопасным языком семейства и, тем не менее, предоставляет ряд небезопасных конструкций, имена которых обычно начинаются с Unchecked_. В свое время Хоар [48] замечал, что для обеспечения надёжности одних статических проверок недостаточно и надёжность, в первую очередь, является следствием простоты. В частности, в отношении **Ada** он предсказывал, что сложность этого языка станет возможной причиной техногенных катастроф. Авария космической ракеты Ариан-5 в 1996 году подтвердила его предсказания. Причиной аварии послужила излишняя сложность спецификаций повторно используемых компонентов, которая не позволила в полной мере проконтролировать диапазон возможных значений переменных в кодах обработчиков исключений, окружающих умышленно небезопасные операции приведения типов.

BitC представляет собой гибридный язык, комбинирующий низкоуровневые возможности **C**, безопасность **Standard ML** и лаконичность **Scheme**. Язык ориентирован на задачи системного программирования.

Common Lisp — мультипарадигменный язык программирования общего назначения, считается типобезопасным языком за счет возможности компилятора размещать динамические проверки для операций, безопасность которых не может быть доказана статически. И, тем не менее, существует возможность установить для компилятора сниженный уровень динамической типобезопасности, и скомпилированный таким образом код уже не может считаться безопасным.

Cyclone является безопасным диалектом языка **C**, заимствующим многие идеи типобезопасности из **ML**, включая типобезопасный параметрический полиморфизм. В силу небезопасной природы **C**, портирование чистого **C**-кода требует дополнительной работы, хотя значительная её часть может быть проделана в рамках самого **C** еще до смены компилятора. Поэтому **Cyclone** часто определяют не как диалект **C**, а как язык, синтаксически и семантически похожий на **C**.

Haskell — стандартизированный чистый функциональный полнотиповый (обеспечивается автоматической проверкой согласования типов) язык программирования общего назначения. Наличие полноты типов делает поведение программ предсказуемым (надежным). Тем не менее, в угоду модным тенденциям (практике низкой культуры программирования) в стандарт языка были введены небезопасные операции, отмеченные идентификаторами, начинающимися с unsafe. Также в стандарт была включена реализация механизма исключений, которая

способна приводить к аварийному завершению программ. В итоге, по мнению некоторых специалистов [49], язык приобрел репутацию «исключительно небезопасного».

ML — язык, который изначально разрабатывался в качестве интерактивной системы доказательства теорем. Вследствие дальнейшего развития стал самостоятельным компилируемым языком общего назначения. Исключительным результатом такой направленности явилось мнение, что многие потомки **ML** априори считаются статически типобезопасными, несмотря на то, что некоторые из них содержат небезопасные возможности явным образом (**Haskell**), либо откладывают некоторые значимые проверки на этап выполнения (**OCaml**). В поддержку впечатления типобезопасности для языков семейства **ML** так же работает развитая поддержка алгебраических типов данных, использование которых существенно способствует предотвращению логических ошибок. Следует отметить, что языки семейства **ML** чаще являются инструментами интерактивного доказательства (доказательства корректности программ на других языках) и не используются для непосредственной разработки программ в силу высокой трудоёмкость использования и низкого быстродействия.

SML (**Standard ML**) — язык семейства **ML**, ориентированный на разработку крупномасштабных программ промышленного быстродействия. Вследствие своего происхождения, имеет строгое формальное определение и его полная типобезопасность математически доказана [50]. После статической проверки согласованности интерфейсов компонентов программы (в том числе и порождаемых), дальнейший контроль безопасности поддерживается runtime-системой языка. В результате, содержащая ошибку программа не может завершиться аварийно (возможное поведение в таких случаях: бесконечное вычисление или выдача сообщения об ошибке). Тем не менее, некоторые реализации (**SML/NJ, Mythryl, MLton**) включают нестандартные библиотеки, предоставляющие небезопасные операции для реализации внешнеязыкового интерфейса, обеспечивающего взаимодействие с не-**ML**-кодом (обычно это код на **C**, реализующий критичные по скорости компоненты программ).

3.4.1.3 Нарушения типобезопасности

Для обозначения различных техник нарушения или обмана системы типов некоторого языка программирования, имеющих эффект, который было бы затруднительно или невозможно обеспечить в рамках формализации языка, в информатике имеется специальный термин — «каламбур» типизации (type punning). Некоторые языки предоставляют явные возможности нарушения типизации, а культура их использования всячески поощряет их применение (**C** и **C++**), другие могут использовать для интерпретации конкретного типа данных более чем один способ, или даже не предусмотренный языком способ (**Pascal**). Так или иначе, многие промышленные языки поступаются типобезопасностью ради более точного (ручного) контроля над исполнением программы. Традиционно возможность построения «каламбура» типизации связывается со слабой типизацией, но и некоторые сильно типизированные языки или их реализации предоставляют такие механизмы.

3.4.1.4 Полиморфизм

Полиморфизм является фундаментальным свойством системы типов. В теории типов и языках программирования полиморфизмом называется способность функции обрабатывать данные разных типов [51]. Существует несколько видов полиморфизма. Наиболее различные (исторически ранее описанные) — специальный полиморфизм (или «ad hoc полиморфизм») и параметрический полиморфизм (вкратце, первый из них описывается принципом «много реализаций с похожими интерфейсами», а второй — «одна реализация с обобщённым интерфейсом»). С учетом свойства полиморфизма, различают следующие системы типов [52]: статическая неполиморфная типизация (потомки **Algol** и **BCPL**), статическая полиморфная типизация (потомки **ML**), динамическая типизация (потомки **Lisp**, **Smalltalk**, **APL**).

Следует отметить, что параметрический полиморфизм и динамическая типизация более существенно повышают коэффициент повторного использования кода (определенная единственный раз, функция реализует без дублирования заданное поведение для бесконечного множества вновь определяемых типов, удовлетворяющих требуемым в функции условиям), чем специальный полиморфизм. В том случае, когда возникает необходимость обеспечить различное поведение функции в зависимости от типа параметра, используется специальный полиморфизм.

В различных парадигмах программирования под термином «полиморфизм» понимаются различные его виды. В частности, в функциональном программировании повсеместно используется параметрический полиморфизм, в объектно-ориентированной парадигме — ad hoc полиморфизм методов классов, связанных в иерархию наследования; а использование параметрического полиморфизма называют обобщённым программированием, или «статическим полиморфизмом».

3.4.1.5 Виды полиморфизма

Функция называется *мономорфной*, если параметру функции сопоставлен ровно один тип. В противном случае она называется полиморфной.

Однако некоторые языки программирования предоставляют такой синтаксический механизм, который позволяет назначение единого имени (идентификатора) нескольким мономорфным функциям. В этом случае, в исходном коде становится возможным осуществлять вызов функции с фактическими параметрами разных типов, но в бинарном представлении фактически происходит вызов различных функций. Такая возможность была названа **ad hoc полиморфизмом** [51] (в русской литературе используется термин **«специальный полиморфизм»**). Термин «ad hoc» (лат. спонтанный, случайный) был выбран специально, чтобы подчеркнуть, что функции являются на самом деле мономорфными, то есть реализующими частные варианты поведения для каждого конкретного типа, и таким образом, возможность обрабатывать данные разных типов в этом случае эмулируется языком на уровне front-end, а не является фундаментальным свойством его системы типов.

Исторически, для упрощения процесса трансляции существовало ограничение на использование более одной процедуры с одним и тем же именем в одной области видимости. С развитием языков вводится механизм перегрузки

(разновидность статического полиморфизма) подпрограмм с разной сигнатурой. Такая возможность повышала гибкость использования кода в рамках процедурной парадигмы без применения объектно-ориентированного программирования. Контроль использования одноимённых процедур происходит на уровне транслятора и абсолютно прозрачен для программиста (естественно, на уровне грамматики языка, контроль достигается использованием специальных для этого случая ключевых слов). Таким образом, использование этого подхода совершенно естественно для статически типизированных языков, где имеется возможность проверки типов аргументов при вызове функции. Естественно, перегрузку функций следует отличать от форм полиморфизма, где правильный метод выбирается во время выполнения, например, посредством механизма виртуальных функций. Одним из положительных аспектов механизма перегрузки является производительность, так как синтаксический анализ уже произведен и вызовы требуемых функций напрямую прописан в бинарном образе (не производится анализ типов во время выполнения программы). Расплачиваться за производительность приходится, прежде всего, размером бинарного образа. Даже после построения безопасной параметрически полиморфной типизации (для развитых систем типов по Хиндли-Милнеру), вскоре обнаружилась необходимость (в некоторых случаях) определения специальных реализаций функций для разных типов. Таким образом, специальный полиморфизм поддерживается до сих пор во многих языках посредством перегрузки функций.

Функция называется *полиморфной*, если ее параметру сопоставлено более одного типа. Ясно, что с каждым фактическим значением связывается всегда один тип, но полиморфная функция рассматривает параметр, прежде всего, на основе его поведения. В частности, одно и то же алгоритмическое действие выполняется не с представлением параметра в памяти, а, например, со взаимной организацией компонентов структурного типа. [51] назвал такую возможность **параметрическим полиморфизмом**. Сама суть концепции параметричности заключается в том, что функция использует аргументы на основе поведения, а не значения (образа в памяти), обращаясь лишь к необходимым свойствам аргументов, что делает её применимой в любом контексте, где тип аргумента удовлетворяет заданным требованиям поведения. Сильно типизированные языки предусматривают возможность описания (эмуляции) параметрически полиморфных типов и позволяют производить статический контроль безопасности использования полиморфных функций. В данном случае, полиморфный тип синтаксически записывается посредством переменной универсального типа (контекстного идентификатора), конкретизирующегося далее в программе. Развитые системы типов (по Хиндли - Милнеру [53, 54]) располагают механизмами для определения полиморфных типов, что делает использование полиморфных функций более естественным, обеспечивая при этом статическую типобезопасность. Классическим примером использования полиморфного типа служит обработка списка элементов произвольного типа. Для такого рода задач во многих статически типизируемых функциональных языках программирования представляются специальные синтаксические конструкции (синтаксический сахар). Параметрический полиморфизм доступен и для некоторых императивных (в частности, объектно-ориентированных) языков программирования, где для его обозначения обычно используется термин «обобщённое программирование». Однако реализация этой концепции может достигаться разными механизмами,

призванными обойти несовершенство системы типов. Например, в **C++** полиморфное поведение обеспечивается самым небезопасным образом — посредством явной передачи необходимых свойств типа через безтиповое подмножества языка. В других подобных языках в результате компиляции порождается такое количество перегруженных мономорфных функций, с каким количеством типов данных функция вызывается в программе, хотя определение функции на уровне исходного кода по-прежнему единственно. Другими словами, параметрический полиморфизм на уровне исходного кода подменяется в ad hoc полиморфизм на уровне целевой платформы. Такой механизм подмены называется мономорфизацией (monomorphizing). Все эти механизмы являются front-end надстройкой над мономорфной системой типов языка. Более развитые системы типов допускают опциональное использование мономорфизации, то есть предоставляют выбор между сохранением единственности тела полиморфной функции и размножением мономорфных тел. В таких системах типов механизм реализации параметрического полиморфизма основывается на «обёрнутом» представление объектов и эффективно оптимизирующие компиляторы могут выполнять полную мономорфизацию программы. Стоит отметить, что мономорфизация позволяет нейтрализовать негативное воздействие параметрического полиморфизма на скорость работы программы, естественно, увеличивая размер выходного машинного кода (причем существенно).

Со временем, развитие языков и систем типов выявило большее разнообразие способов абстрагирования, и их классификация была расширена [55].

В частности, если поведение полиморфной функции ограничивается множеством типов, связанных в иерархию «супертип — подтип», то оно квалифицируется как **полиморфизм включения (или полиморфизм подтипов).** Данный подтип полиморфизма тесно связан с объектно-ориентированным программированием, где связь подтипов с параметрическим полиморфизмом обеспечивается посредством связанной квантификации и ковариантности/контравариантности (или полярности) конструкторов типов. Для данной парадигмы программирования полиморфизм подтипов представляет собой концепцию в теории типов, предполагающую использование единого идентификатора при обращении к объектам некоторого множества классов, выстроенного на основе иерархии с одним базовым суперклассом. Для построения такой иерархии используется механизм наследования. В этом контексте, реализация полиморфизма основывается на так называемой «виртуальной таблице» функций, реализующих полиморфную часть интерфейса класса (каждый экземпляр класса содержит указатель на виртуальную таблицу своего класса, на основе которой производится вызов соответствующего полиморфного метода). Данная модель используется в случаях позднего связывания и одиночной диспетчеризации (одноаргументный полиморфизм — виртуальные функции связываются посредством простого просмотра виртуальной таблицы по первому аргументу, а типы остальных аргументов не учитываются). Существуют и более сложные модели наследования [56] в которых возможна множественная диспетчеризация.

Примером другого подтипа полиморфизма является **политипизм** или **обобщённость типа данных**. Данный подтип поддерживается полиморфизмом конструкторов типов и предназначен для унификации программных интерфейсов. Одним из примеров политипизма может выступать обобщённый алгоритм сопоставления с образцом. Можно сказать, что в некотором смысле политиповая

функция является более обобщённой, чем полиморфная (политиповая функция может быть полиморфной, хотя обратное не справедливо). Для некоторых систем типов политипическое поведение может эмулироваться посредством использования универсального типа данных или параметрическим полиморфизмом.

Еще один подтип полиморфизма, который хотелось бы отметить, реализуется в языках с неявной (латентной или утиной (duck typing)) типизацией — вид динамической типизации, применяемой в таких языках программирования как: **Perl, Smalltalk, Python, Objective-C, Ruby, JavaScript, Groovy, ColdFusion, Boo, Lua, Go, C#**. Для данной типизации область использования объекта определяются его текущим набором методов и свойств, а не наследованием от определённого класса. Иными словами, если объект содержит все методы некоторого интерфейса, то он и реализует данный интерфейс независимо от связей в иерархии наследования. **Утиная типизация** преодолевает такие проблемы иерархической типизации, как невозможность указать путём наследования на совместимость интерфейса со всеми настоящими и будущими интерфейсами, с которыми он идейно совместим и экспоненциальное увеличение числа связей в иерархии типов при хотя бы частичной попытке этого добиться.

3.4.1.6 Влияние системы типов на стиль программирования

Выбор той или иной системы типов (языка программирования, поддерживающего данную систему типов) основывается, прежде всего, на потребностях решаемой задачи и в последнюю очередь на предпочтениях, привлекаемых для ее решения специалистов.

Статически типизированные языки предотвращают ошибки согласования типов на этапе компиляции и, в некоторых случаях, могут порождать более эффективно исполняемый код. Динамическая типизация, в свою очередь, лучше подходит для быстрого прототипирования, а ошибки согласования типов могут составлять лишь малую часть возможных ошибок в программах [57, 58]. С другой стороны, в статически типизированных языках может отсутствовать потребность явной декларации типов (для языков с поддержкой вывода типов), а некоторые динамически типизированные языки производят оптимизацию на этапе выполнения программы [59, 60], которую, частично, проводят посредством применения механизмов вывода типов [61].

В контексте рассматриваемой проблемы, естественным выбором будет использование статической системы типов. Данный выбор основывается на доступных возможностях по предотвращению ошибок и эффективности исполняемого кода. Далее, приведем ряд рекомендаций по обращению с переменными типов для языков со статической типизацией не обладающих полноценным механизмом вывода типов.

3.4.2 Объявление и инициализация переменных

Одна из самых опасных возможностей языка — это неявное объявление переменных. Можно сказать, что языки, требующие явного объявления переменных, заставляют более внимательно относится к их использованию и это бесспорное достоинство таких языков. Если язык поддерживает неявную типизацию (на уровне синтаксического сахара), то рекомендуется придерживаться следующих рекомендаций:

- отключить возможность неявного объявления на уровне компилятора;
- объявлять все переменные, по возможности придерживаясь конвенции наименования;
- проверить использующиеся переменные с помощью инструментальных средств среды разработки — список перекрестных ссылок, UML-диаграммы классов, сообщения компилятора.

Одним из возможных источников ошибок в программировании является процесс инициализации переменных. При неверной инициализации в ходе вычислений могут использоваться самые неожиданные величины по следующим причинам:

- переменной не было присвоено значение прежде ее использования и компилятор не имеет возможности инициализировать переменную (или такие возможности компилятора не были использованы). В результате переменная имеет то значение, которое будет присвоено ей по прихоти ОС — случайное;
- значение переменной устарело для конкретного контекста кода;
- частичное присвоение значения переменной.

Организация грамотного процесса инициализации осуществляется следующим образом:

- применяя Принцип Близости (группировка связных действий в одном месте) [6] к процессу инициализации, получим такую рекомендацию — переменная обязательно инициализируется в том месте, где она используется в первый раз. Требования можно усилить и сформулировать в следующем виде — переменную следует объявлять и инициализировать в том месте, где она используется в первый раз (понятно, что речь идет о локальных переменных). Возможно, такая рекомендация может показаться слишком строгой и разрушающей стиль оформления кода, принятый в процедурно-модульной парадигме, однако соблюдение этого правила делает код более устойчивым в перспективе его сопровождения;
- если компилятор поддерживает возможность автоматической инициализации переменных — используйте эту возможность;
- если значение не планируется изменять, используйте константы вместо переменных;
- используйте конструкторы класса по назначению (для инициализации);
- снабжайте функции механизмом проверки корректности своих входных параметров.

3.4.3 Локализация переменных

Код, расположенный между двумя ближайшими обращениями к переменной называется «окном уязвимости» [6] для этой переменной. Чем шире это окно, тем больше вероятность того, что значение этой переменной может быть непреднамеренно изменено. Таким образом, обращения к переменной целесообразно локализовать путем их группировки. Количественным критерием сгруппированности обращений является определения «интервала» (span) и «время жизни» переменной — число строк, на протяжении которых она используется. Как интервал, так и время жизни переменной желательно сократить для уменьшения

неверного или неумышленного изменения значения, а также для улучшения сопровождаемости кода (отслеживание обращений к переменной). Область видимости переменных можно минимизировать следующим образом:

- переменные, используемые в цикле, инициализируются непосредственно перед циклом, особенно, если используется конструкция вложенных циклов;
- не присваивайте переменной значение вплоть до ее использования;
- группируйте связанные (по использованию переменных) команды;
- инкапсулируйте группы связанных команд в методы. Здесь стоит помнить о том, что использование небольших функций (с малым временем жизни переменных и меньшим интервалом обращений) предпочтительнее обширных методов.

По возможности, следует минимизировать область видимости переменных везде, где это возможно. Глобальные переменные превращать в локальные, защищенные данные-члены класса — в закрытые. Управление областью видимости переменных базируется на двух ортогональных подходах к созданию кода: первый из них основывается на соображениях удобства/скорости кодирования — максимальное использование глобальных переменных (в этом случае отсутствует необходимость создавать списки параметров и заботиться о правилах области видимости), второй использует интеллектуальное управление кодом — максимальное сокрытие и управление кодом через открытые сокеты (параметры функций, свойства классов). В первом случае достигается максимальная скорость написания кода, во втором — удобство сопровождения и устойчивость кода к ошибкам. В контексте рассматриваемой проблемы, второй подход выглядит предпочтительнее.

Еще одним важным моментов в вопросе инициализации переменных является «время связывания» — момент, когда переменная получает конкретные данные [62]. Связь может происходить в момент написания кода (присваивание числа), при компиляции (присваивание переменной, константы), во время загрузки (инициализация из реестра Windows), во время работы программы (интерфейс, БД, файловый ввод). Чем раньше происходит связывание, тем проще код, но ниже его гибкость и устойчивость к ошибкам. С другой стороны, неоправданная гибкость в конечном итоге приводит к высоким накладным расходам на поддержку кода. Тем не менее, из соображения удобства сопровождения кода желательно осуществлять связывание как можно позднее.

3.4.4 Именование переменных

Эффективное решение проблемы именования переменных является важным аспектом качественного программирования. Известно, что в первую очередь, основной трудностью при сопровождении кода является понимание того, как конкретно этим кодом решается поставленная задача и не последнюю роль в создании этой трудности играет неряшливое и бессистемное именование.

Принцип, лежащий в основе наименования, заключается в том, что имя должно полно и точно описывать сущность, представляемую переменной. В отношении точности можно заметить, что имя должно описывать проблему (давать характеристику — *что* это такое), а не ее решение (аспект действия — *как*). Для полноты описания следует использовать столько символов, сколько необходимо, но не больше, чем нужно. Иными словами, если для описания достаточно

короткого имени, то дополнять его подробностями не нужно, с другой стороны, если короткое имя не достаточно характерно, следует расширять его до тех пор, пока проблема не будет полностью описана.

В частности, для переменных с большим временем жизни и интервалом (широкая область видимости) рекомендуется использовать развернутые наименования (дополнять спецификаторами, например, пространство имен), в то время как переменные с ограниченной областью видимости могут иметь сокращенное название [63].

При именовании служебных переменных (счетчиков циклов, временных переменных, переменных статуса и тому подобное) нужно учитывать их специфику. В частности, для индексаторов циклов и перечисляемых типов не обязательно следовать традиции именования счетчиков, восходящие к стилистике наименований языков ассемблера или первых языков высокого уровня (**Fortran**). Использование более выразительного имени поможет быстрее разобраться в специфике задачи (особенно в конструкции вложенных циклов). В отношении временных переменных можно сказать следующее. Как правило, отношение к таким переменным очень поверхностное, что находит свое отражение в их наименовании, однако следует помнить, что наличие таких переменных говорит о не продуманном алгоритме. Часто выясняется, что временные переменные на самом деле таковыми не являются, а ее роль в алгоритме понята не верно. В любом случае, следует избавляться от таких переменных.

Проблему наименования можно снять использованием различных конвенций — стандартных языковых или своих собственных (в таком случае ее нужно описать). Однако жесткое следование правилам не всегда эффективно из-за меняющейся специфики решаемых задач. Тем не менее, конвенции как стандарт могут успешно использоваться в следующих ситуациях:

- принимается одно общее решение вместо нескольких более специфичных при коллективной разработке;
- стандартизация при работе над множеством проектов;
- способствует пониманию поддерживаемого кода (в том числе и сторонними специалистами);
- прикладная область имеет специфическую терминологию, от которой желательно абстрагироваться.

Так или иначе, удачные конвенции специфицируют следующие отношения: различия между именами переменных и именами функций, классами и объектами, глобальными переменными и переменными-членами класса. Они так же содержат соглашения по проблеме форматирования имен.

3.4.5 Использование стандартных типов данных в императивных языках

3.4.5.1 Использование чисел

Общие рекомендации по использованию числовых значений в коде программы сводятся к следующему:

- не использовать значение числа в явном виде (кроме операции инициализации). Вместо этого, для оперирования со значениями целесообразно применять именованные константы и переменные.

Следование этой рекомендации позволит сделать код более читабельным и в дальнейшем упростить процедуру рефакторинга;

- используйте явную нотацию языка для операции преобразования типов во избежание «каламбура типизации»;
- избегайте сравнения разных типов (даже если компилятор позволяет неявное приведение).

3.4.5.2 Целые числа

Поскольку операции деления не является алгебраической операцией на множестве целых чисел, следует обращать особое внимание на ее применение. Конкретный компилятор может иметь свое представление о результате операции деления на данном множестве, отличное от ожидаемого программистом. Необходимо отслеживать результат данной операции, в том числе и для промежуточных операндов выражений.

3.4.5.3 Числа с разделителем целой и дробной частей

Основной сложностью, возникающей при операциях с вещественными числами, является предельная точность представления этого числа в машинной форме. Поскольку такая точность ограничена стандартом представления (например, IEC 60559:1989/IEEE 754 — IEEE Standard Binary Floating-Point Arithmetic [64]), последний разряд числа может содержать ошибку округления или случайное значение (вычислительный мусор). В связи с этим фактом, для операций с вещественными числами имеются следующие рекомендации:

- осуществлять алгоритмический контроль результатов операций таким образом, что бы значимые разряды (с точки зрения конкретного алгоритма) результата не совпадали с ошибочными (осуществлять округление, масштабирование, выравнивание);
- избегать операций прямого сравнения. Для сравнения вещественных чисел между собой или константой используется алгоритмическая малая величина, значение которой, позволяет игнорировать последние ошибочные разряды.

Вообще говоря, при использовании вещественного числа его значение может быть неоднозначным. То есть, при преобразовании исходного вещественного числа в некоторое представление (представление может быть связано с алгебраическими преобразованиями), а затем обратного преобразования полученного результата к исходному числу, результаты могут не совпадать. Неоднозначность возникает из-за потери или изменения одного или нескольких значимых десятичных разрядов в процессе преобразования.

3.4.5.4 Символьные и строковые значения

В целом, рекомендации по использованию литеральных символов и строк схожи с общими рекомендациями по использованию числовых значений. В частности, использование в алгоритме явного содержимого строки затрудняет рефакторинг кода. Операции со строками желательно проводить библиотечными инструментами для целевой платформы разработки или конкретного языка, что гарантирует избежание ошибок при работе с данным типом (копирование по

значению или по ссылке, перебор элементов по индексу или адресу, выделение памяти и ее освобождение).

Есть и некоторые отличия. Прежде всего, следует рассматривать данные значения как ресурс и предусмотреть соответствующие механизмы работы с ним. Такой подход решит следующие проблемы: интернационализации/локализации на разных этапах жизни программы, выбор кодового набора.

Далее, стоит помнить, что использование значений данного типа требует значительно больше ресурсов вычислительной системы, чем числовых типов данных. В случае, когда требуется оптимизация расчетного кода по времени выполнения или использования памяти, от литеральных строк следует отказаться. В свою очередь, использование стандартных операций с литеральными строками позволяет избежать некоторых проблем, возникающих при работе с вещественными числами. В частности, заменив числовое представление литеральным и оперируя в дальнейшем таким представлением, можно реализовать точные алгоритмы сравнения, сортировки и выборки, а также избежать потери точности при смене целевой платформы.

3.4.5.5 Индексируемые типы данных

Прежде всего, стоит помнить, что индексируемые типы данных являются самыми уязвимыми структурируемыми типами, и ситуация еще больше усугубляется возможностями, которые предоставляют некоторые языки программирования, создавать такие типы в процессе работы программы. В частности, согласно проведенным исследованиям [65] при замене таких типов данных контейнерами с четко определенным механизмом доступа к элементам (доступ по ключу, стек, очередь) код становится более надежным (хотя менее производительным). Бесконтрольный произвольный доступ к элементам индексируемых типов имеет те же последствия, что и применение оператора безусловного перехода.

В связи с этим, при использовании таких типов данных требуется соблюдать следующие рекомендации:

- убедитесь, что все значения индексов не выходят за границы допустимых значений. Хорошо, когда компилятор в таком случае осуществляет аварийную остановку работы программы, в противном случае будет достаточно тяжело локализовать ошибку. Что бы предотвращать подобные ситуации, желательно в качестве индексов использовать относительную адресацию, а не числовое значение. В качестве базы можно использовать величину размера массива;
- если логика программы допускает преимущественно последовательный доступ к индексированному типу — следует отказаться от его применения в пользу контейнеров последовательного доступа;
- для многомерных индексируемых типов или при переборе элементов во вложенных циклах возможны ошибки типа «index cross-talk», когда смешивается порядок следования индексов. В этой ситуации поможет наименование индексов более значимыми именами, чем это принято в математике.

3.4.5.6 Указатели

В контексте рассматриваемой проблемы, использование такого типа данных не рекомендуется. Иными словами, если специфика решаемой задачи не вынуждает к обратному, лучше использовать технологию, не основанную на указателях. Программирование с использованием указателей — потенциально один из самых опасных подходов к работе с памятью. На этапе поддержки кода, издержки, связанные с ручным менеджментом памяти могут существенно вырасти как за счет человеческого фактора (указатели вынуждают разбираться в коде, а не читать его), так и за счет технологического — код становится машинно-зависимым. Тем не менее, в случаях, когда продукт предназначается для конкретной технологической платформы и требуется использовать ее преимущества по максимуму, решения, построенные вручную, оказываются достаточно (с учетом накладных расходов на сопровождение) эффективными.

Основным источником труднодиагностируемых ошибок в процессе применения указателей является сама сущность данного типа — прямой доступ к памяти и запись данных по адресам, не предназначенным к хранению этих данных. Эту ошибку легко исправить, но очень непросто обнаружить. Симптомы повреждения памяти могут быть не столь явными при непродолжительном использовании кода, а последствия катастрофическими. В связи с этим, в общем случае, при работе с данным типом следует придерживаться двухэтапной стратегии. Во-первых, при использовании указателей требуются превентивные меры, заключающиеся в создании механизмов проверки состояния памяти до и после их применения, во-вторых, программный код должен быть подвергнут дополнительному тестированию на состояние памяти в процессе работы. В связи со всем вышеуказанным, предлагается ряд рекомендаций по использованию данного типа:

- инкапсулируйте операции с указателями. Абстрагируйте и локализуйте стандартные операции работы с данным типом в отдельные функциональности, повысив тем самым вероятность обнаружения ошибок, степень централизации управления памятью и коэффициент повторного использования кода;
- выполняйте объявление и определение указателей в одном месте, повысив тем самым степень защищенности кода от непреднамеренного ошибочного изменения;
- соблюдайте симметрию при выделении и освобождении памяти для указателей. По выходу из области видимости переменная данного типа должна быть установлена в начальное состояние;
- проверяйте указатель и переменную, на которую ссылается указатель, перед его применением. Указатель должен указывать на память в допустимых диапазонах, а переменная по этому адресу должна иметь допустимое значение;
- делайте код, использующий указатели как можно более ясным и читабельным. Не следует экономить на переменных-указателях с целью создания лаконичного кода или экономии памяти;
- в некоторых сложных случаях требуется создание диаграммы взаимодействия переменных-указателей, которая будет являться частью внешней программной документацией;

- следите за порядком удаления указателей для динамически созданных связанных структур данных;
- используйте дополнительные механизмы защиты при освобождении памяти (удалении указателей). Прежде всего, стоит помнить, что момент времени, когда освобожденная память станет недействительной, не определяется кодом программы и ее содержимое может выглядеть корректным еще какое-то время. Такое положение дел препятствует быстрому обнаружению ошибок. В связи с этим, желательно перед освобождением памяти вручную записывать «мусор» по этим адресам. Другой известный тип ошибок — «висячие указатели» (англ. dangling pointer) — использование указателя после его удаления. Запись данных по таким адресам может не сразу привести к видимым последствиям, тем не менее, такая «эрозия» памяти в конечном итоге ведет к непредсказуемому поведению программы. Чтение данных в этой ситуации предотвратить невозможно, однако, записывая в указатели пустое значение после их удаления, можно надежно предотвратить запись данных по таким адресам. Такой подход позволит предотвратить и другую распространенную ошибку — использование оператора деаллокации на уже освобожденном указателе.

3.5 Организация и управление вычислениями

Для эффективной поддержки жизненного цикла программы важно иметь возможность сразу составить общее представление о последовательности преобразования информации в коде посредством простого чтения, без затратной операции изучения его в отладчике. Для расчетных алгоритмов организация вычислений вполне предсказуема (отражена в первичных требованиях), однако, для системы в целом (с учетом пре/постпроцессоров) процесс преобразования информации не совсем очевиден, поскольку в указанных компонентах системы он может быть организован пользователем в достаточной степени произвольно.

Таким образом, для понимания организации вычислений в программе, ее код должен иметь самодокументирующуюся структуру. Добиться этого можно следующими способами:

- код должен быть организован таким образом, чтобы зависимости между выражениями (последовательность операций/вызовов) были очевидны. Такие связи можно подчеркнуть наименованиями, в которых будут выражаться определенные этапы вычисления, использованием методов с параметрами, выстраивающими цепочку обработки данных;
- в случае, когда вызовы не связаны какой-либо алгоритмической последовательностью, для удобства чтения кода следует применять порядок вызовов, ориентирующийся на очередность (пусть и произвольную) обработки конкретных данных, а не на очередность функций действия над произвольной совокупностью данных. При таком порядке все вызовы, связанные с обработкой определенной группы данных визуально находятся в одном месте и структурно объединены. Такой порядок вычисления естественен для императивных языков и уместен в контексте рассматриваемой проблемы;

- использовать комментарии, когда затруднительно выявить зависимости преобразования данных простым чтением кода.

3.5.1 Общие вопросы управления

Самый большой вклад в общую сложность программы вносят управляющие структуры. Известна методика количественного расчета сложности, базирующаяся на тезисе, что сложность программы определяется ее управляющей логикой [66, 67] — так называемый показатель сложности кода Маккейба. В силу этих обстоятельств, рассмотрим частные случаи управляющих структур и их вклад в проблему сложности.

3.5.2 Использование условных операторов и циклов

Все механизмы управления вычислениями строятся на основе логических структур, а последние проектируются с помощью условных операторов. Ввиду особой важности данной категории операторов, предлагается ряд рекомендаций по их использованию.

Прежде всего, в логических выражениях следует использовать логический тип. В языках, не имеющих встроенного логического типа, читабельность управляющих структур падает, возрастают риски дестабилизации кода. Предпочтительно, в проверяемых логических выражениях использовать неявные сравнения с логическим типом (например, инкапсулируя явные сравнения), повышая тем самым предметную выразительность таких структур.

Нестандартный ход вычислений должен содержаться в исключительных секциях условного оператора, а наиболее вероятные варианты действий размещаются первыми. Таким образом, нормальный ход вычислений оформляется позитивной логикой. Такое позиционирование способствует облегчению анализа организации вычислений. Понятно, что если условный оператор не имеет исключительных секций, то в этом случае сам является организатором нестандартного хода вычислений.

При наличии сложных логических условий для их упрощения следует осуществлять их декомпозицию с помощью новых логических переменных, применять теоремы Деморгана для упрощения выражений, инкапсулировать сложность в отдельные функциональные зависимости. Для особо сложных и громоздких управляющих структур можно построить таблицы решений, переместив тем самым сложность с аспекта управления на аспект данных.

Все логические выражения рекомендуется тщательно форматировать, применяя все доступные возможности оформления кода для используемого языка программирования. Однако стоит помнить, что чрезмерные отступы (или «вложенность») является серьезным препятствием к пониманию кода. В [68, 69] рекомендуется избегать вложенности более четырех уровней. Глубина вложенности может быть уменьшена за счет реструктуризации (повторной проверки части условия). Другой хорошей практикой является организация точек останова анализа сложного условия при частичной проверки его части. При такой организации механизмов управления, вместо глубоко вложенных логических конструкций, имеется ряд последовательных изолированных условий, при анализе которых делается заключение (продолжить/остановить) о ходе вычислений.

При организации циклических вычислений следует придерживаться двух правил. Во первых, минимизировать число факторов управления циклом, во

вторых, рассматривать тело цикла как инкапсулированную сущность, лишенную управляющих механизмов. В этом отношении цикл *foreach* предпочтительнее *for*, а последний — всем остальным типам. Вообще говоря, с точки зрения устойчивости кода, циклические вычисления представляют собой критическую секцию, а последствия не выявленных ошибок могут иметь далеко идущие последствия. В силу общей сложности организации таких вычислений (что является следствием компактности конструкции), уязвимыми являются секции инициализации и завершения цикла, операции с аккумулятором, структура вложенности. Дополнительную сложность обеспечивают циклические структуры обработки многомерных связанных структур данных (массивов и коллекций), которые конструируются посредством вложенности.

Прежде всего, рассмотрим рекомендации по организации точки входа в цикл:

- вход в цикл должен быть единственным;
- код инициализации цикла должен размещаться непосредственно перед ним самим;
- точка входа должна быть настолько простой, насколько это возможно.

Для организации тела цикла существуют следующие правила:

- в силу потенциальной уязвимости циклических вычислений, операторы цикла должны быть выделены средствами разметки текста. Такая визуализация способствует улучшению читабельности и предотвращению ошибок рефакторинга;
- избегать синтаксических спекуляций, при которых тело цикла оказывается пустым (например, размещение тела цикла в секции управления);
- операции по обслуживанию цикла следует размещать в одном месте (в начале или конце тела);
- операторы в теле цикла должны быть подчинены выполнению только одной задачи. Объединение различных задач в одном цикле допускается в том случае, когда оценка производительности покажет проблему при использовании нескольких раздельных циклов.

Для операции завершения цикла имеются следующие рекомендации:

- убедитесь, что работа цикла завершится в любом случае. Требуется промоделировать все исключительные и номинальные варианты работы;
- настолько, насколько это возможно, условие завершения цикла должно быть очевидным. Прежде всего, точка выхода должна быть одна. Если требуется организовать дополнительные точки выхода, то они тоже должны быть очевидными (оформлены специальными операторами выхода, а не путем манипуляций с управляющими переменными цикла);
- не использовать переменные управления цикла в не его конструкции (хорошие компиляторы запрещают такое использование);
- в качестве дополнительного инструмента контроля работы цикла можно использовать счетчики безопасности, призванные контролировать максимальное число проходов цикла (хотя бы оценочно).

При конструировании циклических вычислений, хорошей практикой является контроль граничных точек. Таковыми являются первая и последняя итерации и еще

одна, кроме указанных. Все эти точки должны быть проверены вручную с целью убедиться, что они работают в штатном порядке. Все специальные случаи/исключения, выполнение которых отличается от граничных точек, тоже проверяются вручную. Готовность выполнять такой вид проверки (мысленное моделирование и ручной счет) — ключевая черта профессионального программиста.

Подводя итог сказанному по поводу сложности вычислений в циклах, отметим, что рекомендуется делать короткие циклы для повышения читабельности кода (для этого можно инкапсулировать часть взаимосвязанных операторов) и ограничить их вложенность (не более трех вложений).

3.5.3 Частные случаи управляющих структур

Некоторые подходы к управлению, применяемые в разное время в программировании, при неуместном применении могут привести к полной дискредитации всей системы управления вычислениями. Однако они могут быть очень эффективны при их аккуратном использовании.

В частности, некоторые языки поддерживают способы возврата управления после частичного выполнения метода. С их помощью можно построить ясный механизм обработки ошибок передачи аргументов внутри функции, проводя последовательную проверку каждого аргумента с выходом в случае некорректного значения, вместо сложного условия номинального действия. При бездумном же применении таких возможностей логика метода тяжело поддается пониманию.

Другим таким подходом является рекурсия, которая позволяет осуществить элегантную декомпозицию общей сложности задачи, однако необдуманное ее применение влечет к катастрофическим последствиям. Этот метод управления сложностью был дискредитирован неуместными примерами применения (прежде всего в учебной литературе). Являясь мощным инструментом решения сложных задач, он может существенно замедлить вычисления и привести к непредсказуемому использованию памяти, препятствовать пониманию последовательности вычислений. Иными словами, применение рекурсии оправдано только после рассмотрения возможностей всех других альтернатив. Общие рекомендации по использованию рекурсии следующие:

- убедитесь в наличии надежных условий остановки рекурсии. Для этого можно использовать счетчики безопасности;
- избегайте косвенных рекурсий. Всегда можно перепроектировать методы таким образом, что бы рекурсия была инкапсулирована в одном из них;
- следить за использованием памяти стека. Желательно существенно ограничить число локальных переменных и при возможности избегать генерации автоматических объектов в стеке.

Еще одной «опасной» возможностью построения управляющих структур является оператор безусловного перехода. В истории программирования было сформировано негативное отношение к его использованию [70]. Выдвигались следующие аргументы против применения данного оператора:

- код, использующий оператор безусловного перехода трудно форматировать;

- некоторые компиляторы могут испытывать проблемы с оптимизацией кода, использующего оператор безусловного перехода;
- существует мнение, что код, содержащий данные операторы, не самый быстрый и короткий из всех возможных [71].

Тем не менее, десятилетия развития промышленного программирования не смогли убедительно показать явную вредоносность оператора безусловного перехода. В частности, в [72] сделан вывод, что нереалистические тесты, плохой анализ данных и неубедительные примеры не подкрепляют заявление некоторых теоретиков программирования, что число ошибок в коде пропорционально количеству операторов безусловного перехода. В защиту практики применения данного оператора выдвигаются следующие аргументы:

- большинство аргументов против оператора безусловного перехода обосновывают негативные последствия неразборчивого его использования. Осторожное применение оператора при определенных условиях может способствовать построению ясной и эффективной управляющей структуры;
- уместное применение оператора безусловного перехода способно избавить от проблемы дублирования кода в логических структурах с ветвлениями. Использование других конструкций в некоторых современных языках не позволяют решить указанную проблему;
- в тех же материалах, в которых давалась негативная оценка практики применения оператора безусловного перехода [71] имеются примеры, когда его использование позволяет создавать быстрый и короткий код.

Имеется немало примеров удачного применения оператора в решении узкоспециализированных задач. Например, обработчик ошибок выделения ресурсов.

Подводя итог по использованию этого управляющего оператора, можно заметить, что он позволяет избежать глубокой вложенности управляющих структур и может способствовать повышению читабельности кода. Тем не менее, в промышленном программировании отношение ко всем управляющим структурам (и к оператору безусловного перехода в частности), рассмотренным в этом разделе, скорее скептическое, нежели позитивное. Слишком велик риск нанести ущерб структурированности и возможности управлению сложностью в погоне за гибкостью и контекстному удобству реализации (вычисляемые метки безусловных переходов, плавающие точки входа в методы, самомодифицирующийся код).

3.5.4 Практики уменьшения сложности управляющих структур

С точки зрения поддержки жизненного цикла программного обеспечения, сложный код является признаком того, что задача, поставленная перед разработчиками, была понята ими не до конца. В отношении управляющих структур можно заметить, что наличие глубокой вложенности, сложных логических формул, специфичных конструкций управления показывает, что код нуждается в рефакторинге.

Рассмотрим некоторые практические приемы уменьшения сложности структур управления.

3.5.4.1 Табличная схема представления информации

Табличное представление информации (табличные методы поиска информации) представляет собой альтернативу сложной логической структуре многовариантного поиска. В этом случае аспект сложности проблемы смещается с области действий в область данных. Такой подход является перспективным для расчетного программного обеспечения, поскольку в любой постановке вычислительные задачи базируются на данных, а операторы действия являются лишь правилами отображения. Табличное представление эффективно применяется лишь для сложно структурированной информации, в тривиальных случаях логическая поисковая структура более эффективна. Использование различных типов доступа к информации в табличном представлении допускает применение такого подхода для широкого спектра задач. Дополнительную гибкость этому подходу (в конечном итоге — его успешность) обеспечивает многовариантность представления информации в табличном виде.

3.5.4.2 Структурное программирование

Структурное программирование — это формальный подход к уменьшению сложности управляющих структур на уровне кодирования (не путать со структурным проектированием). Впервые этот термин формализован в [70].

Вся суть подхода состоит в том, что процесс управления реализуется конструкциями с одним входом и одним выходом. Данная конструкция представляется изолированным блоком кода (последовательностью связанных операторов), управление на который передается со строго фиксированной точки, а выход из него имеется только один. Таким образом, программа пишется в упорядоченной, дисциплинированной манере, читается сверху вниз и практически так же выполняется.

Основу структурного программирования составляют три компонента:

- последовательность — набор связанных операторов, выполняющихся последовательно;
- выбор — простая управляющая конструкция выбора последовательности;
- итерация — многократный выбор определенной последовательности.

В [73] показано, что любая управляющая логика может быть реализована этими тремя компонентами. Естественно, современные языки предлагают широкие возможности построения гибких и удобных механизмов управления, однако их непродуманное применение в практике промышленного программирования сопряжено с постоянно растущими издержками.

4. Повышение качества ПО

Согласно [74], качество ПО — это способность программного продукта при заданных условиях удовлетворять установленным или предполагаемым потребностям. Другое определение — весь объем признаков и характеристик программ, который относится к их способности удовлетворять установленным или предполагаемым потребностям [75]. И в том и другом определении речь идет о наличии определенных свойств/ограничений программного обеспечения, способствующих удовлетворению регламентируемых потребностей пользователя.

Разделяют две категории свойств/характеристик качества: потребительская или эксплуатационная (внешняя, обеспечивающая коммерческую привлекательность программного продукта), техническая (внутренняя, обеспечивающая возможность поддержки жизненного цикла программного продукта) (на основе ISO 9126-1 — ISO 9126-4, ISO 14598). К первой категории относятся следующие:

- корректность — отсутствие/наличие дефектов в спецификации, проекте и реализации системы;
- практичность — легкость достижения целей средствами системы (с учетом вычислительной производительности);
- эффективность — технические издержки использования программы (потребление системных ресурсов);
- надежность — способность системы реализовать заявленные функции в установленных ограничениях (предопределенные условия). В качестве количественного критерия выступает средний интервал между отказами;
- целостность — способность системы предотвращать использование несанкционированных данных (обеспечить устойчивость системы по данным), осуществлять политики безопасности в области доступа к системе (обеспечить устойчивость системы по интерактивному воздействию);
- адаптируемость — возможность использования системы без ее изменения в смежных областях или нецелевых средах;
- правильность — адекватность модели представления результатов расчета (как правило, количественные характеристики) для решаемой задачи;
- устойчивость — способность системы реализовать определенную функциональность и обеспечивать сохранность данных в неопределенных условиях.

Внешние характеристики — единственная категория качества, которая интересует конечного пользователя. Никакие технические характеристики сами по себе не способствуют решению конкретных задач потребителя. В этом отношении программный продукт не отличается от продуктов других подобных сфер человеческой деятельности. Тем не менее, коммерческий успех программы не определяется только свойствами потребительской категории, что, в конечном итоге, и отличает эту сферу деятельности от других. Внутренние характеристики качества напрямую влияют на коммерческую привлекательность результата разработки ПО. К этой категории относятся:

- удобство сопровождения — легкость рефакторинга и реинжиниринга;
- гибкость — возможный объем изменения системы с целью ее переориентации (перепроектирование системы с целью ее использования в тех областях или средах, на которые она не была ориентирована ранее);
- портируемость — возможный объем изменения системы в направлении кроссплатформенности;

- возможность повторного использования — масштабность и легкость использования компонент системы в других проектах;
- удобочитаемость — легкость чтения и понимания исходного кода системы (в том числе и на детальном уровне);
- тестируемость — возможная степень покрытия кода функциональными и системными тестами;
- понятность — легкость понимания системы, как на архитектурном уровне, так и на уровне вычислительных алгоритмов и связанных с ними типов данных.

Некоторые из указанных характеристик могут перекрываться, но каждая из них имеет свой уникальный проблемный акцент. Категории качества в целом паретто-оптимальны — достижения оптимума по всем критериям невозможно (улучшение некоторых аспектов качества может приводить к ухудшению других). Критерии, принадлежащие одной категории, также оказывают влияние друг на друга. Например, корректность усиливает надежность, ослабляя при этом устойчивость.

4.1 Процессы, направленные на повышение качества ПО

Контроль качества ПО — это комплексная программа действий, гарантирующая, что система будет обладать указанными в первичной документации (техническое задание) характеристиками. Ниже перечислены виды деятельности (методики), напрямую влияющие на повышение качества.

- *Явное определение целевых внешних и внутренних характеристик качества ПО.*
- *Явный контроль качества*. В случае, когда характеристики качества не включены явным образом в список приоритетных целей при разработке ПО, программный код создается без их учета. И только персональная ответственность, возложенная на разработчика административным порядком, может изменить ситуацию.
- *Выработка стратегии тестирования*. Само по себе тестирование опосредовано влияет на качество. Этот процесс в большей степени применяется для контроля реализации требований к системе. Тем не менее, данный вид деятельности, будучи компонентой системы контроля качества, успешно справляется с задачами диагностики дефектов на разных этапах жизненного цикла программного кода.
- *Формальные и неформальные технические обзоры*. Важным аспектом повышения качества являются приемы его оценки на разных этапах разработки ПО и чем раньше будут выявлены проблемы, тем меньше затрат будет требоваться на их устранение. Такими приемами являются инспекции кода, его аудит, выполняемый сторонними специалистами (формальные технические обзоры) или самим разработчиком (неформальные технические обзоры). Такие обзоры обычно проводятся при смене этапов процесса разработки (построение архитектуры, конструирование, тестирование) и позволяют определить достаточность выполнения специализированных требований для каждого этапа при переходе к следующему.

В процессе разработки ПО возможны такие виды деятельности, которые сами по себе не являются аспектами контроля качества, но, тем не менее, оказывают на него влияние. К ним относятся:

- использование систем версий кода. Изменения, вносимые в систему, сами по себе являются мощным дестабилизирующим фактором, приводящим к снижению ее качества. Бесконтрольный характер таких изменений еще больше усугубляет ситуацию;
- прототипирование. Разработка реалистичных моделей важных функций системы повышает эффективность процесса разработки системы по сравнению с традиционными методиками разработки, основанными на полных спецификациях [76] и контроль характеристик качества для таких моделей менее трудозатратен.

По поводу эффективности рассмотренных методик можно сказать следующее. Традиционным способом оценки является определение процента обнаруженных дефектов из общего их числа (ориентировочное значение по совокупности применения комплекса методик из системы контроля качества) на конкретном этапе. Как было показано в [77], типичная эффективность каждой отдельно взятой методики не превышает 40%. И только применение широкого диапазона методик позволяет достигнуть показателя 95% (совместно использовались различные вариации тестирования и инспекции с использованием спецификаций). По поводу стоимости использования методик имеются исследования [78], показывающие, что инспекции обходятся, как правило, дешевле тестирования.

Таким образом, подводя итог всему выше сказанному, сделаем вывод, что эффективная программа контроля качества должна включать совокупность методик, которые последовательно применяются на всех стадиях разработки. В качестве такой комбинации может выступать следующая [6]: формальные инспекции всех требований, аспектов архитектуры и критических частей системы; прототипирование; инспекции кода; тестирование.

4.2 Методики совместного конструирования

Формальные/неформальные инспекции и технические обзоры, а также различные формы экстремального программирования, (например, парное программирование) являются методиками повышения качества, где ответственность за результаты разделяет группа исполнителей. Существуют исследования [79], показывающие эффективность совместного конструирования. Более того, в [54, 80] утверждается, что методики совместной разработки способны выявлять такие дефекты, которые другими методами (например, тестированием) не обнаруживаются.

При совместном конструировании код программы принадлежит не отдельному человеку, а группе специалистов, что обеспечивает следующие преимущества:

- увеличивается число специалистов, разрабатывающих конкретный код, что способствует повышению его качества;
- потеря одного из участников проекта не приводит к серьезным последствиям;
- имеется возможность разделения работ в группе. Исправление ошибок может происходить параллельно процессу дальнейшего

конструирования, что способствует устранению эффекта накапливания дефектов, искажающих, в конечном итоге, весь процесс разработки.

Сами по себе стандарты программирования будут бесполезны, если не настаивать на их применении. Технические обзоры, которые обычно следят за соблюдением стандартов, являются средством, обеспечивающим обратную связь для исполнителя, касающуюся качества его кода. Одним из видов технического обзора является формальная инспекция, показывающая высокую эффективность обнаружения дефектов и требующая меньших затрат, чем тестирование. Инспекции отличаются от других видов обзора следующими особенностями:

- при инспекциях используются контрольные списки требований к коду, которые концентрируют внимание экспертов на конкретных задачах (в том числе и на областях, с которыми ранее были связаны проблемы);
- целью инспекции является обнаружение, а не исправление дефектов;
- все участники инспекции не является авторами продукта, подвергающегося инспекции;
- данные, полученные при каждой инспекции, в дальнейшем будут использоваться для улучшения будущих инспекций;
- руководство не оказывает административное давление на команду инспекции (допускается участие технических лидеров).

Результаты инспекционных проверок позволяют поддерживать концентрацию разработчиков на важных задачах, отраженных в контрольных списках. Стандартизация таких списков обеспечивает систематичность инспекционных процессов, а наличие формальной обратной связи, используемой для улучшения контрольных списков, делает процесс инспекции самоорганизующимся. Таким образом, благодаря постоянной оптимизации процесса инспекции она становится эффективным способом устранения дефектов. Проведенные исследования (модель зрелости процессов CMM Software Engineering Institute) показывают, что процесс инспекции соответствует самому высокому уровню эффективности (является систематичным, самооптимизирующимся на основе обратной связи, поддающимся оценке).

По поводу преимуществ парного программирования заметим следующее. Данная методика является весьма эффективной в условиях ограниченного времени. При одиночном программировании, накапливающийся в таких условиях стресс подталкивает разработчика к неэффективным (непродуманным) решениям. В то время как рассматриваемый подход обладает всеми преимуществами совместного конструирования при наименьших издержках на команду, а именно: распространение корпоративной культуры (передача опыта между членами команды напрямую), содействие совместному владению результатами работы.

4.3 Тестирование в процессе разработки

Тестирование — одна из самых популярных методик повышения качества, является средством поиска ошибок (в отличии от отладки — средства устранения причин обнаруженных ошибок). Тестирование разделяют на две категории: тестирование методом «черного ящика» и методом «белого ящика». В первом случае специалист не владеет сведениями о внутренней работе тестируемого элемента (тестирование выполняется специализированным персоналом), во втором — такие сведения имеются и именно эта категория используется при тестировании

самим разработчиком. В категорию тестирования «белым ящиком» входят такие виды тестирования:

- блочное тестирование — изолированное тестирование полного класса, метода или небольшого приложения, написанного, как правило, одним программистом;
- тестирование компонента системы, выполняемое в изоляции от остальных ее частей;
- интеграционное тестирование — это применение блочного и компонентного тестирования совместной работы нескольких классов, пакетов, компонент системы.

Тестирование часто становится единственной компонентой программы контроля качества, хотя и является самой дорогой и менее эффективной методикой [81]. Данная методика не доказывает отсутствия ошибок, не повышает качество ПО, а является лишь средством его оценки.

4.3.1 Приемы тестирования

Полное тестирование кода — это NP-полная задача. Поскольку полное тестирование затруднено, на практике используется выборка тестов, способная обеспечить максимальное процентное соотношение обнаружения ошибок. Для составления такой выборки используется ряд методов, показанных ниже.

4.3.1.1 Структурированное базисное тестирование

Основой этого метода является тезис, заключающийся в том, что тестов должно быть столько, что бы ими можно было покрыть каждый оператор хотя бы один раз. Таким образом, для создания *минимального* набора тестов, осуществляющих покрытие всех базисных элементов кода, требуется подсчитать число возможных путей выполнения программы. Данный метод тестирования применяется для выявления дефектов вычисления (тестирование потоков управления) только с одним набором данных и не может являться единственным инструментом в процессе тестирования.

4.3.1.2 Тестирование потоков данных

Еще один метод тестирования, основывающийся на положении, что использование данных подвержено ошибкам в той же мере, что и использование потоков управления.

Данные могут находиться в трех состояниях: определение (инициализированы, но не использованы), использование (в качестве операндов), уничтожение (состояние недействительности). Так же для описания манипуляций с состояниями данных в блоках кода с отдельным адресным пространством (процедуры/функции, модули, стек) используются термины:

- вход — поток управления входит в изолированное адресное пространство, над переменной выполняются действия;
- выход — поток управления покидает метод сразу после выполнения действий над переменной.

Номинальный жизненный цикл переменной состоит из последовательности: определение, использование, уничтожение. Любая другая последовательность является дефектом работы с данными. Эти ошибки должны быть устранены до

процедуры тестирования, так как рассматриваемый метод не применяется при наличии таких дефектов. В дальнейшем, работа всех тестов сводится к анализу комбинации «определение — использование» для всех переменных. Такой анализ может быть выполнен на разных уровнях:

- проверка всех определений («мягкий» уровень). Данный уровень проверки может быть выполнен с использованием методики структурированного базисного тестирования;
- проверка всех комбинаций «определение — использование».

Таким образом, процесс тестирования должен включать набор тестов, сформированных с использованием, как минимум, двух подходов — структурированного базисного тестирования и тестирования потоков данных.

Подводя итог, отметим, что тестирование в процессе разработки является важнейшим компонентом полной стратегии тестирования. Процессу подборки и созданию тестов желательно уделять столько времени, сколько это необходимо, следуя различным методикам тестирования. Процесс тестирования будет максимально эффективным, если проводить его регулярно по мере написания нового кода или по завершении компонента системы.

4.4 Рефакторинг

Существует мнение, что любая модификация кода, проходящая с сохранением большей части выработанной ранее спецификации, технологической платформы и инструментальных средств (так называемая «консервация проекта»), называется рефакторингом. Есть и другая точка зрения, согласно которой рефакторингом имеет смысл называть только те изменения (в рамках установленных ранее технологической платформы и инструментальных средств), которые обеспечивают постепенное повышение качества системы при ее разработке и дальнейшем сопровождении. Так, например, [82] определяет этот процесс как «изменение внутренней структуры ПО без изменения его наблюдаемого поведения, призванное облегчить его понимание и удешевить модификацию». Рассмотрим этот процесс в контексте последнего определения как деятельность, направленная в первую очередь на повышение качества ПО.

Согласно каскадной модели разработки ПО, после тщательной выработки требований и их реализации, значительные изменения кода возможны только при его сопровождении. Такая точка зрения оправдана только для небольших программ, когда действительно удается исчерпывающим образом описать все требования, и предполагается, что возможности программы в дальнейшем не будут развиваться. Во всех остальных случаях, выработка требований существенно затруднена общей сложностью проблемы и в процессе разработки код может претерпевать значительные изменения. Иными словами, предсказуемость кодирования заметно снижается. Все те изменения, которые вносятся в процессе конструирования, при отсутствии должного контроля могут, в конечном итоге, привести к энтропии проекта, когда в условиях ограниченного времени (отведенного на реализацию проекта) теряется возможность управления ходом изменений из-за резко возросшей излишней (в силу хаотичности процесса) сложности. Такая ситуация может сложиться и в процессе сопровождения ПО. Таким образом, после внесения некоторого критического (определяется экспертом) количества изменений в проект, требуется оценить ситуацию в аспекте управления сложностью и осуществить действия (перепроектирование, повторное

кодирование), направленные на уменьшение текущей сложности, то есть осуществить процедуру рефакторинга. Контролируя сложность проекта на определенных этапах конструирования, достигается повышение качества готового продукта.

Выбор момента, когда надо осуществить рефакторинг определяется экспертом, но и для самого разработчика существуют явные подсказки, указывающие на ухудшение состояния кода. Перечислим самые явные.

- *Повторение фрагментов кода*. Такой дефект декомпозиции (ошибка факторизации на этапе проектирования) вынуждает параллельно вносить изменения сразу в несколько мест программы.
- *Инкапсуляция слабосвязанных вычислений*. Тоже является дефектом факторизации, но, в отличие от предыдущего, на другой парадигме — ООП. Имеет вид суперкласса или многофункционального, обширного метода. Естественно, при такой факторизации код теряет гибкость, усложняется процесс тестирования и верификации.
- *Незначительные изменения по данным или по вычислительным потокам ведет к масштабной перестройке иерархии классов*. Является дефектом объектно-ориентированной абстракции по данным или по управлению.
- *Метод класса использует больше данных другого класса, чем своего*. Является дефектом объектно-ориентированной абстракции по организации процесса вычисления.
- *Слишком низкоуровневая реализация поведения сложных абстракций*. Является дефектом объектно-ориентированного моделирования предметной области. Случай, когда более сложная абстракция, которая должна манипулировать абстракциями меньшей сложности, вместо этого использует простые типы данных (явное низкоуровневое представление менее сложных абстракций).
- *Неразвитый класс, имеющий один метод, реализующий поведение объекта*. Является дефектом объектно-ориентированного моделирования предметной области.
- *Различные нарушения сокрытия данных*. Дефект интеллектуальной системы управления данными.
- *Многопараметрический метод*. Данный дефект абстракции интерфейса метода, как правило, указывает на неудачную факторизацию.
- *Многократно используемый набор совместных данных*. Является дефектом абстракции данных (или абстракция для используемой парадигмы отсутствует вовсе).
- *Транзитные данные*. Интерфейс метода содержит данные, которые непосредственно не используются, а передаются вызовам внутри метода. Является дефектом вычислительных потоков.
- *Перед вызовом метода присутствует код, подготавливающий данные, обрабатываемые в данном методе (после вызова метода может присутствовать код восстановления данных)*. Может являться дефектом факторизации.
- *Сложные логические условия, имеющие глубоко вложенную структуру*.

- *Наличие фрагментов кода, снабжаемых обильными комментариями*. Дефект понимания поставленной задачи. Неочевидный ход вычисления снабжается дополнительной спецификацией.
- *Конструирование впрок*. Код программы содержит прототипы моделей вычислений, которые, по мнению разработчика, могут пригодиться в последствие. Является дефектом спецификации (требования изначально не ясны, делаются попытки их угадать). При возникновении такой ситуации рекомендуется [82, 83] произвести подготовку иного характера — уделить внимание ясности и понятности уже существующего кода с целью максимально быстрой его адаптации под меняющиеся требования.

В процессе конструирования присутствуют ситуации, которые сразу могут побуждать к проведению рефакторинга. В таких случаях он будет особенно эффективным, предупреждая возможные проблемы до их появления. Так, проведение рефакторинга будет уместно в следующих ситуациях: при создании нового метода/класса (рефакторинг всех связанных методов/классов), при исправлении дефекта определенного типа (рефакторинг всех секций кода, которые могут содержать подобные дефекты), при сопровождении кода (внося изменения в существующий код, уместно будет произвести его улучшение).

Являясь эффективным инструментом повышения качества ПО, рефакторинг, при неумелом применении, может нанести вред коду программы. Во избежание таких последствий, рассмотрим следующие рекомендации. Перед началом рефакторинга следует убедиться, что можно будет вернуться к коду, с которого начинались изменения (применяя систем управления версиями и резервного копирования). Однотипные виды рефакторинга выполняются в пакетном режиме для всего кода сразу. При проведении поэтапной процедуры модификации кода, следует создавать специальные тесты для таких фрагментов, оценивающие возможный ущерб от непреднамеренно внесенных ошибок. По завершении всего процесса модификации, весь код программы верифицируется системой тестирования, которая содержит дополнительные тесты для измененных фрагментов кода.

Несмотря на большие возможности улучшения качества ПО, данный подход имеет ограничение на применение. Не следует оправдывать написание плохого кода с намерением исправить его последующим рефакторингом. Может возникнуть ситуация, когда энтропия кода достигнет такого уровня, что никакие модификации, направленные на его улучшение, не достигнут своей цели. В этом случае понадобится полный реинжиниринг проекта.

4.5 Повышение производительности ПО

Проблема повышения производительности ПО периодически возникает на разных этапах развития вычислительных платформ. Для некоторых классов задач, производительность, как характеристика качества ПО, играет существенную роль. Наряду с этим, стоит помнить, что производительность является всего лишь одной из компонент качества, причем, в определенных случаях, не ключевой. В целом, следует отметить, что производительность привлекает внимание пользователя лишь в том случае, когда она заметно сказывается на его работе. В большинстве случаев, пользователи склонны концентрировать свое внимание на функциональных возможностях ПО (продуктивности), а не на его абсолютной

производительности. В этом контексте под производительностью программы понимается не абсолютная скорость выполнения команд, а объем информации, которая может быть обработана за некоторый срок (время взаимодействия пользователя с программой), ясный и удобный интерфейс, который повышает продуктивность деятельности пользователя.

В случае, когда производительность является приоритетной характеристикой качества, не следует сразу приступать к улучшению быстродействия на уровне кода. Анализ вычислительной эффективности ПО требуется провести на всех уровнях: требования к программе, архитектура, конструкция классов и методов, взаимодействие со средой выполнения, инструментарий разработки, вычислительная платформа, оптимизация кода.

Довольно распространенным случаем является включение производительности в требования к системе, что обычно влечет за собой разработку сложного и дорогого кода. Учитывая возникающие при этом риски, требования к системе должны быть пересмотрены.

Архитектура программы может, как способствовать производительности, так и существенно ее затруднять. Если вычислительная эффективность является действительно важной характеристикой качества, требуется разрабатывать такую архитектуру, которая позволяет добиваться намеченных показателей. Это достигается за счет выработки целевых количественных показателей производительности для отдельных компонент системы, ее структурных элементов (иерархии классов, связанных цепочек вызовов методов, структуры отдельных методов). Такой подход позволяет получить следующие преимущества:

- наличие отдельных целевых показателей делают производительность системы предсказуемой (в общем случае, эффективность компонент системы обеспечивает производительность программы в целом). Уже на уровне проектирования выявляются проблемные подсистемы, на которые можно сместить акцент производительности;
- простые и ясные целевые показатели делают понятными проблемы, стоящие перед исполнителями, что повышает вероятность достижения поставленной задачи.

На уровне конструирования классов и методов производительность достигается за счет выбора соответствующих представлений данных и алгоритмов их обработки.

Взаимодействие со средой исполнения обычно подразумевает использование системно-зависимых вызовов или работу с динамической памятью. Если такое взаимодействие является узким местом производительности, то системные интерфейсы могут быть перенесены непосредственно в код программы и подвергнуты оптимизации. Побочным эффектом такой оптимизации является ухудшение таких показателей качества системы как адаптируемость, портируемость и, в некоторых случаях, эффективность.

Использование хорошего компилятора позволяет эффективно решать задачу повышения производительности. Для небольших системных утилит ручной подход к оптимизации всего кода может быть хорошим решением, для больших прикладных программ такой подход является утопией.

В некоторых случаях, для узкоспециализированного прикладного ПО с небольшим количеством пользователей, замена вычислительной платформы на

более производительную, будет являться достаточно дешевым решением проблем быстродействия по сравнению с оптимизацией кода.

И наконец, когда все выше рассмотренные варианты повышения производительности не обеспечили должного эффекта, следует прибегнуть к оптимизации кода (code tuning).

4.5.1 Общие вопросы оптимизация кода

Под оптимизацией понимается процесс внесения небольших изменений в корректный код с целью повышение временной эффективности процессов преобразования информации. Известно [84], что это не самый эффективный и достаточно дорогой способ (сложность в реализации и высокие издержки сопровождения) повышения производительности. В силу указанных обстоятельств, рассмотрим некоторые рекомендации, направленные на снижение издержек оптимизации.

Прежде всего, нагрузка на вычислительные потоки подчиняется принципу Парето (что свойственно всем многофункциональным системам, а прикладное ПО является такой системой по своей сути). Принимая это обстоятельство во внимание, становится понятным, что оптимизации должен предшествовать процесс тщательного профилирования. При этом, профайлинг быстродействия должен носить комплексный характер, учитывающий факторы взаимного влияния, как цепочек вызовов, так и системных/платформенных зависимостей. В результате таких действий будут выявлены узкие по производительности места системы и определена их природа. Если проблема сконцентрирована в коде программы, рекомендуется учитывать следующие предупреждения о распространенных заблуждениях, связанных с оптимизацией высокоуровневого кода.

- Сокращение числа строк высокоуровневого кода способствует получению эффективного машинного кода. Данное предположение справедливо только для одного языка — языка ассемблера, для всех остальных языков такая связь не очевидна. Более того, компактная запись высказывания (такая возможность предоставляется «синтаксическим сахаром») может привести к противоположному результату.
- Некоторые операции, по мнению разработчика, выполняются быстрее. Данное утверждение справедливо лишь отчасти. С помощью профилировщика действительно можно выяснить вклад каждой операции в итоговую производительность, затем выполнить оптимизацию. Но стоит помнить о том, что полученное улучшение достигнуто исключительно для текущего состояния среды исполнения, выбранной версии инструментов разработки (прежде всего компилятора) и релиза языка. Таким образом, для портируемых систем, оптимизация кода — процесс с непредсказуемыми результатами. Даже для целевых платформ результат не совсем очевиден. Новая версия компилятора может более качественно выполнять те виды оптимизации, которые уже были сделаны (и, таким образом, усилия были потрачены в пустую), а в некоторых случаях ручная оптимизация и вовсе не позволит компилятору использовать некоторые механизмы построения эффективного машинного кода. В итоге, для поддержания оптимальной производительности, от разработчика требуется выполнять

профилирование и ручную оптимизацию кода при каждой смене среды разработки и целевой платформы.

- Следует сразу писать оптимальный код. Идея о том, что быстродействие отдельных частей обеспечит достаточную производительность программы в целом, справедливо лишь отчасти. Для небольшого кода ограниченной функциональности можно уже на уровне требований выставлять достижимые количественные показатели производительности. Сложность проблем, решаемых такими программами, допускает адекватную оценку быстродействия компонентов без предварительного исследования. Для программного обеспечения, нацеленного на решение сложных задач, существуют следующие риски преждевременной оптимизации.
  - Практически невозможно предугадать узкие места производительности до создания работоспособной версии программы. Время, потраченное в пустую, не позволит провести оптимизацию кода, который действительно в ней нуждается.
  - В случае, когда критические секции кода угадываются правильно, невозможно определить степень их важности в системе в целом. Решение второстепенных проблем лишь усугубляет более важные. Как следствие — общая производительность неудовлетворительна.
  - Тотальная оптимизация в общем случае снижает такие качества системы как удобочитаемость и понятность. Следствием всего этого являются высокие накладные расходы на разработку и сопровождение.

Подводя итог сказанному, отметим, что главный недостаток преждевременной оптимизации — отсутствие перспективности затраченных усилий. Оптимизация, сделанная заранее, может не обеспечить требуемый уровень быстродействия для системы в целом. Данное обстоятельство может повлечь крупный рефакторинг проекта на различных уровнях (проблема производительности может заключаться в неудачной архитектуре или конструкции моделей данных), а фактор времени уже будет упущен.

Вне зависимости от целевой платформы и инструментов разработки существуют объективные причины снижения производительности, обеспеченные, прежде всего, организацией вычислительных систем универсального назначения. В первую очередь — это взаимодействие со средой выполнения. Все операции ввода/вывода, вызовы системных функций, операции со страничной памятью ее динамическое выделение являются дорогими. Следующей весомой причиной является скрытые некритические ошибки в самой программе: неоптимальная работа с системными ресурсами и оборудованием, неудачно спроектированные структуры данных. Участки кода, имеющие отношение ко всему выше указанному, заслуживают повышенного внимания при проведении оптимизации.

Итак, прежде чем приступить к процессу внесения изменений в код с целью его оптимизации, рекомендуется выполнить следующие подготовительные работы:

- осуществить рефакторинг в направлении удобства сопровождения (повысить ясность и читабельность кода, сделать факторизацию, ориентированную на легкость внесения изменений в код);

- сохранить работоспособную версию кода (с использованием систем контроля версий и резервного копирования) в качестве отправной точки оптимизации;
- осуществить профилирование кода на предмет производительности и выявить природу проблемы (системные взаимодействия, издержки применения стороннего кода, неоптимальные вычислительные потоки);
- сделать заключение о причинах неудовлетворительной производительности программы: неадекватные требования, просчеты архитектуры, проблемы конструирования (неудачная иерархия классов/последовательность вызовов, ошибки моделей данных), медленные алгоритмы, наличие фрагментов кода, нуждающихся в оптимизации.

После проведения процесса оптимизации фрагмента кода, оценивается прирост производительности и осуществляется дальнейшее профилирование, с целью выявления последующих узких мест производительности.

4.5.2 Практические приемы оптимизации кода

В этом разделе рассматриваются конкретные приемы оптимизации кода, направленные на повышение, прежде всего, его быстродействия. Некоторые из них способствуют и сокращению объема кода, но в этом случае для достижения максимального эффекта лучше прибегнуть к перепроектированию классов/методов и структур данных.

Некоторые из рассматриваемых приемов, могут быть похожи на рекомендации по проведению рефакторинга, но такое совпадение чисто формально. Стоит помнить, что рефакторинг — это процесс, направленный на улучшение внутренней структуры программы, тогда как описываемые здесь изменения, в общем случае, способствуют ее ухудшению. Если бы это было не так, то оптимизация являлась бы стандартным приемом кодирования, однако, в силу того, что ее эффект сильно зависит от окружения программы — таковым не является.

Прежде всего, рассмотрим эффективное использование управляющей логики.

- При анализе сложных логических условий, используйте сокращенную оценку выражений (если ее не поддерживает компилятор). Таким образом, вместо сложных условий, управляющая логика преобразуется во вложенные структуры. Глубина вложения в данном контексте значения не имеет, гораздо важнее предотвратить лишние проверки логических условий.
- При проведении логических сравнений в циклических структурах, позаботьтесь о преждевременной возможности выхода из цикла по достижении поставленных условий. Это достигается с помощью операторов прерывания номинального хода цикла. Если таковых не имеется, применяется оператор безусловного перехода.
- Для управляющих структур множественного выбора, следует располагать проверки условий по частоте их срабатывания. Самые распространенные варианты располагаются выше в иерархии логических условий. Естественно, при достижении положенного

количества условий, должен быть предусмотрен немедленный выход из всей управляющей структуры.

- Учитывайте быстродействие схожих структур логики для конкретных языков программирования и применяйте самые эффективные.
- Используйте табличное представление выбора, вместо сложных цепочек логических операторов.
- При работе с циклами используйте размыкание (unswitch — если внутри цикла значение условного оператора не меняется, то цикл помещают внутрь оператора), объединение (jamming — обработка коллекций одной размерности в одном цикле), развертывание (unrolling — небольшие циклы фиксированного размера обрабатываются в явном виде полностью или частично). Все вычисления, инвариантные по отношению к циклу, выносятся за его границы (выполняются предварительные вычисления). Вложенные циклические структуры строятся по принципу увеличения размерности задачи в глубину.

По поводу оптимизации вычислительных операций отметим следующее.

- При работе с данными в памяти используйте стратегию отложенных вычислений (если такие данные не будут часто использоваться по всему коду) с кэшированием (сохранением полученных значений для возможного повторного применения). В противном случае, при работе с часто используемыми неизменными наборами данных, применяйте стратегию предварительного вычисления. Суть ее заключается в следующих вариантах. Предварительное вычисление всех требуемых данных и связывание их с константами на этапе компиляции, предварительное вычисление и хранение (в файле, в базе данных) с последующим их однократным чтением (связывание с переменными/константами) и дальнейшим использованием. Однократное вычисление с последующим массовым применением. Получение как можно большего числа значений перед использованием их в цикле.
- Используйте снижение стоимости операций для целевой архитектуры вычислительной системы. В частности для Intel x86, операции деления и умножения являются низко производительными. Для данной архитектуры, с целью повышения производительности операций, рекомендуются следующие правила: замена умножения сложением, замена возведением в степень умножением/делением, для целых чисел, при умножении и делении на степени двойки, рекомендуется использовать сдвиги.
- Снижение стоимости вызовов библиотечных математических функций. В частности, вызовы трансцендентных функций большинства стандартных реализаций обладают избыточной точностью и, в связи с этим, не оптимальны по производительности. Оценив требуемую точность решения поставленной задачи, имеет смысл реализовать алгоритм вычисления таких функций самостоятельно. Если математические функции используются с константными аргументами, вычислите эти значения предварительно, и в дальнейшем, оперируйте полученными значениями как константами.

Удачное проектирование данных вносит существенный вклад в производительность системы. Приведем некоторые рекомендации по оптимизации использования типов данных.

- Везде, где это возможно (без потери точности вычислений), рекомендуется использовать типы меньшей разрядности. В общем случае не рекомендуются к применению универсальные (безтиповые) типы данных и операции приведения типов.
- Не рекомендуется структурировать данные в многомерные индексированные коллекции, следует отдавать предпочтения одномерным структурам. Предпочтительно вообще минимизировать обращение к динамическим перечисляемым типам данных. Если в процессе вычислений неоднократно требуются лишь часть данных из таких структур, можно предварительно выгрузить их значения в переменные простых типов.

Последним инструментом в борьбе за производительность кода является переписывание критических секций на языках низкого уровня. Для программных платформ (.NET, JVM) таким языком будет являться любой нативный высокоуровневый, а для последнего — язык ассемблера для целевой аппаратной платформы. Типичный подход к оптимизации с помощью вставок низкоуровневого кода выглядит следующим образом:

- инкапсулируйте секцию критического кода;
- перепишите последовательность команд наиболее простым способом, избегая специфичных высокоуровневых конструкций языка и синтаксического «сахара»;
- осуществите перевод высокоуровневых инструкций на низкоуровневую последовательность команд;
- осуществите профилирование с целью проверки прироста производительности.

5. Управление сложностью проекта

В данном разделе будут рассмотрены некоторые аспекты влияния сложности на процесс конструирования программы. А также будут отмечены ключевые моменты в деятельности по управлению проектом, способствующие успешному планированию затрат на разработку ПО. В целом, обсуждаются факторы, напрямую влияющие на успешную реализацию проектов с участием небольшой команды разработчиков.

Сложность программного продукта — это комплексный показатель множества критериев. Количественным универсальным критерием обычно выступает число строк исходного кода (без учета кода автоматической генерации). Вклад остальных компонент — алгоритмическая сложность, возможности интерфейса, широта спектра решаемых задач оценивается экспертами по-разному и не имеет количественных оценок. Критерием оценки сложности может выступать даже число пользователей программы [6]. Так, например, согласно этому критерию, простейший вид ПО — это программа, используемая, в основном, разработчиком. Далее, более сложным видом программы является программный продукт, предназначенный для использования другими людьми (коммерческое ПО). Данный продукт может эксплуатироваться в средах, отличных от той, в которой был

создан, и проходит все этапы своего жизненного цикла: выработка требований, разработка, тестирование и верификация, сопровождение. Верхний уровень сложности занимают, так называемые программные системы (пакет программ) или «системные продукты». В этом случае, основная сложность обусловлена проблемами интеграции частей системы (в частности, проблемами интерфейсов). Данный критерий оценки в некоторой степени согласуется с представлениями общей теории систем [85] (точнее, с одной из трактовок данной теории, где к определению сложности системы подходят с позиции параметрической теории систем, согласно которой состояние объекта и его свойства характеризуются значениями параметров его составляющих элементов и их взаимоотношениями).

При оценке сложности проекта, кроме сложности самой программы, учитываются проблемы взаимодействия всех его участников, включая заказчика. С ростом сложности программы, растет и количество участников, а вместе с ним и число вариантов взаимодействия между ними. Причем рост последнего числа происходит, в общем случае, мультипликативно. Естественно вырастает и число ошибок взаимодействия.

Сложность проекта влияет как на число ошибок, так и на их характер. Например, согласно [86], на уровне простейшего ПО, существенную долю составляют ошибки конструирования. На уровне «программного продукта» количество таких ошибок заметно снижается, однако к ним добавляются ошибки в требованиях и архитектуре. Именно поэтому, любой проект такого уровня требует упорядоченности взаимодействий, направленной на существенное их ограничение. Это достигается с помощью формализации отношений (документальные спецификации проекта, такие как технические требования, требования к архитектуре и так далее).

Рост числа связей взаимодействий оказывает существенное воздействие и на производительность работ команды над проектом. Причем это воздействие носит негативный характер. Понятно, что в общем случае, производительность определяется совокупностью факторов, таких как: вид ПО, квалификация персонала, инструментарий, методология. Тем не менее, при прочих равных условиях, производительность для разных типов ПО может различаться в разы [9, 87, 88, 89].

5.1 Влияние сложности на процесс разработки ПО

Как было показано выше, размер проблемы добавляет новые виды операций, выполняемые командой в проекте, и увеличивает необходимость формализации взаимодействий. Так в малых проектах конструирование занимает существенную часть всех работ, но по мере роста сложности, его доля существенно уменьшается в общих трудозатратах. Доминируют такие виды деятельности как разработка архитектуры, интеграция компонент системы, тестирование. Естественно, для больших проектов работы по конструированию достаточно объемны, однако затраты на другие виды деятельности растут заметно быстрее. Это происходит потому, что при росте сложности разные виды деятельности становятся критическими [90]. По критерию роста затрат (при увеличении сложности проблемы) виды деятельности выстраиваются следующим образом: взаимодействие коллектива, планирование, управление, разработка первичных требований, функциональное проектирование системы, проектирование и спецификация интерфейса, выработка требований к архитектуре, процесс

интеграции компонент системы в общую сборку, устранение дефектов (рефакторинг и тестирование), выработка вторичной документации (описание спецификации и конфигурации, система комментариев, руководства по сопровождению различных уровней).

Формализация отношений в процессе конструирования для группы разработчиков достигается посредством использования в работе над проектом специального документа — конфигурации. Конфигурация содержит перечень элементов системы и историю их изменений (также требований, повлекших эти изменения), а также описания контрольных точек восстановления системы (сами точки содержатся в репозитории). Такое протоколирование процесса конструирования (управление конфигурацией) позволяет поддерживать целостность системы при ее постоянном изменении и эффективно использовать время при командном подходе. Применительно к программным проектам процесс управления конфигурацией называется software configuration management (SCM). Согласно [6], при работе над проектом, в котором участвует команда, состоящая более чем из пятидесяти человек, требуется полнофункциональная схема SCM, включающая формальные процедуры резервного копирования и версирования изменений, контроль изменений требований к проекту, а также контроль над техническими документами: исходным кодом, структурным описанием системы, тестовыми вариантами.

5.2 Инструментальные средства

Роль качественно подобранного набора инструментов, применяемых в процессе разработки, весьма существенна в борьбе со сложностью. Качественный инструментарий способен повысить производительность работ вдвое [96, 97]. Не вдаваясь в описание конкретных инструментов, перечислим требования к программным системам, применяемым в качестве инструментов на разных этапах разработки.

5.2.1 Инструменты для проектирования

Инструмент данной категории должен предоставлять графические средства для построения диаграмм проекта с использованием стандартных нотаций: UML, архитектурных блок-схем, структурной иерархии, связей сущностей, потоков данных. Данный инструмент может находиться в составе некоторого CASE-средства (хотя в указанном контексте такое решение представляется избыточным).

Использование средства проектирования позволяет в визуальной форме осуществлять управление абстракциями, сохранять целостность проекта и подготавливать ясные цели для реализации на последующих этапах разработки.

5.2.2 Инструменты для конструирования

Здесь речь пойдет, прежде всего, об интегрированных средствах разработки (IDE).

Во первых, данный инструмент должен быть снабжен качественным текстовым редактором, содержащим исчерпывающие средства для управления текстом, а также адекватные языковой парадигме функциями контроля синтаксиса и семантики.

В дополнение к основным функциям текстового редактора, хорошая IDE поддерживает следующие возможности:

- компиляция и диагностика ошибок без выхода из редактора кода;
- интеграция с системами управления версиями, тестирования и отладки;
- поддержка специальных средств представления кода согласно парадигмы языка — структурная/объектно-ориентированная схема проекта («свертывание кода»);
- интерактивная справочная система (общая и контекстная);
- поддержка нескольких языков и шаблонов различных типов проектов для них (поддержка нескольких фреймверков);
- автоматический рефакторинг;
- поддержка различных видов профайлинга с генераторами отчетов о метриках;
- инструменты развертывания программных продуктов.

Существенную поддержку производительности обеспечивает широкий спектр библиотек, как входящих в средства поддержки базовых фреймверков IDE, так и сторонних производителей. С последними нужно соблюдать особенную осторожность и применять их в случае крайней необходимости, лишь исчерпав все возможности решения задачи средствами библиотек соответствующего фреймверка. Всегда имеется вероятность, что следующая версия IDE или исполнительной среды не будут поддерживать совместимость с некоторыми библиотеками. И в случае, если поставщиками таких инструментов является сторонний производитель, будет непросто получить соответствующую совместимую версию (производитель к тому времени может и вовсе перестать существовать).

В заключении можно отметить, что поскольку хороший инструментарий значительно повышает производительность, не стоит экономить на его приобретении, освоении и внедрении. Следует тщательно подбирать наборы коммерческих библиотек, создавать собственные инструментальные средства, что способствует стандартизации решений некоторых распространенных задач.

6. Культура программирования

Программирование — одна из самых сложных видов инженерной деятельности [91, 92]. В отличие от других инженерных дисциплин, где деятельность, в конечном итоге, регламентируется материальными аспектами окружающего мира, программирование является чистой умственной деятельностью. Не являясь ни полностью искусством, поскольку результатом данного вида деятельности является решение прикладных (технических, научных или социальных) задач, ни наукой (по той же самой причине), программирование в своей низшей форме является «ремеслом», то есть ручной работой, требующей специальных навыков [93]. В своей высшей форме программирование представляется инженерной дисциплиной со своими методиками и подходами, основанными как на практическом опыте решения конкретных задач, так и на теоретических постулатах системного анализа.

Именно в такой форме данный вид деятельности успешно применяется к разрешению сложных проблем. Практически все аспекты программирования направлены на преодоление сложности и являют собой попытки компенсации строго ограниченных способностей разума [91]. В частности, таковыми являются следующие.

- Факторизация (декомпозиция) системы предназначена для упрощения ее структуры для понимания.
- Обзоры, инспекции и тестирование являются способами компенсации ожидаемых человеческих ошибок. В частности, обзоры представляют собой часть «обезличенного программирования» [94], направленного на преодоление ограничений интеллектуальных способностей одного человека (групповое решение сложных проблем).
- Ограничение объема методов, упрощение логических структур призваны упростить понимание логики программы.
- Конструирование в терминах проблемной области, а не низкоуровневых деталей реализации, преследует ту же цель.
- Использование всевозможных соглашений (о наименовании, форматировании, комментариях и так далее) освобождает разум от банальных проблем в программировании.

Понятие культуры программирования включает в себя множество императивов и стандартов, призванных способствовать снижению сложности решаемой проблемы. Принятие данной культуры на основании понимания основного тезиса Дейкстры [91], что основной задачей программирования является обуздание огромной сложности компьютерных наук, позволяет совершить переход от ремесленной формы деятельности в данной дисциплине к инженерной.

Первым из таких императивов является организация деятельности, направленной на выработку абстракций, призванных обобщить существо проблемы и тем самым сформировать ее терминологию. Само развитие программирования идет преимущественно за счет повышения абстрактности программных компонентов. Применение такого подхода к решению задач называется программированием в терминах проблемной области. В частности, при проектировании структуры программы, проблемно-ориентированный подход позволяет выстроить слоевую композицию аспектов проблемной области (уровни абстракции), где каждый слой инкапсулирует свой взгляд на решение поставленной задачи. Примером такой структуры может быть следующая композиция.

Уровень 0: использование машинных команд и возможностей операционной системы. При использовании языков высокого уровня, как правило, не используется, хотя в некоторых случаях может инкапсулировать вопросы низкоуровневой оптимизации.

Уровень 1: выразительные средства языка программирования и стандартные инструменты для него. Используется как техническое отображение проблемной области.

Уровень 2: низкоуровневые алгоритмы и структуры данных. На данном уровне абстракции проблемная область представляется в виде стандартных пакетов данных (стандартные многомерные типы данных с различными механизмами доступа для используемых инструментальных средств разработки) и типовых математических алгоритмов их обработки (сортировка, поиск).

Уровень 3: примитивы проблемной области. Здесь формируется словарь проблемной области, выраженный во фракциях используемой парадигмы. Это могут быть проблемно-ориентированные классы, процедуры или модули,

предназначенные для формирования каркаса, на основе которого будет выстроено решение задачи.

Уровень 4: атрибуты и функции проблемной области. На данном уровне формируются абстракции, позволяющие работать с проблемой в ее собственных терминах. Реализация решения на этом уровне может быть понятна и не специалистам, в частности, заказчикам. Этот уровень полностью свободен от технических проблем инструментальных средств разработки, исполнительной среды, и представляет собой набор средств для работы над проблемой. Естественно, изменения в проблемной области будут сильно сказываться на этом уровне, однако к ним можно относительно легко приспособиться, используя примитивы предыдущего уровня.

Следующим рассматриваемым императивом является организация процесса управлением деятельности по разработке ПО. Для небольших проектов, качество ПО определяется качеством работы конкретного исполнителя и то, как он самостоятельно контролирует процесс разработки. В этом контексте особенную важность имеет обеспечение стабильности первичных требований (качественное техническое задание). Если обеспечить стабильность в целом не удается, следует сразу же выбрать соответствующий ситуации процесс разработки — инкрементный или быстрое прототипирование. К выработке непротиворечивых первичных требований и обеспечению их стабильности необходимо относится с особым вниманием, так как ошибки в требованиях обеспечивают гораздо более серьезные последствия, чем ошибки, допущенные на всех других этапах разработки. Далее, следует помнить, что качество должно встраиваться в ПО сразу — на этапах проектирования и конструирования. Попытки исправить ситуацию позднее (на этапе тестирования) обречены на неудачу. Так как тестирования лишь указывает на проблемные области, не делая при этом программу удобнее (дешевле) в сопровождении. Те специалисты, которые концентрируются исключительно на самом кодировании, игнорируя при этом качество этого процесса, а заодно и другие процессы разработки (выработка требований, проектирование, тестирование), в конечном итоге обеспечивают крах проекта в перспективе его жизненного цикла. Имеется достаточно статистических данных, свидетельствующих об этом.

Важным аспектом культуры программирования является удобочитаемость кода (грамотный выбор имен классов, методов и переменных, тщательное форматирование, грамотное и уместное комментирование, сокращение объема методов, сокрытие сложных логических тестов в булевых функциях, присваивание промежуточных результатов сложных вычислений дополнительным переменным и прочее), которая напрямую влияет на процесс сопровождения ПО. Разработчикам, которые приносят ясность изложения в жертву своим личным амбициям, стоит помнить, что код коммерческого ПО не принадлежит исполнителю, а является собственностью компании, то есть все этапы жизненного цикла изделия являются обезличенными. Данное состояние процесса разработки контролируется руководителем проекта и в первую очередь, через повышение уровня ответственности исполнителей. Аргументы, заключающиеся в том, что «мой код никто читать не будет» и «работа над повышением читабельности кода отнимает много времени» не состоятельны. По поводу первого аргумента можно отметить, что он не соответствует стандарту коммерческого ПО, код которого является максимально ясным, а все ограничения кода документируются. Такие требования

выдвигаются, прежде всего, из-за соображений уменьшения стоимости рисков, связанных с последующим сопровождением программы. Второй аргумент является обоснованным лишь на этапе конструирования и время, сэкономленное в этом случае, не сократит сроки выпуска готового продукта, так как возрастут издержки на этапе тестирования и последующей корректировки сложного для понимания кода. В данном контексте уместен следующий тезис: программируйте с использованием языка, а не на языке. Грамотные специалисты, прежде всего, стремятся выработать ясные цели, а уже потом определяют, как их достичь при помощи существующих инструментов. Существует достаточно объективно проблемных инструментальных средств (имеющих механизмы разрушения своей системы типов или потенциально опасные реализации поддерживаемой парадигмы программирования), пользующихся, тем не менее, достаточной популярностью, которая и способствует неуместному их использованию. Учитывая данный факт, рекомендуется грамотно использовать выразительные средства языка (опираясь на соглашения), повышая тем самым качество кода, а не разрушать его неуместными, в рассматриваемом контексте, возможностями. Иными словами, выбирать самые очевидные пути, предоставляемые языком — и есть программировать на языке, а не с использованием языка, когда идет процесс отбора его средств для достижения поставленных целей.

Вообще говоря, полноценный набор соглашений является интеллектуальным инструментом управления сложностью. В соглашении оговаривается правила использования конкретного аспекта программирования и, в данном случае, не столь важно, насколько эти правила лучше нормируют рассматриваемый аспект. Ценна именно полнота и непротиворечивость системы соглашений в целом. Основное ее призвание — избавить разработчиков от необходимости принимать разрозненные решения в стандартных ситуациях. В дополнение к сказанному, система соглашений способствует предотвращению некоторых известных опасностей. Так можно выработать правила, ограничивающие применение спорных методик и продумать механизмы компенсации их известных недостатков. Так же, соглашения стандартизируют многие низкоуровневые задачи (правила обработки запросов памяти, обработка ошибок, операции ввода/вывода). В конечном итоге, использование определенных стандартов, позволяет повысить читабельность кода для тех специалистов, которые с ними не знакомы.

Литература

1. Buschman, Frank, et al. 1996. Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. New York, NY John Wiley & Sons.
2. Fowler, Martin. 2002. Patterns of Enterprise Application Architecture. Boston, MA: Addison-Wesley.
3. Bass, Len, Paul Clements, and Rick Kazman. Software Architecture in Practice, 2d ed. Boston, MA: Addison-Wesley, 2003.
4. Basili, V. R., and B. T. Perricone. 1984. «Software Errors and Complexity: An Empirical Investigation.» Communications of the ACM 27, no. 1 (January): 42-52.
5. Willis, Ron R., et al. 1998. «Hughes Aircraft's Widespread Deployment of a Continuously Improving Software Process,» Software Engineering Institute/Carnegie Mellon University, CMU/SEI-98-TR-006, May 1998.

6. Макконнелл С. Совершенный код. Мастер-класс / Пер. с англ. — М. : Издательско-торговый дом «Русская Редакция» ; СПб.: Питер, 2005. — 896 стр.: ил.
7. Brooks, Frederick P., Jr. 1987. «No Silver Bullets—Essence and Accidents of Software Engineering» Computer, April, 10-19.
8. Jones, Capers. 1998. Estimating Software Costs. New York, NY McGraw-Hill.
9. Boehm, Barry, et al. Software Cost Estimation with Cocomo II. Boston, MA: Addison-Wesley, 2000.
10. http://netlib.org/
11. А. М. Горелик. Эволюция языка программирования Фортран (1957—2007) и перспективы его развития//Вычислительные методы и программирование, 2008, Т. 9, с. 53-71
12. Герберт Шилдт. Самоучитель C++. — СПб.: БХВ-Петербург, 2003. — 687 стр.: ил.
13. Реймонд, Эрик. Искусство программирования для Unix. — 2005. — С. 357. — 544 с.
14. Фредерик Брукс Мифический человеко-месяц, или Как создаются программные системы — Символ-Плюс, 2010. — 304 с.
15. http://c2.com/cgi/AlanKaysDefinitionOfObjectOriented
16. Don Clugston, CSG Solar Pty Ltd Указатели на функции-члены и реализация самых быстрых делегатов на C++. — RSDN Magazine #6-2004
17. Шилдт, Герберт. Теория и практика C++. Мастер — СПб: БХВ, 1996. — 416 стр.
18. Martin Ward, Language Oriented Programming. — Computer Science Department, Science Labs, 1994
19. http://thread.gmane.org/gmane.comp.version-control.git/57643/focus=57918
20. Ray Tracer Language Comparison (ffconsultancy.com/languages/ray_tracer/)
21. http://www.eclipse.org/pulsar/
22. 1.2 Design Goals of the Java™ Programming Language. http://www.oracle.com/technetwork/java/intro-141325.html
23. http://benchmarksgame.alioth.debian.org/u32q/benchmark.php?test=all&lang=java
24. http://www.theregister.co.uk/2011/06/03/google_paper_on_cplusplus_java_scala_go/
25. MSDN. Введение в WPF. Microsoft
26. http://www.mono-project.com/Supported_Platforms
27. https://www.python.org/about/
28. http://benchmarksgame.alioth.debian.org/u32/benchmark.php?test=all&lang=python&lang2=gpp&box=1
29. http://psyco.sourceforge.net/
30. http://www-01.ibm.com/software/ru/rational/
31. http://www.microsoftproject.ru/articles.phtml?aid=158#agile
32. http://www.sameeradilhan.com/advantages-and-disadvantages-of-waterfall-model-and-v-model

33. http://www.testingexcellence.com/v-model/
34. Meyers, Scott. 1998. Effective C++: 50 Specific Ways to Improve Your Programs and Designs, 2d ed. Reading, MA: Addison-Wesley.
35. Bloch, Joshua. Effective fava Programming Language Guide. — Boston, MA: Addison-Wesley, 2001.
36. Liskov, Barbara. «Data Abstraction and Hierarchy,» ACM SIGPLAN Notices, May 1988.
37. Hunt, Andrew and David Thomas. The Pragmatic Programmer. Boston, MA: Addison-Wesley, 2000.
38. Basili, Victor, L Briand, and W.L. Melo. 1996. «A Validation of Object-Oriented Design Metrics as Quality Indicators,» IEEE Transactions on Software Engineering, October 1996, 751-761.
39. Lieberherr, Karl J. and Ian Holland. 1989. «Assuring Good Style for Object-Oriented Programs.» IEEE Software, September 1989, pp. 38f.
40. Stevens, W, G. Myers, and L Constantine. 1974. «Structured Design.» IBM Systems Journal 13, no. 2 (May): 115-39.
41. Basili, V. R., and B. T. Perricone. 1984. «Software Errors and Complexity: An Empirical Investigation.» Communications of the ACM 27, no. 1 (January): 42-52.
42. R. Hindley, (1969) "The Principal Type-Scheme of an Object in Combinatory Logic", Transactions of the American Mathematical Society, Vol. 146, pp. 29–60
43. ftp://gatekeeper.research.compaq.com/pub/DEC/SRC/research-reports/SRC-045.pdf page 3
44. Brian Kernighan: Why Pascal is not my favorite language
45. Andrew Cooke. Introduction To Computer Languages
46. Vijay Saraswat. Java is not type-safe
47. Wright, Felleisen - A Syntactic Approach to Type Soundness, 1994
48. C.A.R. Hoare — The Emperor’s Old Clothes, Communications of the ACM, 1981
49. Robert Harper Haskell Is Exceptionally Unsafe
50. Robin Milner, Mads Tofte Commentary on Standard ML. — Universiry of Edinburg, University of Nigeria, 1991
51. C. Strachey — Fundamental Concepts in Programming Languages
52. Andrew W. Appel A Critique of Standard ML. — Princeton University, 1992.
53. R. Hindley, (1969) "The Principal Type-Scheme of an Object in Combinatory Logic", Transactions of the American Mathematical Society, Vol. 146, pp. 29–60
54. Milner, (1978) "A Theory of Type Polymorphism in Programming". Journal of Computer and System Science (JCSS) 17, pp. 348–374
55. Luca Cardelli, Peter Wegner On Understanding Types, Data Abstraction, and Polymorphism // ACM Computing Surveys. — New York, NY, USA: ACM, 1985. — B. 4. — T. 17. — C. 471–523. — ISSN 0360-0300. — DOI:10.1145/6041.6042
56. Handbook of Programming Languages (HPL), Volume 4: Functional and Logic Programming Languages / Editor Peter H. Salus. — Macmillan

Technical Publishing, 1998. — 250 с. — ISBN 1-57870-011-6., Jim Veitch, on CLOS, P. 107—158
57. Erik Meijer, Peter Drayton. Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages (англ.). Microsoft Corporation.
58. Amanda Laucher, Paul Snively. Types vs Tests
59. Adobe and Mozilla Foundation to Open Source Flash Player Scripting Engine
60. Psyco, a Python specializing compiler
61. C-Extensions for Python. Cython (2013-05-11). Retrieved on 2013-07-17
62. Thimbleby, Harold. 1988. «Delaying Commitment.» IEEE Software, May, 78-86.
63. Shneiderman, Ben. 1980. Software Psychology: Human Factors in Computer and Information Systems. Cambridge, MA: Winthrop.
64. http://www.ieee.org/index.html
65. Mills, Harlan D., and Richard C. Linger. 1986. «Data Structured Programming: Program Design Without Arrays and Pointers.» IEEE Transactions on Software Engineering SE-12, no. 2 (February): 192-97.
66. McCabe, Tom. 1976. «A Complexity Measure.» IEEE Transactions on Software Engineering, SE-2, no. 4 (December): 308-20.
67. Shen, Vincent Y, et al. 1985. «Identifying Error-Prone Software—An Empirical Study.» IEEE Transactions on Software Engineering SE-11, no. 4 (April): 317-24.
68. Myers, Glenford J. 1976. Software Reliability. New York, NY John Wiley & Sons.
69. Ledgard, Henry E, with John Tauer. 1987a. C With Excellence: Programming Proverbs. Indianapolis: Hayden Books.
70. Edsger Wybe Dijkstra «Go To Statement Considered Harmful» «Communications of the ACM» 1968 г.
71. Knuth, «Structured Programming with go to Statements», 1974
72. Sheil, B. A. 1981. «The Psychological Study of Programming.» Computing Surveys 13, no. 1 (March): 101-20.
73. Bohm, C, and G. Jacopini. 1966. «Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules.» Communications of the ACM 9, no. 5 (May): 366-71.
74. ISO/IEC 25000:2014(en) Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Guide to SQuaRE
75. ГОСТ Р ИСО/МЭК 9126-93. Оценка программной продукции. Характеристики качества и руководства по их применению
76. Gordon, Scott V, and James M. Bieman. 1991. «Rapid Prototyping and Software Quality: Lessons from Industry.» Ninth Annual Pacific Northwest Software Quality Conference, October 7-8. Oregon Convention Center, Portland, OR.
77. Myers, Glenford J. 1978b. «A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections.» Communications of the ACM 21, no. 9 (September): 760-68.

78. Basili, Victor R., and Richard W. Selby. 1987. «Comparing the Effectiveness of Software Testing Strategies.» IEEE Transactions on Software Engineering SE-13, no. 12 (December): 1278-96.
79. Humphrey, Watts. 1997'. Introduction to the Personal Software Process. Reading, MA: Addison-Wesley.
80. Basili, Victor R., Richard W. Selby, and David H. Hutchens. 1986. «Experimentation in Software Engineering.» IEEE Transactions on Software Engineering SE-12, no. 7 Quly): 733-43.
81. Card, David N. 1987. «A Software Technology Evaluation Program.» Information and Software Technology 29, no. 6 (July/August): 291-300.
82. Fowler, Martin. Refactoring: Improving the Design of Existing Code. Reading, MA: Addison-Wesley, 1999.
83. Beck, Kent. Extreme Programming Explained: Embrace Change. Reading, MA: Addison Wesley, 2000.
84. Bentley, Englewood Cliffs. Writing Efficient Programs, NJ: Prentice Hall, 1982.
85. Уемов А. «Системный подход и общая теория систем» Издательство: "Мысль", 1978, 272 с.
86. Jones, Capers. Estimating Software Costs. New York, NY: McGraw-Hill, 1998.
87. Putnam and Meyers «Measures for Excellence», 1992
88. Putnam and Meyers «Industrial Strength Software», 1997
89. Cusumano, Michael, et al. 2003. «Software Development Worldwide: The State of the Practice» IEEE Software, November/December 2003, 28-34.
90. Boehm, Barry and Richard Turner. 2004. Balancing Agility and Discipline: A Guide for the Perplexed. Boston, MA: Addison-Wesley.
91. Dijkstra, Edsger «The Humble Programmer.» Turing Award Lecture. Communications of the ACM 15, no. 10 (October 1972): 859-66
92. Weinberg, Gerald M. The Psychology of Computer Programming: Silver Anniversary Edition. New York, NY: Dorset House, 1998.
93. Винокур Г. О., проф. Ларин Б. А., Ожегов С. И., Томашевский Б. В., проф. Ушаков Д. Н. Толковый словарь русского языка: В 4 т. / Под ред. Ушакова Д. Н. — М.: Государственный институт «Советская энциклопедия»; ОГИЗ (т. 1); Государственное издательство иностранных и национальных словарей (т. 2-4), 1935—1940. — 45 000 экз.
94. Weinberg, Gerald M. 1998. The Psychology of Computer Programming: Silver Anniversary Edition. New York, NY: Dorset House.
95. http://economicus.gsom.pu.ru
96. Jones, Capers. 2000. Software Assessments, Benchmarks, and Best Practices. Reading, MA: Addison-Wesley.
97. Boehm, Barry, et al. 2000a. Software Cost Estimation with Cocomo II. Boston, MA: Addison-Wesley.