12+

А. А. Красильников
Е. А. Лубышев
Г. И. Журавлев

Проектирование архитектуры информационных систем

Учебник

Арсентий Красильников

Проектирование архитектуры информационных систем

«Издательские решения»

Красильников А. А.

Проектирование архитектуры информационных систем /
А. А. Красильников — «Издательские решения»,

Книга посвящена проектированию архитектуры информационных систем и объединяет управленческие и инженерные аспекты их создания. Рассматриваются жизненный цикл ИС, требования, стандарты, методологии разработки, DevOps, CASE-средства, UML, структурные методы моделирования, проектирование процессов и данных, а также современные подходы к организации разработки.

Содержание

Проектирование архитектуры информационных систем

Арсентий Александрович Красильников
Евгений Александрович Лубышев
Георгий Иванович Журавлев

Редактор Марина Андреевна Старосельская

ISBN 978-5-0071-2540-6
Создано в интеллектуальной издательской системе Ridero

Предисловие

Современная информационная система представляет собой значительно более сложный объект, чем совокупность программных модулей, баз данных и технических средств. Она существует в определенной организационной среде, поддерживает конкретные процессы, используется различными группами заинтересованных сторон, развивается во времени и должна соответствовать функциональным, техническим, эксплуатационным, нормативным и качественным требованиям. Поэтому проектирование информационных систем невозможно рассматривать исключительно как задачу программирования. Оно предполагает согласованное решение управленческих и инженерных задач: от определения целей и требований до выбора архитектуры, моделирования процессов и данных, разработки программных компонентов, организации внедрения, эксплуатации и последующего развития системы.

Настоящая книга посвящена комплексному рассмотрению проектирования архитектуры информационных систем и связанных с ним методов, процессов и технологий. Основное внимание уделяется тому, каким образом исходная потребность в автоматизации преобразуется в целостную систему проектных решений. Архитектура при этом рассматривается не как статическая схема компонентов, а как результат последовательного анализа целей, требований, ограничений, процессов, данных, технологий и условий эксплуатации. Такой подход позволяет связать технические решения с задачами организации и показать, почему качество информационной системы определяется не только выбранными программными средствами, но и качеством самого процесса проектирования.

Изложение начинается с основных понятий архитектуры информационных систем, ее видов, архитектурных структур, образцов, эталонных моделей и подходов к проектированию. Рассматриваются монолитная и микросервисная архитектуры, проектирование «сверху вниз» и «снизу вверх», архитектурные принципы, компромиссы, риски и документирование решений. Особое значение придается пониманию архитектуры как развивающейся системы, которая должна сохранять способность к изменению вместе с требованиями, организацией и технологической средой.

Значительная часть книги посвящена жизненному циклу информационных систем, поскольку архитектурные и проектные решения невозможно отделить от процессов создания, внедрения, эксплуатации, сопровождения и модернизации. Рассматриваются каскадная, поэтапная, спиральная и эволюционные модели жизненного цикла, вопросы их выбора и адаптации, а также стандартизация процессов разработки. Последовательно раскрываются отечественные и международные подходы к организации жизненного цикла, пользовательские, системные, функциональные и нефункциональные требования, способы их документирования и проверки. Тем самым требования рассматриваются как связующее звено между потребностями заинтересованных сторон и конкретными архитектурными, программными и технологическими решениями.

Отдельное внимание уделяется современной организации проектной деятельности. Рассматриваются гибкие подходы, Lean, Scrum, Kanban, DevOps и связанные с ним практики, а также автоматизация процессов разработки, интеграции, тестирования, поставки и эксплуатации. Показано значение управления версиями, непрерывной интеграции и поставки, инфраструктуры как кода, контейнеризации, оркестрации, наблюдаемости и управления инцидентами. Наряду с этим рассматриваются DevSecOps, GitOps, DataOps, MLOps, AIOps, Platform Engineering и другие направления, отражающие современное развитие инженерных практик. Вопрос выбора технологического стека связывается с архитектурой, требованиями, ограничениями и условиями эксплуатации, а не сводится к подбору отдельных популярных инструментов.

В книге также подробно рассматриваются технологии и методы непосредственного проектирования информационного и программного обеспечения. Раскрываются возможности CASE-средств, прототипного проектирования и RAD-технологий, объектно-ориентированного подхода и языка UML. Анализируются диаграммы вариантов использования, классов, состояний, компонентов, последовательности и другие средства моделирования, позволяющие описывать структуру и поведение проектируемой системы. Паттерны проектирования рассматриваются как обобщенные способы решения повторяющихся проектных задач и как средство формирования общего профессионального языка при обсуждении программной архитектуры.

Особое место занимают структурные методы анализа и проектирования. SADT и IDEF0 используются для функционального представления системы и последовательной декомпозиции ее деятельности; модели AS-IS и TO-BE позволяют перейти от исследования существующего состояния к проектированию целевого; реинжиниринг бизнес-процессов рассматривается как средство не только автоматизации, но и содержательного преобразования деятельности. IDEF3 применяется для описания последовательностей и сценариев процессов, DFD — для моделирования движения и преобразования информации, а IDEF1X и основы реляционного моделирования — для формирования согласованной структуры данных. Рассмотрение этих методов завершается их совместным применением на комплексных примерах, что позволяет показать взаимосвязь функциональных, процессных, информационных и программных представлений системы.

Важной идеей, проходящей через всю книгу, является необходимость согласования различных моделей и проектных решений. Отдельная диаграмма, архитектурная схема, модель данных или спецификация требований не обеспечивает качества проектирования сама по себе. Функции должны соответствовать потребностям и требованиям, процессы — использовать определенные данные, информационные потоки — иметь понятные источники и получателей, состояния объектов — изменяться конкретными действиями, а программные и инфраструктурные компоненты — реализовывать установленные свойства системы. Проектирование приобретает инженерный характер только тогда, когда отдельные представления образуют непротиворечивую и прослеживаемую систему.

Издание адресовано широкому кругу читателей, вовлеченных в создание и сопровождение информационных систем, и будет полезно для следующих категорий:

Будущим инженерам-разработчикам и архитекторам ПО: книга дает необходимый фундамент для грамотного технического проектирования, работы с моделями данных, применения UML и CASE-инструментов, что критически важно на этапах разработки сложных программных продуктов. Материал может быть использован при изучении дисциплин, связанных с проектированием ИС и технологиями разработки.

Менеджерам цифровой трансформации, специалистам по автоматизации и ИТ-консультантам: издание смещает акцент на управленческую сторону — методологии управления жизненным циклом, стандарты качества, интеграционные аспекты и архитектурное планирование. Это помогает профессионалам в области логистики, финансов и управления бизнес-процессами грамотно формулировать задачи для разработчиков, выстраивать эффективное взаимодействие между бизнес-подразделениями и ИТ-службами, а также принимать обоснованные решения по цифровизации.

Аналитикам и системным архитекторам: книга предлагает целостный взгляд на процесс создания ИС, объединяя структурные и объектно-ориентированные подходы, что позволяет выстраивать сбалансированную архитектуру, удовлетворяющую как техническим, так и бизнес-требованиям.

Благодаря своей универсальной структуре издание выступает практическим мостом между миром инженерии и миром управления, формируя у читателей системное мышление и навыки, востребованные на стыке ИТ и бизнеса в условиях современной цифровой эконо-

мики. Книга рекомендована как для студентов соответствующих направлений подготовки, так и для практикующих специалистов, желающих углубить свои знания в области архитектуры информационных систем.

Таким образом, книга рассматривает проектирование информационных систем как целостную деятельность, объединяющую архитектурное мышление, системный анализ, инженерные методы, организацию процессов и управление изменениями. Ее содержание последовательно ведет от общих представлений об информационной системе и ее архитектуре к формальным моделям, технологиям и практикам, используемым при создании сложных программных решений. Главная задача такого подхода состоит не в накоплении максимального количества схем, стандартов или инструментов, а в формировании способности обоснованно выбирать и согласовывать проектные решения, уменьшать неопределенность и обеспечивать соответствие создаваемой системы реальным задачам и условиям ее дальнейшей эксплуатации.

1. Введение в проектирование архитектуры информационной системы

Информационная система является одним из ключевых элементов современной организации, предприятия, государственного учреждения, образовательной структуры, медицинской организации, банка, торговой компании или цифровой платформы. Практически любая деятельность, связанная с накоплением, обработкой, передачей, хранением и использованием информации, в настоящее время осуществляется с применением информационных систем. При этом информационная система представляет собой не только совокупность компьютерных программ. Она включает данные, программное обеспечение, технические средства, пользователей, организационные процедуры, правила работы, средства связи и механизмы управления. В широком смысле информационная система представляет собой организованную совокупность взаимосвязанных элементов, предназначенных для сбора, регистрации, хранения, обработки, поиска, передачи, анализа и представления информации, необходимой для выполнения определенных функций и достижения поставленных целей. Информационная система существует не сама по себе, а в рамках некоторой предметной области. Такой предметной областью может быть банковская деятельность, электронная торговля, управление университетом, медицинское обслуживание, промышленное производство, транспортная логистика, государственное управление, бухгалтерский учет, управление персоналом или любая другая деятельность, в которой требуется систематическая работа с информацией. В российском законодательстве информационная система определяется как совокупность содержащейся в базах данных информации и обеспечивающих ее обработку информационных технологий и технических средств. Такое определение подчеркивает три базовых компонента: информацию, информационные технологии и технические средства. Однако с точки зрения проектирования реальных информационных систем этого определения недостаточно, поскольку необходимо учитывать пользователей, бизнес-процессы, организационную структуру, нормативные ограничения, регламенты доступа, способы эксплуатации и сопровождения системы. Например, информационная система университета может включать базу данных студентов, преподавателей, учебных планов, дисциплин и оценок; программные модули для формирования расписания, учета успеваемости и регистрации на учебные курсы; серверы, сетевое оборудование и пользовательские компьютеры; личные кабинеты студентов и преподавателей; правила внесения и изменения данных; процедуры зачисления, перевода и отчисления обучающихся; механизмы защиты персональных данных; средства интеграции с бухгалтерскими, библиотечными и государственными системами. Если рассматривать только программный код, невозможно получить полное представление о такой системе. Основным назначением информационной системы является информационная поддержка деятельности. Это означает, что система должна предоставлять пользователям достоверную, актуальную и необходимую информацию в нужное время и в удобной форме. Кроме того, информационная система может автоматизировать операции, контролировать выполнение процессов, поддерживать принятие решений, обеспечивать взаимодействие между подразделениями, формировать отчетность, прогнозировать события и управлять ресурсами. В структуре информационной системы обычно выделяются несколько взаимосвязанных видов обеспечения. Информационное обеспечение включает данные, документы, классификаторы, справочники, модели данных, правила кодирования и способы организации информационных потоков. Программное обеспечение включает программы, сервисы, приложения, операционные системы, системы управления базами данных и вспомогательные программные средства. Техническое обеспечение представляет собой серверы, рабочие станции, устройства хранения данных, сетевое оборудование, перифе-

рийные устройства и телекоммуникационные средства. Организационное обеспечение включает структуру управления системой, распределение обязанностей, регламенты работы, процедуры эксплуатации и сопровождения. Правовое обеспечение охватывает нормативные акты, договоры, требования к защите информации, персональных данных и интеллектуальной собственности. Лингвистическое обеспечение связано с языками взаимодействия, терминологией, классификаторами и способами представления информации. Методическое обеспечение включает инструкции, методики, правила выполнения операций и рекомендации по использованию системы. Следовательно, информационная система является социотехнической системой, поскольку объединяет технические компоненты и деятельность людей. Даже самая совершенная программа не сможет эффективно выполнять свои функции, если не определены роли пользователей, правила внесения данных, ответственность за их достоверность, порядок обработки ошибок и процедуры принятия решений.

1.1 Понятие архитектуры информационной системы

Для понимания принципов проектирования информационных систем необходимо раскрыть понятие архитектуры. В общем смысле архитектура определяет фундаментальную организацию системы, ее основные элементы, связи между ними, принципы взаимодействия и правила дальнейшего развития. Архитектура показывает не только то, из каких частей состоит система, но и то, почему эти части выделены именно таким образом, какие функции они выполняют, как взаимодействуют и какие ограничения должны соблюдаться. Архитектура информационной системы представляет собой совокупность фундаментальных решений относительно структуры системы, ее компонентов, данных, функций, интерфейсов, технологий, способов взаимодействия, развертывания, эксплуатации и развития. Архитектура связывает потребности организации с конкретными техническими и программными решениями. Архитектуру нельзя сводить к одной схеме. Она включает множество взаимосвязанных аспектов. С одной стороны, необходимо описать функции системы и процессы, которые она поддерживает. С другой стороны, требуется определить структуру данных, программные компоненты, техническую инфраструктуру, механизмы интеграции и способы обеспечения безопасности. По этой причине архитектура информационной системы обычно представляется в виде набора моделей и архитектурных представлений. Архитектура отвечает на ряд принципиальных вопросов. Необходимо определить, какие задачи решает система, какие группы пользователей с ней взаимодействуют, какие данные она обрабатывает, из каких подсистем и компонентов состоит, какие интерфейсы используются, каким образом осуществляется взаимодействие между компонентами, где размещаются программные модули и данные, какие технологии применяются, как обеспечиваются производительность, надежность, масштабируемость, безопасность и сопровождаемость. Например, при проектировании информационной системы интернет-магазина требуется определить, будут ли каталог товаров, корзина, оформление заказов, платежи, управление складом и доставка реализованы в одном приложении или в виде отдельных сервисов. Необходимо решить, будет ли использоваться единая база данных или несколько специализированных хранилищ, каким образом система будет взаимодействовать с платежными организациями, службами доставки и учетной системой предприятия, как будет выполняться масштабирование в периоды высокой нагрузки и каким образом будут защищаться персональные и платежные данные пользователей. Архитектурное решение представляет собой выбор, оказывающий существенное влияние на структуру системы, ее свойства и развитие. К архитектурным решениям относятся выбор архитектурного стиля, способ разбиения системы на компоненты, выбор централизованного или распределенного хранения данных, определение способов взаимодействия между подсистемами, выбор технологий интеграции, механизмов безопасности и стратегии развертывания. Некоторые решения легко изменить на поздних этапах разработки, а другие требуют значительных затрат. Например, изменение цвета кнопки в интерфейсе обычно не является архитектурным решением. Переход от централизованной базы данных к распределенному хранению данных, напротив, затрагивает множество компонентов, влияет на целостность данных, транзакции, безопасность и сопровождение, поэтому относится к архитектурным решениям. Проектирование архитектуры информационной системы представляет собой процесс формирования и обоснования архитектурных решений. Этот процесс начинается с изучения целей организации, предметной области, требований заинтересованных сторон и ограничений проекта. Затем определяются основные функции, данные, подсистемы, интерфейсы, технологии и способы развертывания. Полученная архитектура анализируется с точки зрения соответствия функциональным требованиям и требуемым характеристикам качества. Архитектура должна учитывать не только текущие задачи, но и возможное развитие системы. Если архитектура рассчитана исключительно на существующую нагрузку и не

допускает расширения, система может быстро перестать соответствовать потребностям организации. Поэтому при проектировании необходимо оценивать перспективы увеличения числа пользователей, объема данных, количества функций, числа интеграций и уровня требований к безопасности. Архитектура выполняет несколько функций. Организационная функция заключается в разделении сложной системы на понятные части. Коммуникационная функция состоит в создании общего представления о системе для заказчиков, аналитиков, разработчиков, администраторов и руководителей. Техническая функция связана с выбором способов реализации требований. Управленческая функция позволяет оценивать трудоемкость, риски, ресурсы и последовательность разработки. Эволюционная функция определяет направления изменения и развития системы. Качественная архитектура должна обеспечивать достижение требуемых характеристик системы. К таким характеристикам относятся производительность, масштабируемость, надежность, доступность, безопасность, удобство сопровождения, модифицируемость, тестируемость, совместимость, переносимость и удобство использования. Эти характеристики часто называют атрибутами качества. Например, для банковской информационной системы особенно важны безопасность, целостность данных, доступность и возможность аудита операций. Для видеоплатформы критическое значение имеют масштабируемость, производительность и устойчивость к высоким нагрузкам. Для небольшой внутренней системы учета может быть важнее простота разработки и сопровождения. Следовательно, не существует одной архитектуры, одинаково подходящей для всех систем. Архитектурные решения выбираются с учетом целей, масштаба, рисков и ограничений конкретного проекта.

1.1.1 Заинтересованные стороны и архитектурно значимые требования

Архитектура формируется не изолированно, а с учетом интересов различных участников, которых называют заинтересованными сторонами. К ним относятся заказчики, пользователи, владельцы бизнес-процессов, разработчики, архитекторы, системные администраторы, специалисты по информационной безопасности, тестировщики, руководители организации, контролирующие органы и внешние партнеры. У каждой группы имеются собственные интересы. Руководство заинтересовано в снижении затрат, повышении эффективности и управляемости. Пользователи ожидают удобства, понятности и быстрого выполнения операций. Разработчики заинтересованы в четкой структуре, тестируемости и возможности внесения изменений. Администраторы требуют наблюдаемости, надежности и простоты развертывания. Специалисты по безопасности обращают внимание на контроль доступа, аудит, защиту каналов связи и данных. Архитектура должна учитывать эти интересы и находить компромиссы между ними. Требования, оказывающие существенное влияние на архитектуру, называются архитектурно значимыми требованиями. Они могут быть функциональными и нефункциональными. Функциональные требования определяют, что должна делать система. Нефункциональные требования определяют, каким образом и с какими характеристиками система должна выполнять свои функции. Например, требование «система должна позволять пользователю оформить заказ» является функциональным. Требование «95% операций оформления заказа должны завершаться не более чем за две секунды» относится к производительности. Требование «система должна выдерживать не менее десяти тысяч одновременных пользователей» относится к масштабируемости. Требование «система должна сохранять работоспособность при отказе одного сервера» связано с надежностью и доступностью. Требование «доступ к медицинским данным должен предоставляться только уполномоченным сотрудникам» относится к безопасности. Архитектор должен выявить такие требования на ранних этапах, поскольку именно они определяют структуру будущей системы. Если требование к высокой доступности обнаруживается только после создания приложения, может потребоваться серьезная переработка механизмов хранения данных, развертывания и обработки отказов.

1.2 Классификация информационных систем

Классификация информационных систем необходима для систематизации их многообразия. Информационные системы могут различаться по назначению, масштабу, уровню управления, степени автоматизации, характеру обработки данных, архитектуре, режиму работы, сфере применения и другим признакам. Одна и та же система может одновременно относиться к нескольким классам. По масштабу применения выделяются персональные, групповые, корпоративные, межорганизационные, государственные и глобальные информационные системы. Персональная информационная система предназначена для поддержки деятельности одного пользователя. Примерами являются персональный планировщик, программа ведения личных финансов, локальная база контактов или индивидуальная система учета документов. Групповая информационная система используется ограниченной группой сотрудников или одним подразделением. Например, система управления задачами отдела, система учета заявок технической поддержки или база документов кафедры. Корпоративная информационная система охватывает деятельность всей организации или значительной ее части. Она объединяет данные и процессы различных подразделений, обеспечивает единое информационное пространство и поддерживает управление ресурсами предприятия. Примерами являются системы класса ERP, комплексные банковские системы, корпоративные медицинские системы и информационные системы крупных университетов. Межорганизационная информационная система обеспечивает взаимодействие нескольких самостоятельных организаций. Например, система электронного документооборота между поставщиками и заказчиками, платформа электронных торгов, система взаимодействия банков и платежных организаций. Государственная информационная система создается для выполнения функций органов государственной власти и предоставления государственных услуг. Такие системы имеют особые требования к безопасности, надежности, совместимости, хранению данных и нормативному регулированию. Глобальная информационная система функционирует в международном масштабе и обслуживает большое число пользователей, организаций и регионов. К этому классу могут относиться международные платежные, поисковые, коммуникационные и облачные платформы. По уровню управления выделяются операционные, управленческие, аналитические и стратегические информационные системы. Операционные информационные системы поддерживают повседневные повторяющиеся операции. Они регистрируют заказы, платежи, перемещения товаров, обращения клиентов, посещения, начисления и другие события. Основной задачей таких систем является точная и своевременная обработка большого количества транзакций. Например, кассовая система магазина регистрирует продажу товаров, рассчитывает стоимость покупки, принимает оплату, уменьшает остатки на складе и формирует чек. Банковская операционная система обрабатывает переводы, платежи, внесение и снятие средств. Управленческие информационные системы предоставляют руководителям информацию для контроля и управления деятельностью. Они формируют отчеты, показатели, сводные таблицы и уведомления об отклонениях. Такие системы используют данные операционного уровня, но представляют их в агрегированном виде. Например, руководитель торговой сети может получать отчеты о продажах по регионам, магазинам, категориям товаров и периодам. Руководитель университета может анализировать контингент студентов, успеваемость, нагрузку преподавателей и выполнение учебных планов. Системы поддержки принятия решений предназначены для анализа сложных ситуаций, сравнения вариантов и оценки последствий решений. Они используют аналитические модели, прогнозирование, оптимизационные методы, статистику и визуализацию данных. Например, логистическая система может выбирать оптимальные маршруты доставки с учетом расстояний, загрузки транспорта, дорожной обстановки и сроков. Кредитная система может оценивать риск выдачи кредита на основе данных о заем-

щике. Стратегические информационные системы поддерживают долгосрочное планирование и развитие организации. Они позволяют анализировать тенденции, оценивать положение на рынке, прогнозировать изменения и моделировать различные сценарии. По функциональному назначению выделяются бухгалтерские, финансовые, производственные, маркетинговые, логистические, кадровые, медицинские, образовательные, справочно-правовые, геоинформационные и другие системы. Бухгалтерские информационные системы автоматизируют учет хозяйственных операций, формирование проводок, расчет налогов и подготовку отчетности. Финансовые системы поддерживают бюджетирование, управление денежными потоками, анализ доходов и расходов. Производственные системы обеспечивают планирование и контроль производственных процессов, управление оборудованием, материалами и качеством. Логистические системы управляют перевозками, складами, маршрутами и поставками. Кадровые системы хранят сведения о сотрудниках, учитывают рабочее время, поддерживают расчет заработной платы и управление персоналом. По степени автоматизации выделяются ручные, автоматизированные и автоматические системы. В ручной информационной системе операции выполняются человеком без применения специализированных средств автоматизации. Например, учет может вестись в бумажных журналах. В автоматизированной информационной системе часть операций выполняется программными и техническими средствами, а часть решений принимает человек. Большинство современных организационных систем относится именно к автоматизированным, поскольку человек участвует в постановке задач, контроле, подтверждении операций и принятии решений. В автоматической системе операции выполняются без непосредственного участия человека в штатном режиме. Человек осуществляет настройку, наблюдение и вмешивается при возникновении исключительных ситуаций. Примерами могут служить автоматические системы управления технологическими процессами, системы мониторинга оборудования и автоматические торговые алгоритмы. По характеру обработки информации выделяются системы транзакционной обработки, информационно-поисковые, аналитические, экспертные и интеллектуальные системы. Системы обработки транзакций, или TPS, регистрируют и обрабатывают отдельные хозяйственные или операционные события. Для них важны скорость, надежность, целостность и согласованность данных. Информационно-поисковые системы обеспечивают хранение, индексирование и поиск документов или других информационных объектов. Примерами являются библиотечные каталоги, поисковые системы, архивы документов и базы нормативных актов. Аналитические информационные системы предназначены для исследования накопленных данных, выявления закономерностей, построения отчетов и прогнозов. Они могут включать хранилища данных, OLAP-технологии, панели показателей и средства бизнес-аналитики. Экспертные системы используют формализованные знания и правила для решения задач в определенной предметной области. Например, медицинская экспертная система может анализировать симптомы и предлагать возможные диагнозы, однако окончательное решение остается за специалистом. Интеллектуальные информационные системы применяют машинное обучение, обработку естественного языка, компьютерное зрение, рекомендательные алгоритмы и другие методы искусственного интеллекта. По режиму обработки данных системы могут быть пакетными, интерактивными и работающими в реальном времени. Пакетная обработка предполагает накопление данных и их обработку через определенные интервалы времени. Например, расчет заработной платы может выполняться один раз в месяц для всех сотрудников. Интерактивная обработка предполагает непосредственное взаимодействие пользователя с системой. Пользователь вводит данные и получает результат в процессе работы. Обработка в реальном времени требует реакции системы в строго ограниченный промежуток времени. Такие системы применяются в управлении оборудованием, транспорте, медицине, телекоммуникациях и финансовых операциях. Следует различать системы мягкого реального времени и жесткого реального времени. В системе мягкого реального времени задержка нежелательна, но не приводит к катастро-

фическим последствиям. Например, небольшая задержка видеотрансляции ухудшает качество обслуживания, но обычно не угрожает безопасности. В системе жесткого реального времени нарушение временного ограничения может привести к аварии или потере управления. Примером является система управления медицинским оборудованием или промышленным процессом. По способу организации вычислений выделяются централизованные, децентрализованные и распределенные информационные системы. В централизованной системе основные вычисления и хранение данных выполняются в одном центре. Пользовательские устройства обращаются к центральному серверу или вычислительному комплексу. Преимуществами являются простота управления, единообразие данных и централизованный контроль. Недостатками могут быть зависимость от центрального узла и ограниченная масштабируемость. В децентрализованной системе отдельные подразделения или узлы обладают высокой степенью самостоятельности. Они могут иметь собственные данные и приложения. В распределенной информационной системе компоненты размещаются на нескольких вычислительных узлах, взаимодействующих по сети. Пользователь может воспринимать такую систему как единое целое, хотя обработка выполняется в разных местах. Распределенная архитектура позволяет масштабировать нагрузку и повышать отказоустойчивость, но усложняет взаимодействие, согласование данных, мониторинг и безопасность. По типу пользователей можно выделить внутренние, внешние и смешанные информационные системы. Внутренняя система используется сотрудниками организации. Внешняя система предназначена для клиентов, партнеров или граждан. Смешанная система обслуживает и внутренние, и внешние группы пользователей. Например, банковская система включает внутреннюю часть, которой пользуются сотрудники, и внешние каналы — мобильное приложение, интернет-банк, банкоматы и интерфейсы для партнеров. По степени интеграции информационные системы делятся на изолированные, частично интегрированные и интегрированные. Изолированная система практически не обменивается данными с другими системами. Частично интегрированная система имеет ограниченное число интерфейсов. Интегрированная система включена в широкую информационную среду и взаимодействует с множеством внутренних и внешних приложений. По способу доступа выделяются локальные, сетевые, веб-ориентированные, мобильные и облачные системы. Локальная система устанавливается на отдельном компьютере или работает в локальной сети. Веб-ориентированная система доступна через браузер. Мобильная система использует приложения для смартфонов и планшетов. Облачная система развертывается на облачной инфраструктуре и предоставляется через сеть. Классификация информационных систем имеет практическое значение, поскольку разные классы систем предъявляют различные требования к архитектуре. Для небольшой локальной системы может быть достаточна простая монолитная архитектура. Для глобальной платежной платформы необходима сложная распределенная архитектура с высокой доступностью, масштабируемостью, безопасностью и устойчивостью к отказам.

1.3 Основные виды архитектуры информационной системы

Архитектура информационной системы является многомерным понятием. Систему невозможно полноценно описать только с одной позиции. Поэтому в процессе проектирования выделяются различные виды архитектуры, каждый из которых отражает определенную сторону системы. К основным видам относятся функциональная архитектура, системная архитектура, информационная архитектура, программная архитектура и архитектура данных. Границы между этими видами могут различаться в различных методологиях. Например, информационная архитектура и архитектура данных тесно связаны, а программная архитектура может рассматриваться как часть системной архитектуры. Однако их раздельное изучение позволяет более точно анализировать и проектировать систему.

1.3.1 Функциональная архитектура

Функциональная архитектура описывает функции, которые должна выполнять система, и отношения между этими функциями. Она отвечает прежде всего на вопрос: что должна делать система? На данном уровне внимание сосредоточено не на конкретных программах, серверах или базах данных, а на задачах, операциях, процессах и результатах деятельности. Функциональная архитектура формируется на основе целей организации и требований пользователей. Сначала определяется общая функция системы, затем она последовательно разделяется на подфункции. Такой процесс называется функциональной декомпозицией. Например, общей функцией информационной системы интернет-магазина является обеспечение электронной продажи товаров. Эта функция может быть разделена на управление каталогом, поиск товаров, управление корзиной, оформление заказа, прием оплаты, управление складскими остатками, организацию доставки, возврат товаров и формирование аналитической отчетности. Функция «оформление заказа» может быть дополнительно разделена на идентификацию покупателя, проверку содержимого корзины, расчет стоимости, выбор способа доставки, выбор способа оплаты, подтверждение заказа и формирование уведомления. Каждая функция получает определенные входные данные, выполняет преобразование и формирует выходной результат. Функциональная архитектура может описывать функции разного уровня. Стратегические функции связаны с достижением целей организации. Управленческие функции обеспечивают планирование, контроль и принятие решений. Основные функции создают ценность для клиентов или пользователей. Обеспечивающие функции поддерживают деятельность системы, например управление пользователями, аудит, резервное копирование и мониторинг. Функциональное представление важно потому, что оно позволяет отделить потребности бизнеса от способов технической реализации. Одна и та же функция может быть реализована различными технологиями. Например, функция уведомления клиента может выполняться посредством электронной почты, SMS, мобильного уведомления или сообщения в личном кабинете. Функциональная архитектура фиксирует необходимость уведомления, а конкретные каналы определяются на последующих уровнях проектирования. При разработке функциональной архитектуры используются модели бизнес-процессов, диаграммы потоков данных, диаграммы вариантов использования, функциональные модели и другие средства. Важным результатом является определение границ системы: какие функции выполняются самой системой, какие остаются за человеком, а какие передаются внешним системам. Например, информационная система медицинской клиники может регистрировать пациента, хранить историю обращений, планировать прием, передавать результаты анализов и формировать счета. Однако постановка диагноза остается функцией врача, хотя система может предоставлять ему аналитическую поддержку. Функциональная архитектура должна обеспечивать полноту и непротиворечивость функций. Необходимо исключить ситуации, когда важная функция не учтена или одна функция необоснованно дублируется в нескольких подсистемах. Также следует определить зави-

симости между функциями, последовательность их выполнения и общие правила обработки исключительных ситуаций.

1.3.2 Системная архитектура

Системная архитектура описывает информационную систему как целостную совокупность взаимосвязанных элементов. Она охватывает программные, технические, информационные, организационные и человеческие компоненты. Системная архитектура отвечает на вопрос: из каких крупных элементов состоит система и как они совместно обеспечивают выполнение ее целей? Если функциональная архитектура показывает, что система должна делать, то системная архитектура определяет, какими типами элементов будут реализованы функции. Например, одна функция может выполняться программным сервисом, другая — сотрудником организации, третья — внешней системой, четвертая — специализированным устройством. Системная архитектура включает подсистемы, компоненты, технические устройства, каналы связи, внешние системы, пользователей и организационные единицы. Она определяет границы системы и ее окружение. Например, системная архитектура автоматизированной системы управления складом может включать центральное приложение, базу данных, мобильные терминалы сотрудников, сканеры штрихкодов, принтеры этикеток, систему управления транспортом, бухгалтерскую систему, серверы, беспроводную сеть и персонал склада. Системная архитектура показывает, каким образом данные поступают от устройств, обрабатываются программными компонентами, сохраняются в хранилище и используются сотрудниками. Она также отражает взаимодействие с внешними организациями, например поставщиками и транспортными компаниями. В системной архитектуре может выделяться несколько уровней. Контекстный уровень показывает систему и внешнюю среду. Уровень подсистем отражает крупные части решения. Уровень компонентов раскрывает внутреннее устройство подсистем. Физический уровень показывает размещение компонентов на вычислительных узлах. При проектировании системной архитектуры учитываются не только функции, но и ограничения. К ограничениям относятся существующее оборудование, ранее внедренные системы, используемые стандарты, бюджет, сроки, квалификация персонала, законодательные требования и политика безопасности. Например, организация может требовать размещения данных исключительно на собственной инфраструктуре. Это ограничивает возможность использования публичных облачных сервисов. В другом случае система должна быть развернута в нескольких географических регионах, что влияет на распределение компонентов и репликацию данных. Системная архитектура особенно важна для сложных автоматизированных систем, в которых программное обеспечение взаимодействует с оборудованием и персоналом. В таких системах ошибка архитектурного решения может привести не только к снижению производительности, но и к нарушению технологического процесса.

1.3.3 Информационная архитектура

Информационная архитектура описывает информационное содержание системы, информационные потоки, источники и получателей информации, способы представления информации и правила ее использования. Она отвечает на вопросы: какая информация необходима системе, откуда она поступает, каким образом преобразуется, кому передается и в какой форме представляется? Информационная архитектура шире архитектуры данных. Архитектура данных преимущественно сосредоточена на моделях, структурах, хранилищах и технологиях управления данными. Информационная архитектура также учитывает смысл информации, ее связь с деятельностью пользователей, документы, сообщения, отчеты, информационные потоки и способы навигации. Например, в системе университета используются данные о студентах, преподавателях, дисциплинах, учебных планах, расписании, посещаемости и оценках. Однако информационная архитектура рассматривает не только таблицы базы данных. Она определяет, какие сведения требуются деканату, преподавателю, студенту, бухгалтерии и руководству; какие отчеты формируются; как информация передается между подразделениями;

кто имеет право изменять данные; какие документы являются официальными. Информационная архитектура включает понятия информационного объекта, информационного потока, источника информации, получателя информации, жизненного цикла информации и правил доступа. Информационный объект представляет собой осмысленную единицу информации, используемую в деятельности. Это может быть заказ, договор, счет, медицинская карта, заявление, учебный план, отчет или электронное сообщение. Информационный поток представляет собой движение информации между участниками, процессами или системами. Например, заявка клиента поступает в систему, передается на проверку, затем направляется исполнителю, после чего информация о результате возвращается клиенту. Жизненный цикл информации включает создание, регистрацию, проверку, использование, изменение, архивирование и удаление. Для каждого этапа могут действовать различные правила. Например, черновик документа может свободно изменяться автором, утвержденный документ должен быть защищен от несанкционированного редактирования, а архивный документ хранится установленный срок. Информационная архитектура должна обеспечивать логичность, понятность и непротиворечивость информационной среды. Пользователь должен понимать, где найти нужную информацию, как она организована и насколько ей можно доверять. В веб-системах понятие информационной архитектуры часто связывают со структурой разделов, навигацией, классификацией контента и поиском. Например, информационная архитектура интернет-магазина определяет категории и подкатегории товаров, фильтры, карточки товаров, структуру каталога и связи между объектами. Ошибки в такой архитектуре могут привести к тому, что пользователь не сможет найти нужный товар даже при наличии корректной программной реализации.

1.3.4 Архитектура данных

Архитектура данных определяет принципы организации, хранения, обработки, интеграции, защиты и управления данными. Она отвечает на вопросы: какие данные существуют в системе, как они структурированы, где хранятся, каким образом связаны, кто отвечает за их качество и как обеспечивается их доступность? Данные являются одним из наиболее устойчивых активов информационной системы. Программы и технологии могут изменяться, но накопленные данные часто должны сохраняться десятилетиями. Поэтому ошибки в архитектуре данных имеют длительные последствия. Архитектура данных включает концептуальные, логические и физические модели данных. Концептуальная модель данных отражает основные сущности предметной области и связи между ними без привязки к конкретной технологии. Например, для университета такими сущностями могут быть студент, преподаватель, дисциплина, учебная группа, занятие и оценка. Логическая модель данных уточняет атрибуты сущностей, ключи, связи и ограничения. На этом уровне определяется, что студент имеет идентификатор, фамилию, имя, дату рождения, статус обучения и связан с учебной группой. Физическая модель данных описывает реализацию в конкретной системе управления базами данных. Определяются таблицы, столбцы, типы данных, индексы, ограничения, способы распределения и хранения. Архитектура данных определяет выбор типов хранилищ. Реляционные базы данных организуют данные в виде таблиц и обеспечивают строгие связи и транзакции. Они хорошо подходят для финансовых, учетных и операционных систем. Документные базы данных хранят данные в виде документов и удобны для объектов с изменяющейся структурой. Графовые базы данных предназначены для хранения сложных сетей связей, например социальных отношений, маршрутов или зависимостей. Ключ-значение хранилища обеспечивают быстрый доступ по ключу и часто используются для кэширования. Колонночные хранилища эффективны для аналитической обработки больших объемов данных. В одной информационной системе могут использоваться разные типы хранилищ. Такой подход называется полиглотным хранением данных. Например, интернет-магазин может использовать реляционную базу для заказов и платежей, поисковый индекс для каталога, ключ-значение хранилище для пользовательских сессий и аналитическое хранилище для отчетности. Архитектура данных

также определяет подход к централизации и распределению. В централизованной модели данные хранятся в единой базе. Это упрощает обеспечение согласованности, но может создавать ограничения масштабирования и зависимость от одного хранилища. В распределенной модели данные размещаются в нескольких базах или узлах. Это может повышать производительность и отказоустойчивость, но усложняет синхронизацию и контроль целостности. Одной из ключевых проблем является согласованность данных. Если одна и та же информация хранится в нескольких местах, изменения должны корректно распространяться между ними. Например, изменение адреса клиента должно быть учтено в системе заказов, системе доставки и системе взаимоотношений с клиентами. Архитектура данных должна определять источник достоверных данных, или систему, которая считается официальным владельцем определенного вида информации. Например, сведения о сотрудниках могут официально храниться в кадровой системе, а данные о платежах — в финансовой системе. Важным понятием является качество данных. Качественные данные должны быть точными, полными, актуальными, непротиворечивыми, уникальными и пригодными для использования. Низкое качество данных приводит к ошибкам в отчетах, неправильным решениям и сбоям процессов. Архитектура данных включает механизмы управления метаданными. Метаданные — это данные о данных. Они описывают происхождение, структуру, смысл, формат, владельца и правила использования данных. Например, метаданные могут указывать, что поле «дата регистрации» содержит дату первого создания учетной записи и заполняется автоматически. Также архитектура данных охватывает вопросы безопасности. Необходимо определить классификацию данных, права доступа, шифрование, резервное копирование, архивирование и удаление. Особое внимание уделяется персональным, медицинским, финансовым и коммерчески значимым данным.

1.3.5 Программная архитектура

Программная архитектура описывает фундаментальную структуру программного обеспечения, его основные компоненты, интерфейсы, зависимости и способы взаимодействия. Она отвечает на вопрос: из каких программных элементов состоит система и как они совместно выполняют функции? Программная архитектура находится между требованиями и программным кодом. Она представляет систему на более высоком уровне абстракции, чем отдельные классы, функции или строки кода. К элементам программной архитектуры относятся приложения, модули, компоненты, сервисы, библиотеки, интерфейсы, базы данных, очереди сообщений, внешние API и механизмы интеграции. Программная архитектура может описываться на нескольких уровнях. На уровне системы определяются крупные приложения и сервисы. На уровне приложения выделяются модули и компоненты. На более детальном уровне описываются классы, интерфейсы и зависимости. Например, программная архитектура интернет-магазина может включать веб-интерфейс, мобильное приложение, серверную часть, модуль аутентификации, каталог, корзину, управление заказами, платежный модуль, модуль доставки, базу данных и систему уведомлений. Программная архитектура определяет способы взаимодействия компонентов. Взаимодействие может происходить через вызовы функций, API, обмен сообщениями, события, общие базы данных или файлы. Одним из важных свойств является связанность. Высокая связанность означает, что компоненты сильно зависят друг от друга. Изменение одного компонента может потребовать изменения многих других. Низкая связанность позволяет изменять компоненты более независимо. Другим свойством является сцепление, или внутренняя согласованность компонента. Хорошо спроектированный компонент объединяет функции, относящиеся к одной ответственности. Если в одном модуле смешаны обработка платежей, управление пользователями, формирование отчетов и отправка писем, такой модуль обладает низкой внутренней целостностью. Принцип высокой внутренней связности и низкой внешней связанности считается одним из базовых принципов программной архитектуры. Компонент должен отвечать за логически связанную область, но минимально зависеть от других компонентов. Программная архитектура влияет на возможность тестиро-

вания, сопровождения и развития системы. Если компоненты четко разделены и взаимодействуют через стабильные интерфейсы, разработчики могут изменять отдельные части без полной переработки системы.

1.3.6 Взаимосвязь различных видов архитектуры

Функциональная, системная, информационная, программная архитектура и архитектура данных не существуют изолированно. Они описывают одну систему с разных позиций и должны быть согласованы. Функциональная архитектура определяет, какие функции необходимы. Программная архитектура показывает, какие программные компоненты реализуют эти функции. Архитектура данных определяет, какие данные необходимы компонентам. Информационная архитектура показывает движение и использование информации. Системная архитектура объединяет программные, технические и организационные элементы. Например, функция «оформить заказ» может быть реализована программным компонентом управления заказами. Компонент использует данные о клиенте, товарах, цене и доставке. Информация поступает из пользовательского интерфейса и внешних сервисов. Компонент развернут на определенном сервере и взаимодействует с базой данных и платежной системой. Все эти аспекты должны быть согласованы. Если функциональная архитектура предусматривает обработку большого количества заказов, а системная архитектура предполагает один маломощный сервер, возникает несоответствие. Если программная архитектура разделяет систему на независимые сервисы, а архитектура данных требует постоянного совместного доступа к одной сложной структуре, также могут возникнуть проблемы. Поэтому проектирование архитектуры представляет собой итеративный процесс согласования различных представлений.

1.4 Монолитная архитектура

Монолитная архитектура представляет собой способ организации программной системы, при котором основные функциональные части объединены в единое приложение и обычно развертываются как одно целое. В монолите пользовательский интерфейс, бизнес-логика, работа с данными и другие функции могут быть частью одного программного продукта. Монолит не обязательно означает отсутствие внутренней структуры. Качественное монолитное приложение может быть разделено на модули, слои и компоненты. Ключевая особенность заключается в том, что эти элементы собираются, тестируются и развертываются совместно. Например, информационная система интернет-магазина в монолитной архитектуре может включать каталог, корзину, заказы, платежи, управление пользователями и отчеты в одном серверном приложении. Все части могут обращаться к единой базе данных и выполняться в одном процессе или в составе одного развертываемого пакета. Традиционный монолит часто строится по слоистой архитектуре. В нем выделяется слой представления, слой бизнес-логики и слой доступа к данным. Пользовательский запрос поступает в слой представления, затем передается в бизнес-логику, после чего выполняется обращение к базе данных. Преимуществом монолитной архитектуры является относительная простота начальной разработки. Все компоненты находятся в одном проекте, взаимодействие осуществляется через обычные внутрипроцессные вызовы, не требуется сложная сетевая инфраструктура. Монолит проще развертывать: необходимо установить и запустить одно приложение. Также проще обеспечить согласованные транзакции, поскольку все модули могут использовать единую базу данных. Отладка монолита часто проще, особенно на ранних этапах, поскольку запрос проходит внутри одного процесса. Не требуется анализировать множество сетевых вызовов и распределенных журналов. Монолитная архитектура может быть экономически оправданной для небольших и средних систем, ограниченных команд разработки и проектов с относительно стабильными требованиями. Однако по мере роста системы монолит может становиться сложным. Если внутренние границы модулей не соблюдаются, возникает неструктурированный монолит, в котором компоненты тесно связаны, используют общие данные и напрямую обращаются к внутренним функциям друг друга. В таком приложении изменение одной части может вызывать неожиданные последствия в других. Сборка и тестирование занимают больше времени, разработчики мешают друг другу, а внедрение небольшой функции требует развертывания всей системы. Проблемой может стать масштабирование. Монолит обычно масштабируется целиком. Если высокая нагрузка возникает только на каталог товаров, приходится создавать дополнительные экземпляры всего приложения, включая редко используемые модули. Кроме того, монолит усложняет использование разных технологий. Все части приложения обычно создаются на едином технологическом стеке. Переход на новую платформу может потребовать переработки значительной части системы. Следует различать модульный монолит и неструктурированный монолит. Модульный монолит имеет четкие внутренние границы, отдельные модули, контролируемые зависимости и стабильные интерфейсы. Такой вариант может быть эффективным и поддерживаемым даже для достаточно крупной системы. Например, модульный монолит интернет-магазина может включать независимые модули каталога, заказов, клиентов и платежей. Они находятся в одном приложении, но не обращаются напрямую к внутренним таблицам и классам друг друга. Взаимодействие происходит через определенные интерфейсы. Модульный монолит часто рассматривается как разумная отправная точка. Он позволяет быстрее создать систему, проверить бизнес-модель и определить реальные границы предметной области. Если отдельные модули начинают существенно отличаться по нагрузке или требованиям, их можно постепенно выделять в отдельные сервисы.

1.5 Микросервисная архитектура

Микросервисная архитектура представляет собой архитектурный подход, при котором система разделяется на набор небольших автономных сервисов. Каждый сервис реализует определенную бизнес-возможность, имеет четкую ответственность и взаимодействует с другими сервисами через сетевые интерфейсы. Например, интернет-магазин может быть разделен на сервис каталога, сервис заказов, сервис пользователей, сервис платежей, сервис склада, сервис доставки и сервис уведомлений. Каждый микросервис может иметь собственный программный код, собственную базу данных, отдельный цикл разработки и отдельное развертывание. Команда может изменять и выпускать сервис независимо от остальных, если сохраняется совместимость интерфейсов. Ключевой принцип микросервисной архитектуры заключается в разделении по бизнес-возможностям, а не только по техническим слоям. Например, сервис заказов должен включать логику, связанную с заказами, а не представлять собой общий слой доступа к данным для всей системы. Другим важным принципом является автономность сервисов. Сервис должен обладать достаточной самостоятельностью и минимально зависеть от внутренних деталей других сервисов. В микросервисной архитектуре взаимодействие происходит через сеть. Для синхронного взаимодействия могут использоваться HTTP API, REST, gRPC и другие протоколы. Для асинхронного взаимодействия используются очереди сообщений и события. Например, после создания заказа сервис заказов может опубликовать событие «Заказ создан». Сервис склада уменьшит остатки, сервис уведомлений отправит сообщение клиенту, а аналитическая система зарегистрирует событие для отчетности. Преимуществом микросервисов является независимое развертывание. Изменение сервиса уведомлений не требует обязательного развертывания сервиса заказов или каталога. Микросервисы позволяют независимо масштабировать части системы. Если сервис поиска испытывает высокую нагрузку, можно увеличить только число его экземпляров. Разные сервисы могут использовать различные технологии и типы баз данных, если это оправдано задачами. Например, сервис заказов может использовать реляционную базу данных, а сервис рекомендаций — графовое или аналитическое хранилище. Микросервисная архитектура также позволяет распределить ответственность между командами. Каждая команда может владеть определенной группой сервисов и отвечать за их разработку, тестирование и эксплуатацию. Отказ одного сервиса не обязательно приводит к полному отказу системы. Например, при временной недоступности сервиса рекомендаций интернет-магазин может продолжать оформлять заказы без персональных предложений. Однако такая устойчивость возникает только при специально спроектированной обработке отказов. Микросервисная архитектура имеет и существенные недостатки. Основной проблемой является распределенная сложность. Взаимодействие по сети менее надежно, чем вызов функции внутри одного процесса. Запрос может задержаться, потеряться, выполниться несколько раз или завершиться частично. Необходимо учитывать сетевые задержки, тайм-ауты, повторные попытки, балансировку нагрузки, обнаружение сервисов, версионирование API и защиту каналов связи. Сложнее обеспечить целостность данных. В монолите операция может выполняться в рамках одной транзакции базы данных. В микросервисной системе данные распределены между несколькими сервисами, и единая транзакция может быть невозможна или нежелательна. Например, оформление заказа включает резервирование товара, списание средств и создание доставки. Если платеж выполнен, но резервирование товара завершилось ошибкой, система должна корректно обработать частичный результат. Для этого используются распределенные процессы, компенсационные операции и шаблоны типа Saga. Возрастает сложность тестирования. Необходимо проверять отдельные сервисы, их интерфейсы, совместимость версий и совместное поведение системы. Эксплуатация требует развитой инфраструктуры. Необходимы автоматическое развертывание, централизованное

журналирование, мониторинг, трассировка запросов, управление конфигурациями, контейнеризация и оркестрация. Большое количество сервисов увеличивает число сетевых соединений, журналов, версий и точек отказа. Без зрелых инженерных процессов микросервисная архитектура может оказаться значительно сложнее монолита. Особенно опасно создавать чрезмерно мелкие сервисы. Если каждое простое действие требует последовательного обращения к множеству сервисов, производительность снижается, а система становится трудноуправляемой. Размер микросервиса определяется не количеством строк кода, а целостностью его бизнес-ответственности и возможностью независимого изменения.

1.6 Сравнение монолитной и микросервисной архитектуры

Выбор между монолитом и микросервисами нельзя осуществлять на основе представления о том, что микросервисы являются более современными и поэтому всегда лучшими. Архитектура должна соответствовать масштабу и задачам системы. Для небольшого проекта монолит часто оказывается предпочтительным. Он проще в разработке, тестировании, развертывании и сопровождении. Если над системой работает небольшая команда и отсутствуют экстремальные нагрузки, микросервисы могут создать неоправданную сложность. Микросервисы оправданы, когда система имеет значительный масштаб, разные части обладают различными требованиями к нагрузке, множество команд должны независимо развивать функциональность, а организация располагает зрелой инфраструктурой автоматизации. Монолит обеспечивает простое внутрипроцессное взаимодействие, тогда как микросервисы используют сетевые вызовы. Монолит обычно работает с единой базой данных, тогда как микросервисы стремятся владеть собственными данными. Монолит развертывается целиком, а микросервисы — независимо. Монолит масштабируется как единое приложение, а микросервисы позволяют масштабировать отдельные функции. Однако модульный монолит может сочетать простоту эксплуатации с качественным разделением ответственности. Поэтому на начальном этапе часто целесообразно создать модульный монолит, а затем выделять сервисы только при наличии реальной необходимости. Например, стартап разрабатывает новую платформу бронирования. На начальном этапе требования быстро меняются, предметная область еще недостаточно понятна, а команда состоит из пяти разработчиков. Создание двадцати микросервисов усложнит работу. Модульный монолит позволит быстрее проверить продукт. Позже, если модуль поиска будет испытывать высокую нагрузку, а платежный модуль потребует особых требований безопасности, их можно выделить в отдельные сервисы.

1.7 Архитектурные образцы

Архитектурный образец, или архитектурный паттерн, представляет собой обобщенное повторно применяемое решение типовой проблемы проектирования. Паттерн не является готовой программой или точной схемой. Он описывает общий принцип организации элементов, условия применения, преимущества, ограничения и последствия. Архитектурные образцы формируются на основе опыта создания множества систем. Они позволяют не разрабатывать каждое решение с нуля, а использовать проверенные подходы. Однако паттерн нельзя применять механически. Необходимо учитывать контекст и требования конкретной системы. Одним из наиболее известных является слоистый архитектурный паттерн. Система разделяется на уровни, каждый из которых выполняет определенную ответственность. Например, выделяются слой представления, слой бизнес-логики, слой доступа к данным и слой хранения данных. Слой представления взаимодействует с пользователем. Слой бизнес-логики реализует правила предметной области. Слой доступа к данным выполняет запросы к хранилищу. Такое разделение упрощает понимание и сопровождение системы. Недостатком слоистой архитектуры может быть прохождение каждого запроса через множество уровней и появление излишних зависимостей. Кроме того, при неправильном проектировании слой бизнес-логики может превратиться в набор процедур, тесно связанных с базой данных. Клиент-серверная архитектура разделяет систему на клиентов, запрашивающих услуги, и серверы, предоставляющие их. Клиент отвечает за взаимодействие с пользователем, сервер — за обработку запросов и управление ресурсами. В двухзвенной архитектуре клиент напрямую взаимодействует с сервером базы данных или серверным приложением. В трехзвенной архитектуре между клиентом и базой данных располагается сервер приложений. Многоуровневая архитектура включает дополнительные уровни и сервисы. Архитектура модель—представление—контроллер, или MVC, разделяет приложение на модель, представление и контроллер. Модель хранит состояние и бизнес-логику, представление отображает данные, контроллер обрабатывает действия пользователя и координирует взаимодействие. MVC особенно распространен в веб-разработке. Например, при открытии страницы товара контроллер принимает запрос, обращается к модели, получает данные и передает их представлению для формирования страницы. Сервисно-ориентированная архитектура, или SOA, организует систему как совокупность сервисов, предоставляющих функции через стандартизированные интерфейсы. Сервисы могут использоваться различными приложениями и бизнес-процессами. SOA часто применяется в крупных организациях для интеграции разнородных систем. Например, сервис проверки клиента может использоваться банковской системой, мобильным приложением и системой кредитования. Микросервисная архитектура имеет общие идеи с SOA, но обычно предполагает более мелкие автономные сервисы, независимое развертывание, децентрализованное управление данными и тесную связь сервисов с отдельными бизнес-возможностями. Событийно-ориентированная архитектура строится вокруг событий. Компонент публикует событие о произошедшем факте, а другие компоненты реагируют на него. Отправитель может не знать, какие получатели обработают событие. Например, событие «Платеж подтвержден» может быть обработано сервисом заказов, системой учета, сервисом уведомлений и аналитической платформой. Событийная архитектура снижает прямую связанность компонентов и хорошо подходит для асинхронных процессов. Однако она усложняет отслеживание последовательности операций, обработку ошибок и обеспечение согласованности данных. Архитектура каналов и фильтров представляет обработку данных как последовательность преобразований. Каждый фильтр выполняет отдельную операцию, а каналы передают данные между фильтрами. Такой паттерн применяется в компиляторах, системах обработки изображений, потоковой аналитике и интеграции данных. Например, поток документов может пройти стадии загрузки, проверки

формата, извлечения текста, классификации и сохранения. Архитектура репозитория предполагает наличие центрального хранилища, к которому обращаются различные компоненты. Общая база данных является примером репозитория. Преимуществом является единообразие данных. Недостатком — сильная зависимость компонентов от структуры хранилища. Архитектура брокера используется в распределенных системах для организации взаимодействия между компонентами через посредника. Брокер принимает запросы, определяет получателя и передает сообщения. Примерами являются брокеры сообщений и интеграционные шины. Архитектура микрокернела, или архитектура подключаемых модулей, состоит из минимального ядра и расширений. Ядро реализует базовые функции, а дополнительные возможности подключаются как плагины. Такой подход используется в интегрированных средах разработки, браузерах, системах управления контентом и корпоративных платформах. Гексагональная архитектура, также называемая архитектурой портов и адаптеров, отделяет бизнес-логику от внешних технологий. Центральная часть системы содержит правила предметной области. Взаимодействие с базами данных, пользовательским интерфейсом, внешними сервисами и очередями осуществляется через порты и адаптеры. Благодаря этому бизнес-логика меньше зависит от конкретной базы данных или веб-фреймворка. Например, вместо прямого обращения к определенной СУБД бизнес-компонент использует абстрактный интерфейс хранения, а конкретный адаптер реализует этот интерфейс. Чистая архитектура и луковичная архитектура развивают похожие идеи. Основные правила предметной области размещаются в центре, а технические детали находятся на внешних уровнях. Зависимости направлены внутрь, к более стабильной бизнес-логике. Архитектурные паттерны могут комбинироваться. Например, микросервисная система может использовать событийное взаимодействие, а каждый сервис внутри может быть построен по гексагональной или слоистой архитектуре.

1.7.1 Отличие архитектурного паттерна от шаблона проектирования

Архитектурный паттерн необходимо отличать от шаблона проектирования. Архитектурный паттерн определяет крупномасштабную структуру системы или приложения. Шаблон проектирования решает более локальную задачу организации классов и объектов. Например, MVC является архитектурным паттерном, поскольку определяет структуру приложения. Шаблон «Фабрика» определяет способ создания объектов, а шаблон «Наблюдатель» — способ уведомления зависимых объектов об изменениях. Граница между уровнями может быть условной, однако архитектурные паттерны обычно оказывают влияние на всю систему и связаны с фундаментальными решениями.

1.8 Эталонные модели

Эталонная модель представляет собой абстрактную систему понятий, функций и отношений, описывающую определенную предметную или технологическую область. Она служит основой для понимания, классификации и сравнения систем. Эталонная модель не определяет конкретные программные продукты или технологии. Она описывает, какие сущности и виды взаимодействий существуют в рассматриваемой области. Одним из известных примеров является эталонная модель взаимодействия открытых систем OSI. Она делит сетевое взаимодействие на семь уровней: физический, канальный, сетевой, транспортный, сеансовый, представительный и прикладной. Модель OSI не является готовой сетевой архитектурой конкретной организации. Она предоставляет понятийную структуру, позволяющую понимать функции протоколов и их положение в системе взаимодействия. Другим примером является эталонная модель открытой распределенной обработки RM-ODP. Она предлагает описывать распределенную систему с нескольких точек зрения: организационной, информационной, вычислительной, инженерной и технологической. Организационная точка зрения описывает цели, правила, роли и процессы. Информационная точка зрения рассматривает структуру и смысл информации. Вычислительная точка зрения показывает функциональное разделение на взаимодействующие объекты. Инженерная точка зрения описывает механизмы распределенного взаимодействия. Технологическая точка зрения определяет конкретные технологии реализации. Эталонные модели помогают создать общий язык для участников проекта. Если все стороны используют согласованные понятия, уменьшается вероятность неоднозначного понимания.

1.9 Эталонные архитектуры

Эталонная архитектура представляет собой обобщенную архитектурную структуру для определенного класса систем. Она конкретнее эталонной модели и может содержать типовые компоненты, их обязанности, интерфейсы, потоки данных и рекомендуемые паттерны. Эталонная архитектура не является полностью готовым решением. Она служит основой, которую необходимо адаптировать к требованиям конкретного проекта. Например, эталонная архитектура системы электронной торговли может включать пользовательские каналы, управление каталогом, заказы, платежи, склад, доставку, управление клиентами, интеграционный слой, аналитическое хранилище и средства безопасности. При разработке конкретного интернет-магазина эта архитектура уточняется: выбираются конкретные технологии, определяются границы сервисов, модели данных и способы развертывания. Эталонные архитектуры позволяют повторно использовать накопленный опыт, снижать риски и обеспечивать единообразие решений. Они особенно полезны в крупных организациях, где создается множество похожих систем. Например, банк может разработать корпоративную эталонную архитектуру цифровых каналов. Все новые мобильные и веб-приложения должны использовать единые механизмы аутентификации, журналирования, интеграции и безопасности.

1.10 Эталонные варианты архитектур

Эталонный вариант архитектуры представляет собой более конкретный типовой вариант реализации эталонной архитектуры для определенного сценария, масштаба или набора требований. Если эталонная архитектура задает общую структуру, то эталонный вариант показывает один из допустимых способов ее практического применения. Например, для веб-системы могут быть предложены несколько эталонных вариантов. Первый вариант предназначен для небольшой нагрузки и включает один сервер приложений и одну базу данных. Второй вариант рассчитан на высокую доступность и включает балансировщик нагрузки, несколько экземпляров приложения и кластер базы данных. Третий вариант предназначен для глобальной системы и предусматривает развертывание в нескольких регионах, распределенное кэширование и репликацию данных. Эталонные варианты особенно распространены в облачных платформах. Поставщик облачных услуг может предлагать архитектурные варианты для веб-приложений, потоковой обработки данных, машинного обучения, резервного копирования и аварийного восстановления. При этом эталонный вариант не должен восприниматься как универсальная инструкция. Его необходимо проверять на соответствие требованиям безопасности, производительности, стоимости и нормативным ограничениям.

1.11 Соотношение паттерна, эталонной модели и эталонной архитектуры

Архитектурный паттерн, эталонная модель и эталонная архитектура различаются по уровню абстракции и назначению. Архитектурный паттерн описывает повторяющийся принцип решения проблемы. Например, слоистая архитектура или событийно-ориентированная архитектура. Эталонная модель описывает основные понятия и отношения в определенной области. Например, OSI или RM-ODP. Эталонная архитектура показывает типовую структуру класса систем. Она может использовать несколько паттернов и опираться на эталонную модель. Эталонный вариант архитектуры представляет конкретизированную конфигурацию, адаптированную к определенному сценарию. Например, модель OSI дает понятия уровней сетевого взаимодействия. Клиент-серверный паттерн задает общий способ взаимодействия. Эталонная архитектура корпоративной сети описывает типовые сегменты, сервисы и средства защиты. Эталонный вариант показывает конкретное размещение серверов, межсетевых экранов и сетевых зон для организации определенного масштаба.

1.12 Архитектурные структуры

Архитектурная структура представляет собой способ организации архитектурных элементов и отношений между ними. Одна система может иметь множество структур, поскольку различные задачи требуют рассмотрения разных элементов и связей. Например, с точки зрения программного кода система состоит из модулей и зависимостей между ними. С точки зрения выполнения она состоит из процессов, потоков и каналов связи. С точки зрения развертывания она состоит из программных компонентов и вычислительных узлов. Каждая из этих структур отражает реальную сторону архитектуры. Обычно архитектурные структуры объединяются в несколько крупных групп: модульные структуры, структуры компонентов и соединителей и структуры распределения. Модульные структуры описывают организацию программного кода и единиц разработки. Элементами являются модули, пакеты, библиотеки, классы и подсистемы. Отношения показывают зависимости, использование, наследование и включение. Одной из модульных структур является структура декомпозиции. Она показывает, как система разделена на подсистемы и модули. Например, система может быть разделена на управление пользователями, каталог, заказы, платежи и отчеты. Другой модульной структурой является структура использования. Она показывает, какой модуль использует функции другого модуля. Такая структура помогает контролировать зависимости и предотвращать циклические связи. Структура слоев показывает распределение модулей по уровням абстракции. Верхние слои используют нижние, но нижние не должны зависеть от верхних. Структуры компонентов и соединителей описывают систему во время выполнения. Элементами являются процессы, сервисы, компоненты, базы данных, очереди и другие исполняемые единицы. Соединители представляют вызовы, сообщения, события, потоки данных и протоколы. Например, структура выполнения интернет-магазина может включать веб-клиент, API-шлюз, сервис заказов, платежный сервис, брокер сообщений и базу данных. Такая структура позволяет анализировать производительность, параллелизм, надежность и сетевое взаимодействие. Структуры распределения, или структуры размещения, показывают, как программные элементы связаны с окружением. К ним относится структура развертывания, которая отображает размещение компонентов на серверах, виртуальных машинах, контейнерах и сетевых узлах. Структура назначения работ показывает распределение модулей между командами разработки. Структура размещения файлов показывает организацию исходного кода и артефактов. Например, сервис заказов может быть развернут в трех контейнерах на двух вычислительных узлах, а база данных — на отдельном кластере. Такая информация относится к структуре развертывания. Архитектурные структуры помогают отвечать на разные вопросы. Структура модулей показывает, какие части кода придется изменить. Структура выполнения показывает, как пройдет запрос пользователя. Структура развертывания показывает, какие серверы будут задействованы и что произойдет при их отказе.

1.12.1 Архитектурные представления

Архитектурное представление является описанием системы с определенной позиции и для определенных заинтересованных сторон. Оно включает архитектурные элементы, отношения между ними и пояснения. Необходимо различать структуру, представление и точку зрения. Структура существует как организация элементов системы. Представление является ее документированным изображением или описанием. Точка зрения задает правила построения такого представления: какие элементы включать, какие обозначения использовать и какие вопросы решать. Например, структура развертывания существует в системе как реальное размещение программных компонентов. Диаграмма развертывания является архитектурным представлением. Набор правил создания диаграмм развертывания является архитектурной точкой зрения. Разные заинтересованные стороны нуждаются в разных представлениях.

Руководителю не требуется детальная схема классов. Ему важны крупные подсистемы, риски и затраты. Разработчику необходимы модули, интерфейсы и зависимости. Администратору требуется схема серверов, сетей и развертывания. Специалисту по безопасности важны границы доверия, потоки конфиденциальных данных и механизмы контроля доступа. Попытка создать одну универсальную диаграмму для всех участников приводит к перегруженному и малополезному документу. Поэтому архитектура описывается набором взаимосвязанных представлений. Одной из известных моделей является модель представлений «4+1». Она включает логическое представление, представление процессов, представление разработки, физическое представление и сценарии. Логическое представление описывает основные функциональные элементы и объекты предметной области. Оно показывает, каким образом система реализует требуемую функциональность. Представление процессов описывает выполняющиеся процессы, потоки, параллелизм, взаимодействие и синхронизацию. Оно важно для анализа производительности и надежности. Представление разработки показывает организацию программного кода, модулей, библиотек и пакетов. Оно ориентировано на разработчиков. Физическое представление описывает развертывание программных элементов на технической инфраструктуре. Оно важно для системных инженеров и администраторов. Сценарии, обозначаемые как «+1», демонстрируют, как элементы различных представлений взаимодействуют при выполнении конкретных пользовательских задач. Например, сценарий оформления заказа связывает пользовательский интерфейс, сервисы, процессы, данные и инфраструктуру. В современной практике также широко используется модель C4. Она предлагает четыре уровня: контекст системы, контейнеры, компоненты и код. Диаграмма контекста показывает систему, пользователей и внешние системы. Диаграмма контейнеров показывает крупные исполняемые приложения и хранилища данных. Диаграмма компонентов раскрывает внутреннее устройство контейнера. Диаграмма кода описывает классы и другие элементы реализации. Модель C4 позволяет постепенно увеличивать детализацию, не перегружая представления.

1.12.2 Требования к архитектурным представлениям

Архитектурные представления должны быть понятными, непротиворечивыми, актуальными и соответствовать целям аудитории. Каждая диаграмма должна отвечать на конкретный вопрос. Например, схема контекста должна показывать границы системы и внешние взаимодействия. Она не должна содержать сотни внутренних классов. Диаграмма развертывания должна показывать узлы и размещение компонентов, а не подробно описывать бизнес-процессы. Необходимо использовать согласованные обозначения. Если один и тот же компонент на разных диаграммах называется по-разному, возникает неоднозначность. Представления должны быть связаны между собой. Компонент, указанный на логической схеме, должен иметь соответствующее место в структуре реализации и развертывания. Важна актуальность документации. Архитектурная схема, не соответствующая реальной системе, может быть опаснее отсутствия схемы, поскольку вводит участников в заблуждение. При этом документация не должна становиться самоцелью. Необходимо создавать те представления, которые действительно помогают принимать решения, разрабатывать, тестировать и эксплуатировать систему.

1.13 Проектирование архитектуры «сверху вниз»

В исходной формулировке темы дважды указано проектирование «снизу вверх». В теории архитектуры обычно рассматриваются два взаимодополняющих подхода: проектирование «сверху вниз» и проектирование «снизу вверх». Для полного понимания проектирования архитектуры необходимо раскрыть оба подхода. Проектирование сверху вниз начинается с целей, требований и общего представления системы. Сначала система рассматривается как единое целое, затем последовательно разделяется на подсистемы, компоненты и более мелкие элементы. На первом этапе определяются цели организации и границы системы. Затем выявляются основные функции, информационные потоки и внешние взаимодействия. После этого функции распределяются между подсистемами, а подсистемы разбиваются на компоненты. Например, при проектировании системы университета сначала определяется общая цель — автоматизация образовательной деятельности. Затем выделяются крупные области: управление контингентом, учебные планы, расписание, успеваемость, электронное обучение и отчетность. Далее каждая область разделяется на более конкретные функции и модули. Подход сверху вниз обеспечивает соответствие архитектуры целям и требованиям. Он помогает избежать ситуации, когда система представляет собой случайный набор технологий и компонентов. Функциональная декомпозиция является характерным инструментом проектирования сверху вниз. Общая функция разделяется на подфункции до тех пор, пока элементы не станут достаточно конкретными для реализации. Преимуществом подхода является целостность. Архитектура формируется на основе общей картины, а локальные решения подчиняются системным целям. Однако чистое проектирование сверху вниз имеет ограничения. Архитектор может создать логически стройную модель, которая не учитывает реальные технологии, существующие системы, ограничения инфраструктуры и доступные компоненты. Например, архитектура может предполагать использование полностью независимых сервисов, но существующая база данных и организационная структура не позволяют разделить данные и ответственность. Поэтому проектирование сверху вниз должно сопровождаться анализом практических ограничений снизу.

1.14 Проектирование архитектуры «снизу вверх»

Проектирование снизу вверх начинается с анализа существующих компонентов, технологий, данных, систем, библиотек, оборудования и технических возможностей. Из отдельных элементов формируются более крупные подсистемы, а затем общая архитектура. Такой подход особенно характерен для проектов модернизации, интеграции и развития наследуемых систем. В организации уже могут существовать базы данных, приложения, серверы, интерфейсы и процессы. Архитектор не может игнорировать их и проектировать полностью новую систему без учета реального состояния. Например, предприятие планирует создать единую систему аналитики. Уже существуют бухгалтерская система, система управления складом, CRM, производственная система и множество файловых отчетов. Проектирование снизу вверх начинается с изучения источников данных, форматов, интерфейсов и качества информации. Затем создается интеграционная архитектура и аналитическое хранилище. Подход снизу вверх может использоваться и при создании новой системы, если применяются готовые платформы, библиотеки и облачные сервисы. Архитектор анализирует возможности доступных компонентов и на их основе формирует решение. Например, если облачная платформа предоставляет готовые сервисы аутентификации, хранения файлов, очередей сообщений и мониторинга, архитектура может быть построена с учетом этих сервисов. Преимуществом проектирования снизу вверх является реалистичность. Решение учитывает существующие ограничения и доступные ресурсы. Можно повторно использовать проверенные компоненты, сократить сроки и снизить затраты. Однако у подхода имеются риски. Если архитектура формируется только из существующих компонентов, она может не соответствовать целям бизнеса. Система превращается в набор технических решений без единой концепции. Например, организация может использовать определенную базу данных только потому, что она уже установлена, хотя новая система требует другого типа хранения. Или архитектура может строиться вокруг старого приложения, которое давно ограничивает развитие. Проектирование снизу вверх может приводить к локальной оптимизации. Каждый компонент хорошо решает свою задачу, но система в целом оказывается сложной, несогласованной и дорогой в сопровождении. Поэтому необходимо оценивать не только возможность повторного использования компонента, но и его влияние на долгосрочную архитектуру.

1.15 Сочетание подходов «сверху вниз» и «снизу вверх»

На практике архитектура редко проектируется исключительно одним способом. Наиболее эффективным является итеративное сочетание проектирования сверху вниз и снизу вверх. Сверху определяются цели, требования, функции, принципы и целевая структура. Снизу анализируются существующие компоненты, данные, технологии и ограничения. Затем результаты сопоставляются и уточняются. Например, проектирование сверху вниз показывает необходимость выделить сервис управления клиентами. Анализ снизу вверх обнаруживает, что в организации уже существует CRM-система с необходимыми данными и интерфейсами. Архитектура корректируется: вместо разработки нового сервиса используется интеграция с CRM. В другом случае существующая база данных может не обеспечивать требуемую масштабируемость. Тогда целевая архитектура сверху требует перехода на новое распределенное хранилище, несмотря на наличие старой системы. Процесс носит итеративный характер. Архитектор формирует предварительное решение, проверяет его на реализуемость, выявляет ограничения, корректирует структуру и повторно оценивает требования. Такой подход позволяет избежать двух крайностей. Первая крайность — идеальная, но нереализуемая архитектура, созданная без учета реальности. Вторая крайность — технически удобное, но стратегически бессистемное решение, собранное из доступных компонентов.

1.16 Этапы проектирования архитектуры информационной системы

Проектирование архитектуры начинается с анализа контекста. Необходимо определить цели организации, проблему, которую должна решить система, границы проекта, заинтересованные стороны и внешние системы. Следующим этапом является сбор и анализ требований. Выявляются функциональные требования, атрибуты качества, ограничения и риски. Особое внимание уделяется архитектурно значимым требованиям. Затем формируется функциональная модель. Определяются основные функции, процессы, роли пользователей и информационные потоки. После этого разрабатывается концептуальная архитектура. Система разделяется на крупные подсистемы и компоненты, определяются их обязанности и способы взаимодействия. Далее проектируется архитектура данных: выделяются основные сущности, источники данных, хранилища, потоки, правила качества и безопасности. На следующем этапе формируется программная архитектура. Выбираются архитектурные стили, паттерны, интерфейсы и механизмы интеграции. Затем проектируется системная и технологическая архитектура: определяются серверы, сети, облачные ресурсы, контейнеры, механизмы развертывания, резервирования и мониторинга. Полученное решение оценивается по атрибутам качества. Проверяется, обеспечивает ли архитектура требуемую производительность, надежность, безопасность и масштабируемость. Архитектура может проверяться с помощью сценариев. Например, рассматривается сценарий резкого увеличения нагрузки, отказа сервера, компрометации учетной записи, изменения бизнес-процесса или подключения новой внешней системы. Если архитектура не удовлетворяет требованиям, решения корректируются. Проектирование продолжается итеративно.

1.16.1 Архитектурные принципы

В процессе проектирования могут использоваться архитектурные принципы — общие правила, направляющие принятие решений. Одним из принципов является разделение ответственности. Каждый компонент должен иметь четко определенную роль. Принцип модульности предполагает разделение системы на относительно независимые части. Принцип минимальной связанности требует сокращения необязательных зависимостей между компонентами. Принцип скрытия информации означает, что компонент должен скрывать внутреннюю реализацию и предоставлять только необходимые интерфейсы. Принцип повторного использования предполагает применение общих компонентов и сервисов там, где это оправдано. Принцип простоты требует избегать неоправданной сложности. Архитектура должна быть настолько сложной, насколько это необходимо, но не сложнее. Принцип эволюционности предполагает возможность постепенного изменения системы. Принцип безопасности по проекту означает, что безопасность учитывается с самого начала, а не добавляется после завершения разработки. Принцип отказоустойчивости требует учитывать возможность ошибок и отказов компонентов. Принцип наблюдаемости предполагает наличие журналов, метрик, трассировки и средств диагностики. Архитектурные принципы должны быть конкретными и проверяемыми. Формулировка «система должна быть современной» не является полезным принципом. Более конкретным правилом будет требование использовать стандартизированные API для интеграции или обеспечивать независимое развертывание определенных компонентов.

1.16.2 Компромиссы в архитектуре

Архитектурное проектирование всегда связано с компромиссами. Улучшение одного свойства может ухудшить другое. Например, сильная нормализация базы данных уменьшает дублирование, но может увеличить количество соединений таблиц и снизить скорость аналитических запросов. Распределение системы повышает масштабируемость, но увеличивает сложность. Шифрование повышает безопасность, но требует дополнительных вычислительных

ресурсов. Кэширование увеличивает производительность, но создает риск использования устаревших данных. Строгая согласованность данных повышает надежность операций, но может снижать доступность распределенной системы. Архитектор должен не искать идеальное решение, а выбирать баланс, соответствующий приоритетам проекта. Например, для банковской операции целостность и согласованность важнее небольшой задержки. Для ленты новостей допустимо временное отображение немного устаревших данных ради высокой доступности. Все значимые компромиссы должны быть документированы. Необходимо фиксировать, какое решение принято, какие варианты рассматривались, почему выбран данный вариант и какие последствия он имеет.

1.16.3 Архитектурные риски

Архитектурный риск представляет собой возможность того, что принятое решение не позволит достичь требуемых характеристик системы. К рискам относятся недостаточная производительность, невозможность масштабирования, зависимость от одного поставщика, сложность интеграции, уязвимости безопасности, потеря данных, отсутствие компетенций и высокая стоимость эксплуатации. Например, использование новой недостаточно изученной технологии может ускорить разработку определенной функции, но создает риск отсутствия специалистов и трудностей сопровождения. Риск должен оцениваться по вероятности и возможным последствиям. Наиболее опасные решения следует проверять с помощью прототипов, нагрузочного тестирования и архитектурных экспериментов. Если неизвестно, сможет ли база данных обрабатывать необходимое число запросов, следует провести нагрузочное тестирование до окончательного утверждения архитектуры.

1.16.4 Документирование архитектурных решений

Архитектура должна быть документирована не только в виде схем, но и в виде решений. Для этого могут использоваться записи архитектурных решений, или ADR. Такая запись содержит контекст проблемы, рассматриваемые варианты, принятое решение, обоснование и последствия. Например, может быть зафиксировано решение использовать асинхронный обмен сообщениями между сервисами заказов и уведомлений. В обосновании указывается, что отправка уведомления не должна задерживать оформление заказа. В последствиях отмечается необходимость обработки повторных сообщений и мониторинга очереди. Документирование решений важно, поскольку через некоторое время участники могут забыть причины выбора. Без объяснения новое решение может показаться нелогичным и быть отменено, хотя оно было принято из-за существенных ограничений.

1.16.5 Архитектура как развивающаяся система

Архитектура информационной системы не является неизменной. Требования, технологии, нагрузка и внешняя среда меняются. Поэтому архитектура должна развиваться. На раннем этапе система может быть монолитной. По мере роста отдельные функции выделяются в сервисы. Локальная база данных может быть заменена кластером. Ручные процессы развертывания могут быть автоматизированы. Изменение архитектуры должно быть управляемым. Необходимо сохранять совместимость, мигрировать данные, обновлять документацию и контролировать риски. Архитектурный долг возникает, когда временные решения накапливаются и усложняют развитие. Например, быстрые прямые интеграции между системами могут привести к сети трудноуправляемых зависимостей. Архитектурный долг не всегда является ошибкой. Иногда временное решение оправдано сроками. Важно, чтобы его последствия были осознаны и учитывались в планах развития.

1.17 Значение архитектуры информационной системы

Архитектура обеспечивает связь между целями организации и технической реализацией. Без архитектуры разработка сложной информационной системы превращается в набор несогласованных локальных решений. Хорошая архитектура не гарантирует успех проекта, но плохая архитектура значительно повышает вероятность неудачи. Даже качественный программный код не сможет компенсировать фундаментальные ошибки в разделении ответственности, организации данных и выборе способов взаимодействия. Архитектура помогает управлять сложностью. Большая система разбивается на элементы, каждый из которых можно понимать, разрабатывать и тестировать отдельно. Она обеспечивает коммуникацию между участниками, позволяет оценивать риски, планировать развитие и контролировать качество. При этом архитектура не должна быть чрезмерно сложной или оторванной от реализации. Ее ценность определяется тем, насколько она помогает создавать, эксплуатировать и изменять систему. Таким образом, проектирование архитектуры информационной системы представляет собой систематический процесс определения функций, данных, компонентов, интерфейсов, технологий и принципов организации системы. В ходе проектирования учитываются цели, требования, ограничения, атрибуты качества и интересы заинтересованных сторон. Информационные системы могут классифицироваться по масштабу, назначению, уровню управления, степени автоматизации, характеру обработки данных, режиму работы и способу организации вычислений. Каждый класс систем предъявляет собственные требования к архитектуре. Функциональная архитектура описывает функции системы. Системная архитектура рассматривает совокупность программных, технических, информационных и организационных элементов. Информационная архитектура описывает смысл, движение и использование информации. Архитектура данных определяет структуры, хранилища и правила управления данными. Программная архитектура описывает программные компоненты и способы их взаимодействия. Монолитная архитектура объединяет функции в одном развертываемом приложении и отличается относительной простотой. Микросервисная архитектура разделяет систему на автономные сервисы и обеспечивает независимое развитие, но увеличивает распределенную сложность. Архитектурные паттерны предоставляют повторно используемые принципы решения типовых проблем. Эталонные модели формируют понятийную основу предметной области. Эталонные архитектуры описывают типовые структуры классов систем, а эталонные варианты предлагают конкретизированные способы реализации. Архитектурные структуры показывают различные способы организации элементов системы, а архитектурные представления документируют эти структуры для определенных заинтересованных сторон. Проектирование сверху вниз обеспечивает движение от целей и функций к компонентам. Проектирование снизу вверх использует существующие технологии, данные и компоненты. На практике оба подхода должны применяться совместно, обеспечивая одновременно целостность и реализуемость архитектуры.

2. Жизненный цикл информационных систем

Жизненный цикл информационной системы представляет собой упорядоченную совокупность состояний, процессов, работ и изменений, через которые проходит информационная система с момента возникновения потребности в ее создании до окончательного прекращения эксплуатации и вывода из использования. Жизненный цикл охватывает не только непосредственную разработку программного обеспечения, но и формирование замысла системы, изучение объекта автоматизации, определение требований, проектирование архитектуры, программную реализацию, испытания, внедрение, эксплуатацию, сопровождение, модернизацию, перенос данных, замену системы и ее окончательное снятие с эксплуатации. Понятие жизненного цикла позволяет рассматривать информационную систему не как однажды созданный и неизменный программный продукт, а как развивающийся объект. После внедрения система продолжает изменяться под воздействием новых требований, изменений законодательства, развития организации, увеличения объемов данных, появления новых технологий, выявления ошибок и изменения ожиданий пользователей. Поэтому жизненный цикл информационной системы обычно значительно продолжительнее периода ее первоначальной разработки. Например, разработка информационной системы университета может занимать два года, однако эксплуатация и сопровождение такой системы могут продолжаться десять или пятнадцать лет. За это время изменяются учебные планы, правила приема, формы отчетности, требования к защите персональных данных, способы взаимодействия со студентами и государственными информационными ресурсами. Следовательно, система должна не только быть создана, но и постоянно поддерживаться в актуальном состоянии. В широком смысле жизненный цикл начинается в тот момент, когда организация осознает наличие проблемы или потребности, которую предполагается решить с помощью информационных технологий. Например, руководство предприятия может установить, что существующий порядок учета складских запасов приводит к ошибкам, задержкам и избыточным закупкам. На этом этапе самой информационной системы еще не существует, но ее жизненный цикл уже начинается, поскольку формируется потребность, анализируются возможные решения и оценивается целесообразность автоматизации. Завершается жизненный цикл не простым прекращением запуска программы, а организованным снятием системы с эксплуатации. При этом необходимо сохранить юридически и организационно значимые данные, перенести информацию в новую систему, закрыть права доступа, прекратить интеграционное взаимодействие, удалить или архивировать конфиденциальные данные, обновить эксплуатационные регламенты и определить дальнейшую судьбу оборудования и программных компонентов. Жизненный цикл информационной системы необходимо отличать от жизненного цикла проекта. Проект имеет определенные цели, сроки, бюджет и момент завершения. Например, проект внедрения системы может завершиться после ее приемки заказчиком. Жизненный цикл самой системы при этом продолжается, поскольку начинается промышленная эксплуатация, сопровождение и последующее развитие. Следует также различать жизненный цикл информационной системы, жизненный цикл программного обеспечения и жизненный цикл программного продукта. Информационная система включает не только программы, но и данные, технические средства, пользователей, организационные процессы, документацию, средства связи и правила эксплуатации. Жизненный цикл программного обеспечения относится преимущественно к программным компонентам. Жизненный цикл программного продукта может дополнительно включать его продвижение, распространение, лицензирование, продажи, поддержку потребителей и прекращение коммерческого распространения. Например, бухгалтерская информационная система предприятия может включать прикладную программу, базу данных, серверное оборудование, рабочие места бухгалтеров, средства резервного копирования, правила разграничения доступа и регламенты подготовки

отчетности. Обновление программного компонента является частью жизненного цикла программного обеспечения, но переход организации на новые формы учета и изменение обязанностей сотрудников относятся к жизненному циклу информационной системы в целом. Основными категориями описания жизненного цикла являются стадия, этап, процесс, работа, задача, операция, результат, контрольная точка, итерация, инкремент, версия и выпуск. Стадия жизненного цикла представляет собой относительно крупную часть развития системы, объединенную общей целью и завершающуюся получением значимого результата. Например, стадиями могут быть формирование требований, проектирование, разработка, внедрение и эксплуатация. Этап является более детальной частью стадии. Например, стадия внедрения может включать подготовку объекта автоматизации, обучение персонала, предварительные испытания, опытную эксплуатацию и приемочные испытания. Процесс представляет собой совокупность взаимосвязанных действий, преобразующих определенные входные данные в результаты. Процесс управления требованиями преобразует потребности заинтересованных сторон в согласованные и контролируемые требования к системе. Процесс тестирования преобразует программный продукт, требования и тестовые данные в сведения о качестве и соответствии системы установленным требованиям. Работа объединяет несколько связанных задач, выполняемых для достижения определенного результата. Задача представляет собой конкретное действие или совокупность действий, закрепленных за исполнителем. Операция обычно рассматривается как наиболее детальная единица технологической деятельности. Контрольная точка обозначает момент, в котором оценивается достижение требуемого результата и принимается решение о дальнейшем движении проекта. В контрольной точке может утверждаться техническое задание, архитектура, опытный образец, версия программного продукта или готовность системы к эксплуатации. Итерация представляет собой повторное выполнение определенной последовательности работ с целью получения более полного и качественного результата. Каждая итерация уточняет требования, архитектуру, программные компоненты или другие элементы системы. Инкремент представляет собой функциональное приращение системы, то есть новую часть работоспособного продукта. Например, первым инкрементом системы университета может стать управление контингентом студентов, вторым — формирование расписания, третьим — учет успеваемости. Версия является определенным состоянием системы или программного продукта, имеющим идентифицируемый набор функций и изменений. Выпуск, или релиз, представляет собой версию, официально подготовленную для передачи пользователям или эксплуатации. Жизненный цикл описывается с помощью модели жизненного цикла. Такая модель определяет, как во времени организуются процессы, стадии и работы, в какой последовательности они выполняются, когда проверяются результаты, допускаются ли возвраты к предшествующим этапам, каким образом выпускаются версии системы и как учитываются риски. Необходимо различать модель жизненного цикла и методологию разработки. Модель жизненного цикла задает общую временную и логическую организацию работ. Методология определяет более конкретные принципы, роли, документы, методы и способы управления проектом. Например, каскадная и спиральная модели являются моделями жизненного цикла, а рациональный унифицированный процесс представляет собой более развернутую процессную технологию. Гибкие методы разработки также включают конкретные правила организации командной работы, планирования и обратной связи. Модель жизненного цикла не является описанием всех процессов организации. В пределах одной модели одновременно могут выполняться управление проектом, управление требованиями, проектирование, программирование, тестирование, управление конфигурацией, обеспечение качества, документирование, управление рисками и взаимодействие с заказчиком. Модель показывает общую организацию этих процессов во времени. Международный стандарт ISO/IEC/IEEE 12207:2026 устанавливает общую систему процессов жизненного цикла программных систем, продуктов и услуг. Он охватывает замысел, приобретение, поставку, разработку, эксплуата-

цию, поддержку и прекращение применения программного обеспечения. При этом стандарт прямо не предписывает единственную модель жизненного цикла: процессы могут выполняться параллельно, итерационно, рекурсивно и инкрементно в зависимости от особенностей проекта. Таким образом, жизненный цикл и модель жизненного цикла не являются тождественными понятиями. Жизненный цикл существует у каждой системы объективно, поскольку она создается, используется, изменяется и в определенный момент прекращает существование. Модель жизненного цикла является способом упорядочения и описания этого развития.

2.1 Содержание жизненного цикла информационной системы

Несмотря на различия конкретных моделей, в жизненном цикле информационной системы обычно присутствует несколько принципиальных областей деятельности. Первой является формирование потребности и концепции системы. На этом этапе определяется, почему требуется новая система, какие проблемы необходимо решить, какие цели должны быть достигнуты и какие заинтересованные стороны будут участвовать в ее создании и использовании. Например, причиной разработки медицинской информационной системы может быть необходимость сократить время оформления пациентов, обеспечить доступ врачей к истории болезни, исключить дублирование исследований и автоматизировать передачу сведений в государственные информационные ресурсы. На начальной стадии изучается существующее положение. Описываются процессы организации, используемые документы, информационные потоки, программные средства, техническая инфраструктура и существующие проблемы. Анализируется, можно ли решить проблему организационными изменениями, приобретением готового продукта, развитием существующей системы или созданием новой системы. Результатом концептуальной деятельности может стать обоснование необходимости разработки, концепция системы, предварительная оценка стоимости, перечень рисков, описание предполагаемого эффекта и решение о начале проекта. Следующей областью является формирование и управление требованиями. Требования определяют, какие функции должна выполнять система и какими свойствами она должна обладать. Они могут относиться к функциональности, производительности, надежности, безопасности, интерфейсам, данным, эксплуатации, сопровождению и нормативному соответствию. Требования не только выявляются, но и анализируются, согласовываются, документируются, проверяются и изменяются. В сложном проекте управление требованиями продолжается на протяжении всего жизненного цикла. Даже после внедрения возникают новые требования, которые становятся основанием для модернизации системы. Проектирование включает разработку архитектуры, структуры данных, программных компонентов, технической инфраструктуры, пользовательских интерфейсов, интеграционных механизмов и средств защиты. В процессе проектирования требования преобразуются в систему взаимосвязанных технических решений. Реализация включает программирование, настройку готовых программных продуктов, разработку баз данных, конфигурирование оборудования, создание интеграций и подготовку программной документации. Для информационной системы реализация может состоять не только в написании нового программного кода. Значительная часть работ может быть связана с настройкой стандартной платформы, переносом данных и объединением существующих систем. Верификация представляет собой проверку того, правильно ли создается продукт в соответствии с установленными спецификациями. Она отвечает на вопрос: соответствует ли результат предыдущим требованиям, моделям и проектным решениям? Валидация направлена на подтверждение того, что созданная система действительно пригодна для предполагаемого использования и удовлетворяет реальные потребности пользователей. Формально корректная система может пройти верификацию, но не пройти валидацию, если она неудобна, не поддерживает фактический рабочий процесс или не обеспечивает требуемого эффекта. Например, система может правильно рассчитывать показатели по утвержденным формулам, но требовать от пользователя ввода такого количества данных, что ее реальное применение становится неэффективным. В этом случае программная реализация соответствует спецификации, однако сама спецификация не полностью отражает потребности пользователей. Испытания включают проверку отдельных компонентов, интеграционного взаимодействия, полной системы, производительности, безопасности, устойчивости и удобства эксплуатации. Испытания могут проводиться разработчиками, независимой

группой контроля, заказчиком или приемочной комиссией. Внедрение охватывает установку системы, подготовку инфраструктуры, перенос данных, обучение пользователей, разработку организационных регламентов, опытную эксплуатацию и приемку. Внедрение нельзя сводить к копированию программы на сервер. Даже технически качественная система может оказаться неуспешной, если пользователи не обучены, данные перенесены с ошибками, обязанности сотрудников не определены или отсутствует поддержка руководства. Эксплуатация представляет собой использование системы по назначению. В процессе эксплуатации выполняются обработка данных, администрирование, управление пользователями, наблюдение за состоянием системы, резервное копирование, реагирование на сбои и обеспечение безопасности. Сопровождение заключается в контролируемом изменении системы после ее передачи в эксплуатацию. Сопровождение может включать исправление ошибок, адаптацию к новым условиям, повышение производительности, улучшение удобства использования и предотвращение будущих проблем. Обычно различают корректирующее сопровождение, направленное на исправление обнаруженных дефектов; адаптивное сопровождение, обеспечивающее работу в изменившейся технической или организационной среде; совершенствующее сопровождение, связанное с улучшением функций и характеристик; профилактическое сопровождение, направленное на предупреждение будущих дефектов и повышение сопровождаемости. Например, исправление ошибки расчета является корректирующим сопровождением. Переход на новую версию операционной системы относится к адаптивному сопровождению. Добавление нового аналитического отчета является совершенствующим сопровождением. Переработка сложного программного модуля для снижения вероятности последующих ошибок относится к профилактическому сопровождению. Модернизация может затрагивать отдельные функции или всю архитектуру системы. В определенный момент объем накопленных изменений становится настолько значительным, что вместо сопровождения требуется новый проект развития или полная замена системы. Снятие с эксплуатации включает принятие решения о прекращении использования, подготовку новой системы, перенос и архивирование данных, прекращение обслуживания старой системы, закрытие интеграционных интерфейсов и выполнение требований к хранению документации.

2.1.1 Принципы управления жизненным циклом

Управление жизненным циклом основывается на нескольких принципах. Принцип целостности требует рассматривать процессы разработки, внедрения, эксплуатации и сопровождения во взаимосвязи. Решение, принятое на ранней стадии, влияет на стоимость и сложность последующих стадий. Например, если при проектировании не предусмотрены средства журналирования и диагностики, эксплуатация и поиск причин отказов будут затруднены. Если не определена стратегия переноса данных, внедрение может задержаться. Если программный код не документирован, сопровождение потребует дополнительных ресурсов. Принцип прослеживаемости означает возможность установить связи между потребностями заинтересованных сторон, требованиями, проектными решениями, программными компонентами, тестами и результатами приемки. Благодаря прослеживаемости можно определить, каким требованием обусловлена конкретная функция и какими испытаниями подтверждается ее выполнение. Принцип управляемости изменений требует регистрации, анализа, согласования и контроля изменений. Неконтролируемое изменение требований приводит к росту сроков, стоимости и сложности системы. Принцип непрерывного обеспечения качества означает, что качество создается на всех стадиях, а не только проверяется в конце. Ошибка в исходных требованиях не может быть полностью устранена качественным программированием, поскольку разработчик может правильно реализовать неверно сформулированную функцию. Принцип управления конфигурацией обеспечивает идентификацию версий компонентов, документов и программных продуктов. Необходимо точно знать, какие версии исходного кода, библиотек, схем баз данных, настроек и документов входят в определенный выпуск системы. Принцип адапта-

ции процессов предполагает, что стандартная модель должна быть приспособлена к особенностям конкретного проекта. Небольшая внутренняя система и государственная информационная платформа не требуют одинакового объема документов, контроля и формальных процедур.

2.2 Каскадная модель жизненного цикла информационной системы

Каскадная модель, также называемая водопадной моделью, представляет жизненный цикл как последовательное прохождение заранее определенных стадий. Каждая стадия должна быть в основном завершена до начала следующей, а результат предыдущей стадии служит исходной основой для последующей. Классическая последовательность каскадной разработки может включать формирование и анализ требований, проектирование, реализацию, интеграцию, испытания, внедрение, эксплуатацию и сопровождение. Графически стадии располагаются сверху вниз, поэтому движение проекта напоминает поток воды, спускающийся по ступеням. Исторически каскадную модель часто связывают с работой Уинстона Ройса 1970 года. При этом важно учитывать, что Ройс не рекомендовал безусловно применять упрощенную однонаправленную схему. Он обращал внимание на риск позднего обнаружения проблем и предлагал дополнительные проверки, обратные связи, предварительное моделирование и создание пробной версии. Поэтому распространенное представление о том, что Ройс предложил абсолютно линейную разработку без возвратов, является упрощением его позиции. Основным организационным принципом каскадной модели является завершенность результата каждой стадии. Перед переходом к следующей стадии выполняются проверка, согласование и утверждение документации. Например, программирование начинается после утверждения проектных решений, а приемочные испытания проводятся после завершения реализации и внутреннего тестирования. На стадии анализа требований изучается предметная область, выявляются потребности пользователей и формируется согласованный набор требований. Результатом могут быть спецификация требований, техническое задание, модели бизнес-процессов и описание ограничений. На стадии проектирования разрабатываются архитектура, модели данных, структура программных компонентов, интерфейсы, алгоритмы и решения по технической инфраструктуре. Проектная документация должна содержать достаточно сведений для последующей реализации. На стадии реализации создаются программы, базы данных, настройки и технические компоненты. На стадии интеграции отдельные части объединяются в единую систему. Затем выполняются испытания, в ходе которых проверяется соответствие требованиям. После успешных испытаний система внедряется, передается в эксплуатацию и в дальнейшем сопровождается. В строгой каскадной модели предполагается, что требования достаточно полно известны до начала проектирования, проектные решения определены до программирования, а основная проверка готовой системы выполняется после завершения реализации. Такое построение удобно для календарного и ресурсного планирования, поскольку работы можно разделить на крупные этапы с определенными результатами, сроками и ответственными исполнителями. Каскадная модель тесно связана с документальным управлением проектом. Передача результатов между стадиями обычно сопровождается созданием формализованных документов. Техническое задание передается проектировщикам, проектная документация — разработчикам, программа и методика испытаний — испытательной группе, эксплуатационная документация — пользователям и сопровождающей организации. Одним из основных преимуществ каскадной модели является понятность организации работ. Последовательность стадий легко представить заказчику, руководителю и исполнителям. Для каждой стадии можно определить цели, состав работ, сроки, бюджет и результаты. Другим преимуществом является формальная контролируемость. Завершение стадий подтверждается экспертизой, утверждением документов и прохождением контрольных точек. Это важно в проектах, где требуется юридически значимая приемка, государственное финансирование, сертификация или строгая ответственность сторон. Каскадная модель обеспечивает высокий уровень документирования. Такая документация может быть особенно важна для систем длительной эксплуатации, ответственных систем,

проектов с большим числом участников и случаев, когда разработку и сопровождение выполняют разные организации. Модель может быть эффективна, если требования устойчивы, технология хорошо изучена, предметная область формализована, а стоимость изменений после начала реализации высока. Она применяется в проектах, где последовательность работ определяется не только разработкой программ, но и изготовлением оборудования, строительством объектов или прохождением обязательных процедур. Например, при создании программного обеспечения для уже спроектированного технического устройства набор функций, интерфейсов и ограничений может быть относительно стабилен. В таком случае подробное предварительное проектирование имеет большое значение. Однако каскадная модель имеет существенные ограничения. Первым является необходимость раннего определения требований. В реальных информационных системах пользователи часто не могут заранее полностью сформулировать свои потребности. Некоторые требования становятся понятны только после демонстрации работающей системы. Например, сотрудники организации могут согласовать письменное описание интерфейса, но только после начала опытной эксплуатации понять, что последовательность действий неудобна и не соответствует реальному рабочему процессу. Вторым ограничением является позднее получение работающего результата. Значительная часть проекта может быть завершена документально, однако пользователи увидят полноценную систему только после программирования и интеграции. Если исходные представления были ошибочными, проблема обнаружится слишком поздно. Третьим недостатком является высокая стоимость возврата. Ошибка в требованиях, найденная на стадии приемочных испытаний, может потребовать изменения архитектуры, модели данных, программного кода, тестов и документации. Четвертым ограничением является устаревание требований за время разработки. В длительном проекте могут измениться законодательство, структура организации, технологии, экономические условия и потребности пользователей. В результате система будет формально соответствовать первоначальному техническому заданию, но к моменту внедрения уже не полностью отвечать реальной ситуации. Пятым недостатком является ограниченная обратная связь пользователей. Пользователи участвуют в формировании требований и приемке, но могут быть недостаточно вовлечены в промежуточное уточнение продукта. Шестым является концентрация рисков в конце проекта. Интеграционные, эксплуатационные и пользовательские проблемы часто становятся заметны после создания основной части системы. Каскадная модель не исключает управление изменениями, однако любое изменение нарушает первоначальную последовательность и требует официального пересмотра результатов уже завершенных стадий. Чем позднее возникает изменение, тем дороже его реализация. Например, если после утверждения технического задания изменяется одно текстовое поле пользовательской формы, последствия могут быть небольшими. Если изменяется принцип расчета финансового показателя, это может потребовать пересмотра требований, архитектуры данных, алгоритмов, отчетов и тестовой документации. Следовательно, каскадная модель лучше всего подходит для проектов, в которых требования могут быть достаточно точно сформулированы заранее, изменения ограничены, технология известна, а строгий документальный контроль важнее скорости получения промежуточных версий.

2.3 Поэтапная модель жизненного цикла с промежуточным контролем

Поэтапная модель жизненного цикла с промежуточным контролем является развитием каскадного подхода. Она сохраняет последовательное деление проекта на стадии, но допускает организованные возвраты, уточнения и корректировки результатов. Эту модель также называют каскадной моделью с обратными связями, модифицированной каскадной моделью или поэтапной моделью с межстадийным контролем. Ее основная идея заключается в том, что результаты каждой стадии не только передаются дальше, но и проверяются с точки зрения требований предыдущих и последующих работ. В строгом каскаде движение изображается почти исключительно сверху вниз. В модели с промежуточным контролем между стадиями существуют обратные связи. Если на стадии проектирования обнаруживается неполнота требований, проект возвращается к их уточнению. Если в ходе программирования выявляется невозможность реализации определенного проектного решения, корректируется проект. Если тестирование обнаруживает ошибку в архитектуре, выполняется возврат к проектированию. Промежуточный контроль может осуществляться в форме анализа документации, технического совещания, экспертизы, демонстрации прототипа, проверки модели, контрольного тестирования, аудита качества или утверждения результата заказчиком. На границе стадий устанавливается контрольная точка качества. В ней оценивается полнота результатов, их непротиворечивость, соответствие требованиям, готовность к использованию на следующей стадии и приемлемость рисков. Например, перед началом программирования проверяется, определены ли интерфейсы компонентов, согласованы ли модели данных, описаны ли требования безопасности и подготовлены ли критерии испытаний. Если эти условия не выполнены, стадия проектирования не считается завершенной. Промежуточный контроль бывает внутристадийным и межстадийным. Внутристадийный контроль выполняется в процессе работы. Например, проектировщики проводят взаимную проверку архитектурных решений. Межстадийный контроль осуществляется перед передачей результата следующей группе. Важной особенностью модели является наличие не только обратной, но и опережающей связи. Исполнители последующей стадии могут заранее участвовать в проверке результатов предыдущей. Разработчики анализируют проектную документацию до ее утверждения, тестировщики участвуют в проверке требований, специалисты по эксплуатации оценивают архитектуру с точки зрения развертывания и сопровождения. Такой подход позволяет обнаруживать проблемы раньше. Если тестировщик участвует в анализе требований, он может указать, что определенное требование невозможно объективно проверить. Если администратор участвует в проектировании, он может заранее выявить отсутствие средств резервного копирования и мониторинга. Поэтапная модель с промежуточным контролем снижает вероятность накопления ошибок. В чистом каскаде ошибка может переходить из одного документа в другой и обнаружиться только в конце. При промежуточных проверках каждый результат анализируется до начала массового использования. Например, неверно определенная структура справочника может повлиять на десятки программных модулей. Если ошибка выявлена при проверке логической модели данных, исправляется только проект. Если она обнаружена после заполнения базы данных и разработки отчетов, потребуется значительная переработка. Модель повышает качество документации, поскольку документы рассматриваются не только как формальный результат, но и как объект проверки. Требования, модели и инструкции уточняются по результатам анализа. Другим преимуществом является более тесное взаимодействие участников. Заказчик, аналитики, архитекторы, разработчики, тестировщики и специалисты по эксплуатации включаются в контроль результатов. Однако промежуточный контроль увеличивает трудоемкость и продолжительность проекта. Необходимо планировать экспертизы, согласования

и повторное выполнение работ. Если каждый незначительный вопрос требует официального возврата и нового утверждения документов, процесс становится бюрократическим. Количество обратных связей должно быть управляемым. Неограниченные изменения могут привести к постоянному возвращению на предыдущие стадии и невозможности завершить проект. Поэтому определяются порядок внесения изменений, полномочия участников, критерии повторной проверки и влияние изменений на сроки и стоимость. Поэтапная модель с промежуточным контролем не превращается автоматически в итерационную или гибкую модель. Основная структура по-прежнему остается последовательной, а конечная система обычно поставляется после завершения основных стадий. Возвраты выполняются преимущественно для исправления и уточнения результатов, а не для регулярного выпуска функциональных приращений. В учебной литературе такую модель часто представляют как каскад, в котором от каждой стадии проведены обратные стрелки к предшествующим стадиям. На практике возврат не обязательно ограничивается соседней стадией. Ошибка, обнаруженная при эксплуатации, может потребовать пересмотра требований, архитектуры и организационной модели. Модель с промежуточным контролем отличается и от V-образной модели. В V-образной модели стадии определения и проектирования сопоставляются с соответствующими уровнями испытаний. Требования связаны с приемочными испытаниями, архитектура — с системными и интеграционными испытаниями, детальный проект — с тестированием компонентов. Поэтапная модель с промежуточным контролем делает акцент на проверках и возвратах между стадиями, но не обязательно устанавливает такое симметричное соответствие. Примером применения модели может быть создание корпоративной финансовой системы. После формирования требований проводится их экспертиза финансовыми специалистами и службой безопасности. После проектирования выполняется архитектурный контроль. В процессе программирования отдельные решения проверяются на соответствие проекту. Перед опытной эксплуатацией проводится интеграционное тестирование. Результаты опытной эксплуатации могут привести к возврату на стадию реализации или проектирования. Если выявлено неправильное расположение поля в форме, достаточно изменить пользовательский интерфейс. Если обнаружено, что неправильно определен процесс согласования платежа, может потребоваться пересмотр требований и функциональной архитектуры. Глубина возврата определяется причиной обнаруженного несоответствия. Поэтапная модель с промежуточным контролем эффективна, когда требуется сохранить формализованную стадийность, но невозможно предполагать абсолютную безошибочность ранних решений. Она особенно полезна для крупных проектов с разделением ответственности между организациями и необходимостью официальной приемки промежуточных результатов.

2.4 Стандартизация процессов разработки программ и программной документации

Стандартизация разработки программ и программной документации представляет собой установление и применение согласованных правил, требований, терминов, процессов, видов документов и способов контроля. Ее назначение состоит в том, чтобы сделать деятельность участников проекта понятной, воспроизводимой, управляемой и проверяемой. Без стандартизации разные участники могут по-разному понимать содержание стадий, назначение документов, критерии завершения работ и ответственность сторон. Заказчик может считать систему готовой после реализации функций, разработчик — после завершения программирования, а эксплуатационная организация — только после подготовки инструкций, мониторинга и резервного копирования. Стандарты формируют общий профессиональный язык. Они определяют такие понятия, как процесс, деятельность, задача, требование, верификация, валидация, конфигурация, сопровождение, эксплуатация и прекращение применения. Стандартизация выполняет организационную функцию, поскольку определяет порядок работ и взаимодействия участников. Техническая функция заключается в установлении требований к проектированию, разработке и испытаниям. Информационная функция обеспечивает единообразное представление результатов. Контрольная функция позволяет оценивать полноту и качество работ. Правовая функция проявляется в возможности использовать стандарты и связанные с ними документы при заключении и исполнении договоров. Стандарты жизненного цикла не следует воспринимать как подробную инструкцию по программированию. Они преимущественно определяют, какие процессы должны быть предусмотрены, какие цели они преследуют и какие результаты должны давать. Конкретные языки программирования, средства моделирования, алгоритмы и инструменты выбираются с учетом проекта. Международная стандартизация разделяет процессы жизненного цикла системы и процессы жизненного цикла программного обеспечения. ISO/IEC/IEEE 15288:2023 применяется к полному жизненному циклу систем, включая формирование замысла, разработку, производство, использование, поддержку и снятие с эксплуатации. Он рассматривает систему как целостный объект, включающий программные, технические, организационные и человеческие элементы. ISO/IEC/IEEE 12207:2026 устанавливает систему процессов жизненного цикла программных систем, продуктов и услуг. Он применяется к приобретению, поставке, разработке, эксплуатации, сопровождению и прекращению использования программного обеспечения. Стандарт допускает применение процессов в различных моделях жизненного цикла, включая последовательные, итерационные, инкрементные и гибкие подходы. Это означает, что стандарт процессов и модель жизненного цикла выполняют разные функции. Стандарт определяет состав и назначение процессов, а модель устанавливает их временную организацию. Один и тот же процесс тестирования может использоваться в каскадной, спиральной, инкрементной или гибкой разработке. В каскадной модели основное системное тестирование выполняется после реализации. В итерационной модели тестирование проводится в каждой итерации. В непрерывной разработке автоматические тесты запускаются при каждом изменении программного кода. Сам процесс тестирования сохраняется, но меняется его положение в жизненном цикле. В российской системе стандартизации применяется ГОСТ Р ИСО/МЭК 12207—2010 «Информационная технология. Системная и программная инженерия. Процессы жизненного цикла программных средств». Он был создан на основе международной редакции ISO/IEC 12207:2008. Международный стандарт позднее обновлялся, а в 2026 году опубликована новая редакция ISO/IEC/IEEE 12207:2026. Поэтому при практическом применении необходимо различать действующий российский национальный стандарт и актуальную международную редакцию. Важным российским нормативным документом является ГОСТ Р 59793—2021 «Информационные технологии. Комплекс стан-

дартов на автоматизированные системы. Автоматизированные системы. Стадии создания». Он устанавливает стадии и этапы создания автоматизированных систем. На территории Российской Федерации этот стандарт применяется вместо ГОСТ 34.601—90, который утратил силу с 1 декабря 2023 года. ГОСТ Р 59793—2021 предусматривает движение от формирования требований и разработки концепции к техническому заданию, проектированию, созданию рабочей документации, вводу системы в действие и сопровождению. Конкретный состав стадий и этапов может адаптироваться с учетом особенностей создаваемой системы. Стадия формирования требований к автоматизированной системе включает обследование объекта, обоснование необходимости создания системы, формирование пользовательских требований и оформление результатов. На стадии разработки концепции изучается объект автоматизации, при необходимости проводятся научно-исследовательские работы, формируются варианты концепции, выбирается предпочтительное решение и оцениваются риски проекта. Стадия технического задания завершается разработкой и утверждением документа, который устанавливает назначение, цели, требования, состав работ, порядок контроля и условия приемки системы. На проектных стадиях разрабатываются предварительные и окончательные решения по структуре системы, ее функциям, данным, техническим средствам, программному обеспечению, безопасности и организации эксплуатации. Стадия ввода в действие может включать подготовку объекта, обучение персонала, поставку и монтаж технических средств, загрузку данных, предварительные испытания, опытную эксплуатацию и приемочные испытания. Сопровождение включает работы по гарантийному и послегарантийному обслуживанию, устранению недостатков, поддержанию работоспособности и развитию системы. Отдельное направление стандартизации связано с Единой системой программной документации — ЕСПД. Она устанавливает виды программ и программных документов, стадии разработки, правила оформления, содержание технического задания, описание программы, программу и методику испытаний, эксплуатационные документы и другие результаты разработки. С 30 января 2025 года в России действует ГОСТ 19.101—2024 «Единая система программной документации. Виды программ и программных документов», заменивший ГОСТ 19.101—77. В 2026 году к новой редакции была введена поправка. Стандарт распространяется на программы для средств вычислительной техники и устанавливает виды программ и программных документов. ГОСТ 19.102—77 «Единая система программной документации. Стадии разработки» продолжает действовать и устанавливает стадии разработки программ и программной документации. В нем определены техническое задание, эскизный проект, технический проект, рабочий проект и внедрение. Стандарт допускает исключение или объединение отдельных стадий и этапов по согласованию с заказчиком, то есть даже традиционная нормативная схема предусматривает адаптацию к конкретному проекту. На стадии технического задания обосновывается необходимость разработки, формулируется задача, собираются исходные материалы, определяются требования, критерии качества, сроки, этапы и виды испытаний. Эскизный проект содержит предварительные решения по структуре входных и выходных данных, методам решения задачи и общему алгоритму. Технический проект уточняет данные, алгоритмы, структуру программы, формы представления информации и конфигурацию технических средств. Рабочий проект включает программирование, отладку, разработку программной документации и испытания. Внедрение предусматривает подготовку и передачу программы и документации для эксплуатации, сопровождения или изготовления. Система программной документации может включать техническое задание, спецификацию, текст программы, описание программы, пояснительную записку, программу и методику испытаний, описание применения, руководство пользователя, руководство оператора, руководство программиста, формуляр и другие документы. Назначение технического задания состоит в фиксации целей, назначения, требований, ограничений, стадий разработки и порядка приемки. Спецификация определяет состав программы и документации. Описание программы раскрывает логическую структуру и функционирование. Программа и

методика испытаний устанавливает, какие требования проверяются и какими способами. Эксплуатационные документы обеспечивают правильное использование, настройку и обслуживание системы. Международный стандарт ISO/IEC/IEEE 15289:2019 посвящен содержанию информационных объектов жизненного цикла, то есть документации и других зафиксированных результатов процессов. Он определяет назначение и содержание документов, создаваемых в ходе жизненного цикла систем и программного обеспечения. Современное понимание документации шире традиционного текстового документа. Информационным объектом жизненного цикла может быть модель, электронная запись, набор требований, журнал изменений, описание интерфейса, протокол испытаний, схема развертывания, программный код с комментариями или автоматически сформированный отчет. Стандартизация документации обеспечивает полноту, однозначность, согласованность, идентифицируемость, актуальность и прослеживаемость информации. Полнота означает наличие всех необходимых сведений. Однозначность исключает различные толкования. Согласованность требует отсутствия противоречий между документами. Идентифицируемость позволяет определить наименование, версию, автора и статус документа. Актуальность означает соответствие текущему состоянию системы. Прослеживаемость связывает документ с требованиями, решениями и результатами проверки. Большое значение имеет управление версиями документации. Техническое задание, проектная модель и руководство пользователя могут изменяться. Участники должны использовать утвержденные версии, соответствующие текущей конфигурации системы. Например, если тестировщик проверяет систему по устаревшей версии требований, результаты испытаний не будут достоверными. Если пользователь применяет старую инструкцию после изменения интерфейса, возрастает вероятность ошибок. Стандартизация не означает обязательного создания максимального количества документов. Состав документации должен соответствовать размеру, критичности и условиям проекта. Для небольшой внутренней программы может быть достаточно краткого описания требований, инструкции и набора тестов. Для государственной или промышленной системы потребуется значительно более формализованный комплект. Важным принципом является адаптация стандарта, то есть выбор применимых процессов, работ и документов. Адаптация должна быть обоснованной. Нельзя исключать испытания или управление конфигурацией только ради сокращения сроков, если это создает неприемлемые риски. Стандарты не отменяют профессиональное решение. Они создают основу, но конкретная организация определяет роли, инструменты, контрольные точки, степень детализации и способы автоматизации процессов.

2.5 Схема жизненного цикла больших программных комплексов по В. В. Липаеву

Владимир Васильевич Липаев является одним из основателей отечественной школы программной инженерии. В его работах жизненный цикл крупных программных комплексов рассматривается как сложный, повторяющийся и управляемый процесс, включающий не только первоначальную разработку, но и длительную эксплуатацию, сопровождение и модернизацию. Схема жизненного цикла больших программных комплексов по В. В. Липаеву создавалась с учетом особенностей крупных систем, разработка которых требует участия коллективов специалистов, значительных ресурсов, формального управления, документирования, испытаний и длительного сопровождения. Липаев подчеркивал различие между небольшими программами, которые могут создаваться одним разработчиком, и крупными программными комплексами. Большой программный комплекс имеет множество компонентов, высокую стоимость, длительный срок эксплуатации, существенные требования к надежности и сложные связи с внешними системами. Для него необходимы регламентированные процессы, управление конфигурацией, документирование и независимый контроль качества. В обобщенном виде схема включает системный анализ, проектирование и разработку, внедрение и эксплуатацию, а также сопровождение и модернизацию. Между этими частями существуют обратные связи, позволяющие исправлять локальные ошибки, пересматривать архитектуру и при необходимости начинать новый цикл системного анализа. Первой крупной областью является системный анализ. Он начинается с постановки задачи и изучения объекта, в котором предполагается использовать информационную систему. Анализируется текущее состояние организации, действующие процессы, информационные потребности, недостатки существующей системы и причины необходимости автоматизации. В ходе системного анализа определяется, какие цели должна обеспечивать будущая система, какие функции подлежат автоматизации, какие ограничения существуют и какой эффект предполагается получить. Например, при анализе деятельности склада изучаются порядок приема товаров, размещение, инвентаризация, комплектование заказов, выдача, возвраты и взаимодействие с бухгалтерией. Выявляются задержки, дублирование данных и ошибки ручного учета. На основе выявленных проблем формируется потребность в новой системе и выполняется технико-экономическое обоснование. Необходимо определить, даст ли автоматизация достаточный результат и соответствует ли предполагаемая стоимость ожидаемому эффекту. Далее выбирается направление совершенствования объекта. Может быть принято решение разработать новую систему, приобрести готовый программный продукт, модернизировать существующую систему или изменить организацию процессов. Фаза системного анализа завершается формированием и утверждением технического задания. Техническое задание связывает потребности организации с последующим проектированием. Оно определяет назначение, функции, требования, ограничения, состав работ и порядок приемки. Второй крупной областью является проектирование и разработка системы. На основе технического задания формируется функциональная архитектура, то есть состав функций, задач и функциональных подсистем. Функциональная архитектура отвечает на вопрос, какие виды деятельности должна поддерживать система. Например, в складской системе выделяются приемка, размещение, хранение, комплектование, отгрузка, инвентаризация и отчетность. После функциональной архитектуры разрабатывается системная архитектура. Она определяет состав программных, информационных, технических и организационных компонентов, а также связи между ними. На этом уровне устанавливается, какие функции реализуются программными модулями, какие данные хранятся в базе, какие устройства применяются, как осуществляется сетевое взаимодействие и какие обязанности выполняет персонал. Затем выполняется физическое проектирование и конструирование: разрабатываются программы, базы

данных, инструкции, средства взаимодействия и другие компоненты. Результатом становится программное изделие или программный комплекс, подготовленный для испытаний и внедрения. Третья область включает опытное внедрение и ввод в промышленную эксплуатацию. В ходе опытного внедрения проверяется работоспособность отдельных элементов и связей, выявляются локальные ошибки и уточняются технические решения. Если обнаруживается ошибка в конкретном программном компоненте, выполняется возврат к его разработке и исправлению. После устранения дефектов опытное внедрение повторяется. На следующем уровне проверяется система в целом. Оценивается соответствие функций требованиям заказчика, правильность взаимодействия подсистем, качество обработки данных и готовность к реальной эксплуатации. Если выясняется, что состав функций системы не соответствует деятельности организации, локального исправления программы недостаточно. Требуется возврат к функциональной архитектуре и повторное прохождение части проектных работ. После приемки начинается промышленная эксплуатация. Система используется по назначению, а разработчики и сопровождающая организация получают сведения о ее поведении в реальной среде. Эксплуатация является источником фактической информации о нагрузке, удобстве, надежности, ошибках и соответствии системы потребностям. Некоторые проблемы невозможно полностью выявить в лабораторных испытаниях, поскольку они проявляются только при длительной работе, большом объеме данных или участии множества пользователей. Четвертой областью является сопровождение. В процессе сопровождения анализируются сообщения пользователей, устраняются дефекты, адаптируется программное обеспечение, расширяются функции и совершенствуются характеристики. Если требуется небольшое изменение, оно выполняется на уровне программного компонента. Если изменяются функции, выполняется возврат к функциональному проектированию. Если изменяется техническая платформа или общий способ организации системы, пересматривается системная архитектура. В схеме выделяется несколько характерных циклов обратной связи. Первый охватывает весь путь от системного анализа до сопровождения и представляет собой цикл первоначального создания системы. Второй цикл возникает после опытного внедрения, когда выявляются частные ошибки элементов проекта. Исправление выполняется на уровне реализации, после чего опытное внедрение повторяется. Третий цикл возникает при выявлении после приемки ошибок функциональной архитектуры. В этом случае необходимо вернуться к определению состава подсистем, задач и связей между ними. Четвертый цикл возникает, когда новые условия требуют изменения системной архитектуры. Например, система должна быть перенесена на новую техническую платформу, объединена с другими системами или переведена на распределенное выполнение. Пятый, наиболее глубокий цикл возникает при моральном устаревании системы или полном несоответствии новым потребностям. Тогда требуется возвращение к постановке задачи и выполнение нового системного анализа. В учебных изложениях схема Липаева представляется как последовательность двенадцати взаимосвязанных блоков, объединяющих анализ объекта, обоснование автоматизации, техническое задание, функциональную и системную архитектуру, реализацию, опытное внедрение, исправление ошибок, промышленное внедрение, эксплуатацию и сопровождение. Главной особенностью схемы является не количество блоков, а наличие обратных связей разной глубины. Принципиально важной характеристикой является повторяемость последовательности «системный анализ — разработка — сопровождение — новый системный анализ». Информационная система рассматривается как динамический объект, который должен изменяться вместе с организацией и внешней средой. Например, первоначально информационная система банка может обслуживать только отделения. Затем появляется необходимость мобильного обслуживания. Если архитектура допускает развитие, создаются новые компоненты и интерфейсы. Если старая архитектура принципиально не поддерживает требуемый масштаб и безопасность, начинается новый цикл системного анализа и проектирования. Схема Липаева занимает промежуточное положение между строгой каскадной моделью и современ-

ными эволюционными подходами. В ней сохраняется последовательность основных фаз, но признается неизбежность возвратов и длительного развития системы.

2.6 Спиральная модель жизненного цикла информационных систем

Спиральная модель представляет собой итерационную модель жизненного цикла, в которой разработка организуется в виде последовательных циклов, а выбор содержания каждого цикла определяется анализом целей, альтернатив и рисков. Классическая спиральная модель была сформулирована Барри Боэмом. Ее принципиальным отличием является ориентация на риски. Каждому витку спирали соответствует очередное уточнение системы, а перед выполнением значительных работ анализируются неопределенности и возможные потери. Графически развитие изображается как движение от центра по расширяющейся спирали. Начальные витки связаны с концепцией, требованиями и проверкой принципиальной реализуемости. Последующие витки приводят к созданию архитектуры, прототипов, компонентов, версий и готовой системы. Расстояние от центра условно может отражать накопленные затраты или степень завершенности проекта. Угловое положение соответствует прохождению определенных видов деятельности внутри итерации. Каждый виток обычно включает четыре взаимосвязанные области: определение целей, альтернатив и ограничений; анализ и снижение рисков; разработку и проверку очередного результата; планирование следующего витка. В первой области определяются цели итерации. Например, целью может быть проверка возможности обработки десяти тысяч запросов в секунду, уточнение требований пользователей или создание базовой архитектуры. Одновременно рассматриваются альтернативы. Для хранения данных можно использовать централизованную реляционную базу, распределенное хранилище или сочетание нескольких технологий. Для каждой альтернативы определяются ограничения по стоимости, срокам, безопасности и квалификации команды. Во второй области выполняется анализ рисков. Риск представляет собой неопределенное событие или условие, способное отрицательно повлиять на проект или систему. К техническим рискам относятся недостаточная производительность, сложность интеграции, отсутствие совместимости, ошибки безопасности и неподтвержденная масштабируемость. К организационным рискам относятся недостаточная квалификация участников, слабое взаимодействие с заказчиком, изменение руководства и отсутствие владельца требований. К экономическим рискам относятся превышение бюджета, изменение стоимости лицензий, недостаточная эффективность и потеря финансирования. К рискам требований относятся неполнота, противоречивость, нестабильность и различное понимание потребностей участниками. После выявления риска выбирается способ его снижения. Это может быть создание прототипа, проведение эксперимента, моделирование, дополнительное обследование, приобретение экспертной консультации или разработка альтернативного решения. Например, если существует риск, что новая база данных не обеспечит требуемую производительность, создается экспериментальный стенд и проводится нагрузочное тестирование. Не требуется сначала разрабатывать всю систему. Если пользователи не могут определить требования к интерфейсу, создается интерактивный прототип. Пользователи оценивают его и уточняют свои ожидания. Если существует риск интеграции с внешней платежной системой, в ранней итерации разрабатывается пробный интеграционный компонент и проверяется взаимодействие. В третьей области создается результат итерации. В зависимости от стадии это может быть концепция, модель, прототип, архитектура, программный компонент или готовая версия системы. Затем результат проверяется. Выполняются техническая оценка, тестирование, демонстрация заинтересованным сторонам и анализ достигнутых целей. В четвертой области планируется следующий виток. Уточняются требования, определяется содержание работ, оцениваются ресурсы, сроки и новые риски. Заказчик или руководство принимает решение о продолжении, изменении направления или прекращении проекта. Таким образом, спиральная модель включает регулярные точки принятия решений.

Проект может быть остановлен после раннего витка, если выясняется, что система технически невозможна, экономически нецелесообразна или не нужна пользователям. Это является преимуществом по сравнению с моделью, в которой значительные ресурсы расходуются до получения информации о критических рисках. Спиральная модель не означает простого многократного повторения программирования. Итерация начинается с целей и анализа риска. Если проект не содержит значительного риска в определенной области, соответствующие работы могут быть сокращены. Главным достоинством модели является раннее выявление критических проблем. Наиболее опасные вопросы рассматриваются до вложения основной части ресурсов. Другим преимуществом является постепенное уточнение требований. Пользователи могут оценивать прототипы и промежуточные версии, поэтому требования формируются на основе практического опыта. Спиральная модель поддерживает эволюционное развитие архитектуры. Архитектурные решения проверяются экспериментально и уточняются на следующих витках. Модель допускает использование разных способов разработки на разных витках. Для хорошо определенной части системы может применяться каскадная последовательность, а для неопределенной части — прототипирование. Например, в проекте медицинской системы правила хранения документов могут быть строго определены нормативными требованиями и проектироваться последовательно. Пользовательский интерфейс врача может уточняться через прототипы, а алгоритм интеллектуальной поддержки решений — через эксперименты. К недостаткам спиральной модели относится сложность управления. Необходимо уметь выявлять, оценивать и контролировать риски. Если риск-анализ выполняется формально, модель теряет свое основное преимущество. Другим недостатком является сложность первоначального планирования окончательных сроков и стоимости. Поскольку содержание последующих витков зависит от результатов предыдущих, точный план всего проекта может быть неизвестен. Спиральная модель требует участия квалифицированных архитекторов, аналитиков и специалистов по управлению рисками. Небольшая команда может не располагать такими ресурсами. Существует опасность бесконечного совершенствования, когда каждый виток порождает новые требования, а критерии завершения системы остаются неопределенными. Поэтому необходимо устанавливать цели продукта, границы проекта и условия приемки. Модель может быть избыточной для небольших, хорошо понятных проектов с низким риском. Разработка простой внутренней формы учета не требует полноценного риск-ориентированного спирального процесса. Особенно оправдано применение спиральной модели при создании крупных, дорогостоящих, инновационных и технически сложных систем, где неудачное архитектурное решение может привести к значительным потерям. Примером может служить разработка распределенной системы управления транспортом. На первом витке уточняется концепция и проверяется возможность получения данных от транспортных средств. На втором создается прототип сбора и отображения данных. На третьем проверяется масштабирование. На четвертом разрабатываются функции управления маршрутами. На последующих витках повышаются безопасность, надежность и полнота функций. В результате каждого витка существует проверяемый результат и новая информация, снижающая неопределенность.

2.7 Сравнение каскадной, поэтапной и спиральной моделей

Каскадная модель ориентирована на последовательное выполнение заранее определенных стадий. Основным механизмом управления является план и утверждение результатов. Поэтапная модель с промежуточным контролем также сохраняет стадийность, но вводит обратные связи, экспертизы и корректировки. Основным механизмом повышения качества является ранняя проверка промежуточных результатов. Спиральная модель строится вокруг повторяющихся циклов и управления рисками. Основным механизмом выбора работ является анализ неопределенности и возможных потерь. В каскадной модели требования предполагаются относительно стабильными. В поэтапной модели допускается их уточнение при обнаружении несоответствий. В спиральной модели постепенное уточнение требований является естественной частью процесса. В каскадной модели работающая система появляется преимущественно в конце. В модели с промежуточным контролем могут создаваться проверочные макеты, но конечный продукт также обычно передается после завершения основных стадий. В спиральной модели прототипы и версии появляются на отдельных витках. Каскадная модель удобнее для жесткого договорного планирования. Спиральная лучше работает при высокой неопределенности, но требует более гибкого управления бюджетом и содержанием проекта. Ни одна модель не является универсально лучшей. Выбор зависит от устойчивости требований, масштаба системы, критичности, технологической новизны, договорных отношений, квалификации команды и необходимости ранней поставки результата.

2.8 Эволюция моделей жизненного цикла информационных систем

Эволюция моделей жизненного цикла отражает развитие программной инженерии и постепенное изменение представлений о создании информационных систем. Основными причинами эволюции стали рост сложности программных комплексов, нестабильность требований, увеличение стоимости ошибок, необходимость быстрого выпуска продуктов и тесного взаимодействия с пользователями. На раннем этапе программирования широко использовался подход, который условно называется «кодирование и исправление». Разработчик начинал писать программу после общего понимания задачи, затем проверял ее, исправлял ошибки и добавлял функции. Для небольших программ такой способ мог быть приемлем. Однако при увеличении размера системы возникали проблемы: отсутствовала общая архитектура, изменения нарушали ранее созданные функции, сроки невозможно было оценить, а сопровождение становилось крайне сложным. Рост крупных программных проектов привел к формированию последовательных инженерных моделей. Программирование начали разделять на анализ требований, проектирование, реализацию, испытания и эксплуатацию. Так сложилась каскадная модель. Каскадная модель перенесла в программную разработку инженерные идеи предварительного проектирования, документального контроля и поэтапной приемки. Она позволила управлять крупными коллективами и договорами, но исходила из предположения о возможности заранее определить основную часть требований. Практика показала, что информационные системы отличаются от многих материальных объектов. Программное обеспечение относительно легко изменять технически, но трудно заранее определить, какие именно изменения потребуются. Пользователи лучше понимают свои потребности после взаимодействия с работающим продуктом. Для устранения чрезмерной жесткости каскада появились модифицированные каскадные модели с обратными связями. Они допустили возвраты между стадиями, промежуточные проверки и уточнение документов. Следующим направлением стала V-образная модель. В ней каждому уровню определения и проектирования соответствует определенный уровень проверки. Пользовательские требования сопоставляются с приемочными испытаниями, системные требования — с системным тестированием, архитектура — с интеграционным тестированием, детальный проект — с тестированием компонентов. V-образная модель усилила связь между разработкой и проверкой. Испытания начинают планироваться одновременно с требованиями и проектированием, а не после завершения программирования. Однако она сохраняет преимущественно последовательную структуру и также предполагает достаточно раннюю стабилизацию требований. Другим направлением развития стало прототипирование. Прототип представляет собой предварительную или упрощенную версию системы, создаваемую для уточнения требований, проверки концепции или оценки технического решения. Исследовательский прототип используется для понимания задачи и может быть отброшен после получения необходимых знаний. Эволюционный прототип постепенно совершенствуется и становится частью конечной системы. Прототипирование особенно полезно при проектировании пользовательских интерфейсов, сложных аналитических функций и новых технологий. Оно сокращает разрыв между письменным описанием и реальным представлением пользователей. Однако быстрый прототип может создать ложное впечатление высокой готовности. Пользователь видит работающий экран, но не видит отсутствия безопасности, надежности, полноценной архитектуры и обработки ошибок. Дальнейшим шагом стала инкрементная модель. Система создается и вводится в эксплуатацию частями. Каждый инкремент добавляет законченный набор функций. Например, сначала внедряется учет клиентов, затем обработка заказов, после этого склад и аналитика. Пользователь получает полезный результат раньше, а проект распределяет риски и затраты по этапам. Инкрементная разработка требует правиль-

ного определения границ приращений. Первый инкремент должен быть не случайным набором функций, а работоспособной частью, создающей ценность. Итерационная модель предполагает многократное уточнение системы. В каждой итерации анализируются требования, проектируются решения, выполняется реализация и проверка. Результат постепенно становится более полным. Итерация и инкремент не тождественны. Итерация связана с повторным уточнением, а инкремент — с добавлением функциональности. На практике они часто сочетаются: каждая итерация создает очередной инкремент. Спиральная модель объединила итерационное развитие с явным управлением рисками. Содержание очередного витка определяется не только перечнем функций, но и необходимостью устранить наиболее опасные неопределенности. В 1980—1990-е годы развивались методы быстрой разработки приложений, основанные на прототипировании, инструментальной автоматизации, повторном использовании компонентов и активном участии пользователей. Их целью было сократить длительные циклы традиционной разработки. Появились унифицированные итерационные процессы, в которых проект разделяется на фазы и итерации. Большое внимание уделяется архитектуре, вариантам использования, управлению требованиями и последовательному снижению рисков. В начале XXI века широкое распространение получили гибкие подходы к разработке. Их развитие было связано с необходимостью быстрее реагировать на изменения, чаще поставлять работающий продукт и усиливать взаимодействие с заказчиком. Гибкая разработка не отменяет требования, проектирование, тестирование и документирование. Она изменяет их организацию. Эти работы выполняются небольшими порциями на протяжении всего проекта, а не только в одной выделенной стадии. Например, требования формируются в виде упорядоченного перечня задач и уточняются перед реализацией. Проектирование выполняется перед каждой значимой функцией и одновременно поддерживается общая архитектура. Тестирование включается в каждую итерацию. Гибкие подходы используют короткие циклы, регулярную демонстрацию результата и обратную связь. Это позволяет быстрее обнаруживать неправильное понимание требований. Однако гибкость не означает отсутствие планирования и документации. Недостаток архитектурного управления может привести к накоплению технического долга. Для ответственных систем требуется сочетание коротких итераций с формальными проверками, документацией и управлением безопасностью. Следующим этапом эволюции стало развитие непрерывной интеграции, непрерывной поставки и культуры совместной работы разработки и эксплуатации, часто обозначаемой термином DevOps. Ранее разработка и эксплуатация могли существовать как отдельные последовательные стадии. Разработчики создавали систему, а затем передавали ее администраторам. В современной практике эксплуатационные требования учитываются с начала проекта, а развертывание, тестирование и контроль инфраструктуры автоматизируются. Изменение программного кода может автоматически запускать сборку, тесты, анализ безопасности и подготовку версии. Небольшие изменения передаются пользователям значительно чаще. Жизненный цикл становится не цепочкой редких крупных выпусков, а непрерывным потоком изменений. При этом процессы эксплуатации и разработки сближаются. Современная модель жизненного цикла все чаще является продуктовой, а не проектной. Команда отвечает не только за создание системы к установленной дате, но и за ее длительную результативность, развитие, надежность и ценность для пользователей. В проектном мышлении основной вопрос заключается в том, завершены ли работы в рамках срока и бюджета. В продуктовом мышлении дополнительно оценивается, достигает ли система полезного результата, используется ли она, насколько удовлетворены пользователи и как изменяются показатели деятельности. Развитие облачных технологий и микросервисной архитектуры усилило непрерывный характер жизненного цикла. Отдельные компоненты могут обновляться независимо, а инфраструктура создается и изменяется с помощью программно описанных конфигураций. Одновременно возрастает роль безопасной разработки. Безопасность должна учитываться в требованиях, архитектуре, программировании, тестировании, развертывании и эксплуатации. Такой

подход часто называется интеграцией безопасности в жизненный цикл. Проверка зависимостей, анализ программного кода, моделирование угроз, управление уязвимостями и обновление компонентов становятся постоянными процессами. Для информационных систем, использующих данные и искусственный интеллект, жизненный цикл дополнительно включает сбор и подготовку данных, обучение моделей, оценку качества, контроль изменений данных, наблюдение за поведением модели и ее переобучение. Такая система может формально не изменять программный код, но изменять результаты из-за появления новых данных. Поэтому управление жизненным циклом должно охватывать не только программные версии, но и наборы данных, модели, параметры и критерии качества. Современная эволюция не означает полного исчезновения каскадных подходов. В реальных организациях применяются гибридные модели. Общая программа может управляться по стадийной схеме, а программные компоненты разрабатываться итерационно. Например, государственный проект может иметь формальные стадии технического задания, проектирования и приемки. Внутри стадии реализации команда может работать короткими итерациями, регулярно демонстрировать результат и автоматизировать тестирование. При создании медицинского оборудования аппаратная часть может разрабатываться по последовательной модели, программный интерфейс — итерационно, а экспериментальный аналитический модуль — по спиральной модели. Таким образом, эволюция моделей жизненного цикла представляет собой не простую замену старых моделей новыми, а накопление способов организации работ. Каждая модель решает определенный класс проблем. Каскадная модель обеспечивает планируемость и документальную управляемость. Модель с промежуточным контролем усиливает раннюю проверку и обратные связи. V-образная модель связывает уровни разработки с уровнями испытаний. Прототипирование помогает уточнять требования. Инкрементная модель ускоряет получение полезного результата. Итерационная модель обеспечивает постепенное уточнение. Спиральная модель управляет рисками. Гибкие подходы повышают способность реагировать на изменения. Непрерывная поставка объединяет разработку и эксплуатацию.

2.9 Выбор и адаптация модели жизненного цикла

Выбор модели должен основываться на характеристиках конкретной информационной системы. Основными факторами являются устойчивость требований, техническая сложность, критичность системы, масштаб команды, длительность проекта, договорная модель, необходимость сертификации, уровень рисков и потребность в раннем получении результата. Если требования стабильны, технология известна, а результаты должны проходить формальную приемку, может быть эффективна каскадная или V-образная модель. Если требования в основном понятны, но возможны ошибки и уточнения, целесообразна поэтапная модель с промежуточным контролем. Если требуется раннее получение отдельных функций, применяется инкрементная модель. Если требования формируются постепенно, используются итерации и прототипирование. Если проект содержит критические технические или экономические риски, оправдана спиральная модель. Если продукт развивается в изменчивой среде и необходимо регулярно поставлять обновления, применяются гибкие и непрерывные процессы. Одна модель может использоваться на разных уровнях. Вся система разрабатывается по спиральной модели, отдельный хорошо определенный компонент — по каскадной, а пользовательский интерфейс — через прототипирование. Адаптация модели включает определение стадий, процессов, ролей, результатов, документов, контрольных точек, итераций, критериев завершения и порядка управления изменениями. При адаптации необходимо установить, какие документы создаются, кто их утверждает, какие испытания выполняются, как регистрируются риски, как управляются версии и каким образом система передается в эксплуатацию. Слишком тяжелая модель увеличивает бюрократическую нагрузку. Слишком упрощенная модель создает риск хаотичной разработки. Цель адаптации заключается в достижении необходимой управляемости без неоправданных затрат.

2.10 Ошибки при организации жизненного цикла

Распространенной ошибкой является отождествление жизненного цикла с программированием. В результате недостаточно внимания уделяется требованиям, внедрению, данным, обучению и сопровождению. Другой ошибкой является выбор модели по признаку популярности. Микросервисная архитектура не требует автоматически гибкой разработки, а гибкий подход не является универсальным решением для любого договора и любой критической системы. Ошибкой является формальное применение стандарта. Наличие комплекта документов не гарантирует качества, если документы не используются для принятия решений и не соответствуют реальному состоянию системы. Опасно исключать пользователей до стадии приемки. Даже подробно написанное техническое задание не всегда позволяет заранее определить все особенности реальной работы. Другой крайностью является бесконтрольное изменение требований. Готовность реагировать на изменения должна сочетаться с оценкой их влияния, приоритетов, стоимости и рисков. Существенной ошибкой является перенос тестирования в конец жизненного цикла. Чем позже обнаружен дефект требований или архитектуры, тем больше компонентов требуется переработать. Недостаточное внимание к эксплуатации приводит к отсутствию мониторинга, резервного копирования, диагностики, инструкций и подготовленного персонала. Игнорирование снятия с эксплуатации создает риски потери данных, нарушения требований хранения информации, сохранения несанкционированных доступов и зависимости от устаревших технологий. Грамотно организованный жизненный цикл связывает замысел, требования, проектирование, реализацию, проверку, эксплуатацию, сопровождение и прекращение использования системы в единый управляемый процесс. Качество информационной системы определяется не только качеством программного кода, но и качеством решений, принимаемых на всех стадиях ее существования.

3. Стандарты проектирования архитектуры информационных систем, требования к программному обеспечению

Стандартизация проектирования информационных систем представляет собой установление и применение единых принципов, терминов, процессов, стадий, требований и правил документирования, используемых при создании, развитии, эксплуатации и сопровождении информационных систем. Необходимость стандартизации обусловлена тем, что современная информационная система создается не одним специалистом и обычно не ограничивается одним программным модулем. В ее проектировании участвуют заказчики, пользователи, системные аналитики, архитекторы, разработчики, специалисты по базам данных, тестировщики, администраторы, специалисты по информационной безопасности, технические писатели, руководители проектов и представители организаций, принимающих систему в эксплуатацию.

Каждая из перечисленных групп рассматривает информационную систему со своей позиции. Пользователю важно, чтобы система помогала выполнять повседневные задачи. Заказчика интересуют достижение организационных целей, стоимость и сроки. Архитектор определяет структуру системы, границы компонентов и способы их взаимодействия. Разработчик реализует конкретные программные функции. Тестировщик должен понимать, каким образом проверить соответствие системы установленным требованиям. Администратор оценивает возможность установки, настройки, наблюдения и восстановления системы. Если участники используют различные термины и по-разному понимают содержание стадий и документов, проект становится трудноуправляемым. Стандарты создают единое понятийное и процессное пространство. Они позволяют определить, какие работы должны быть выполнены, какие результаты должны быть получены, каким образом формируются требования, что включается в архитектурное описание, как проводится проверка системы и какие документы передаются заказчику. При этом стандарт не всегда устанавливает конкретную технологию. Он может определять требования к результату, но оставлять разработчику право выбирать язык программирования, систему управления базами данных, архитектурный стиль и инструментальные средства. Например, стандарт может требовать определить архитектуру информационной системы, документировать ее значимые элементы и учитывать интересы заинтересованных сторон. Однако он не обязан предписывать использование монолитной или микросервисной архитектуры. Такое решение принимается в зависимости от масштаба, нагрузки, требований к надежности, компетенций команды и ограничений проекта. Стандарты проектирования архитектуры информационных систем необходимо рассматривать совместно со стандартами жизненного цикла и инженерии требований. Архитектура не разрабатывается независимо от потребностей пользователей. Она является результатом преобразования целей, пользовательских и системных требований в структурные и технические решения.

В современной нормативной системе можно условно выделить несколько взаимосвязанных уровней. Первый уровень определяет стадии создания автоматизированной системы. Второй уровень описывает процессы жизненного цикла систем и программного обеспечения. Третий уровень регулирует инженерию требований. Четвертый уровень устанавливает принципы описания архитектуры. Пятый уровень определяет состав и содержание проектной, программной и эксплуатационной документации. Дополнительные стандарты регулируют качество программного продукта, тестирование, информационную безопасность, управление конфигурацией, сопровождение и другие специальные области.

В российской нормативной базе стадии создания автоматизированных систем устанавливает ГОСТ Р 59793—2021 «Информационные технологии. Комплекс стандартов на автоматизированные системы. Автоматизированные системы. Стадии создания». Требования к техническому заданию на создание автоматизированной системы определяет ГОСТ 34.602—2020. Требования к содержанию основных документов, разрабатываемых при создании автоматизированных систем, устанавливает ГОСТ Р 59795—2021. Процессы жизненного цикла систем рассматриваются в действующем ГОСТ Р 57193—2025, а способы представления и документирования архитектуры — в ГОСТ Р 57100—2025. Последний определяет основные понятия архитектурного описания, архитектурные точки зрения, представления, модели и языки описания архитектуры.

Международный уровень представлен стандартами ISO/IEC/IEEE 12207, посвященными процессам жизненного цикла программного обеспечения, ISO/IEC/IEEE 15288, посвященными процессам жизненного цикла систем, ISO/IEC/IEEE 29148, регулирующими инженерию требований, ISO/IEC/IEEE 42010, посвященными описанию архитектуры, а также серией ISO/IEC 25000, устанавливающей модели и методы определения и оценки качества систем и программных продуктов. Стандарты не должны восприниматься как независимые и конкурирующие документы. Они рассматривают один объект с различных сторон. Стандарт стадий создания отвечает на вопрос, в какой общей последовательности организуется проект. Стандарты процессов жизненного цикла определяют, какие процессы необходимо выполнять. Стандарт инженерии требований устанавливает правила получения, анализа и документирования требований. Стандарт архитектурного описания регулирует представление архитектурных решений. Стандарты качества помогают определить характеристики, которыми должна обладать система.

3.1 Отечественный стандарт жизненного цикла автоматизированных систем

В отечественной практике длительное время основным документом, определявшим стадии создания автоматизированных систем, являлся ГОСТ 34.601—90 «Автоматизированные системы. Стадии создания». Его положения оказали значительное влияние на организацию разработки государственных, производственных, банковских, учетных и управленческих систем. На основе этого стандарта формировались технические задания, проектная документация, программы испытаний и договорные отношения между заказчиками и исполнителями.

В настоящее время национальным стандартом, непосредственно устанавливающим стадии и этапы создания автоматизированных систем, является ГОСТ Р 59793—2021. Он введен в действие 30 апреля 2022 года и распространяется на автоматизированные системы, используемые в управлении, проектировании, исследованиях и других видах деятельности. Стандарт определяет восемь основных стадий: формирование требований к автоматизированной системе, разработку концепции, техническое задание, эскизный проект, технический проект, рабочую документацию, ввод в действие и сопровождение автоматизированной системы.

Под автоматизированной системой понимается организационно-техническая система, в которой информационные технологии и средства автоматизации применяются совместно с деятельностью персонала. Поэтому стандарт охватывает не только разработку программного кода. Он предусматривает обследование объекта автоматизации, изменение организационных процессов, подготовку персонала, комплектацию техническими средствами, загрузку данных, испытания и последующее обслуживание. Процесс создания автоматизированной системы в стандарте рассматривается как совокупность упорядоченных во времени и взаимосвязанных работ, объединенных в стадии и этапы. Каждая стадия должна завершаться определенным результатом. Стадийное деление необходимо для рационального планирования, распределения ответственности и контроля готовности системы.

3.1.1 Формирование требований к автоматизированной системе

Первая стадия называется «Формирование требований к автоматизированной системе». Она включает обследование объекта автоматизации, обоснование необходимости создания системы, формирование требований пользователя и оформление отчета о выполненной работе. Объект автоматизации представляет собой организацию, подразделение, производственный процесс, технологический объект или иной комплекс деятельности, для которого создается система. До начала проектирования необходимо понять, как этот объект функционирует в существующем состоянии. В ходе обследования собираются сведения о структуре организации, функциях подразделений, составе пользователей, документах, информационных потоках, применяемых программах, технических средствах и проблемах существующей деятельности. Необходимо установить, какие операции выполняются вручную, где происходит повторный ввод данных, какие задержки и ошибки возникают, какие сведения отсутствуют у руководства и какие процессы требуют автоматизации. Например, при обследовании склада анализируются поступление товара, приемка, размещение, хранение, перемещение, инвентаризация, комплектование заказов и отгрузка. Исследуются бумажные документы, электронные таблицы, используемые программы и обязанности сотрудников. Если одна и та же информация многократно вводится кладовщиком, бухгалтером и менеджером, это указывает на проблему дублирования данных.

Обследование не должно ограничиваться фиксацией пожеланий руководителя. Необходимо изучать фактическую деятельность пользователей. Формальный регламент и реальный порядок работы могут различаться. Сотрудники могут использовать неофициальные элек-

тронные таблицы, дополнительные журналы или устные согласования, которые не отражены в документации, но являются важной частью процесса. После сбора информации оценивается качество функционирования объекта и выявляются проблемы, которые могут быть решены средствами автоматизации. При этом автоматизация не является самоцелью. Некоторые проблемы вызваны не отсутствием программ, а неправильным распределением полномочий, противоречивыми регламентами или отсутствием ответственности за данные. В таких случаях одна только разработка информационной системы не устранит причину проблемы. На этой же стадии выполняется оценка целесообразности создания системы. Анализируются предполагаемые затраты, сроки, организационные последствия, экономический или социальный эффект. Например, создание системы может уменьшить количество ошибок, сократить время обработки заявки, повысить прозрачность работы или обеспечить выполнение требований законодательства. Затем формируются требования пользователя. В них описываются цели автоматизации, необходимые функции, ограничения стоимости и сроков, ожидаемые эффекты, условия работы и общие характеристики системы. Стандарт прямо предусматривает подготовку исходных данных и оформление требований пользователя к автоматизированной системе. Результатом первой стадии является отчет, содержащий результаты обследования, обоснование необходимости создания системы и сформулированные требования пользователя. На основании отчета может быть подготовлена заявка на разработку технического задания.

3.1.2 Разработка концепции автоматизированной системы

Вторая стадия называется «Разработка концепции автоматизированной системы». Ее назначение заключается в поиске и выборе принципиального варианта будущей системы. На этой стадии выполняется более глубокое изучение объекта автоматизации. При необходимости проводятся научно-исследовательские работы. Это особенно важно, если система использует новые алгоритмы, сложные методы обработки данных, специализированное оборудование или технологии, применимость которых еще не подтверждена. Разработчик формирует несколько возможных вариантов концепции. Например, организация может выбирать между развитием существующей системы, приобретением готового программного продукта, разработкой новой системы или использованием облачного решения. Каждый вариант должен оцениваться по функциональным возможностям, стоимости, срокам, рискам, требованиям к инфраструктуре, сложности сопровождения и соответствию потребностям пользователей. Недостаточно выбрать самый дешевый или технологически современный вариант. Необходимо определить, какой вариант обеспечивает требуемый результат с приемлемыми рисками и затратами. Концепция может включать назначение системы, общую структуру, предполагаемые подсистемы, основные информационные потоки, способы взаимодействия с внешними системами, принцип организации данных, требования к технической инфраструктуре и предполагаемый порядок внедрения. Например, концепция университетской информационной системы может предусматривать единое централизованное хранилище данных, личные кабинеты студентов и преподавателей, интеграцию с бухгалтерской и библиотечной системами, электронное расписание и поэтапный ввод функциональных подсистем. В действующем отечественном стандарте отдельным этапом предусмотрена оценка рисков проекта. Необходимо выявить риски, оценить их вероятность и последствия, определить приоритеты и подготовить мероприятия по предупреждению рисков или реагированию на них. К рискам могут относиться недостаток финансирования, невозможность интеграции с существующей системой, низкое качество исходных данных, сопротивление пользователей, отсутствие специалистов, несоответствие выбранной технологии требованиям производительности или изменение законодательства. Результатом стадии является отчет, содержащий описание рассмотренных вариантов и обоснование выбранной концепции.

3.1.3 Техническое задание

Третья стадия называется «Техническое задание». На ней разрабатывается, согласовывается и утверждается техническое задание на создание автоматизированной системы. Техническое задание является основным документом, устанавливающим назначение системы, цели ее создания, требования, состав работ, порядок контроля и условия приемки. Оно связывает потребности заказчика с деятельностью разработчика. Техническое задание должно быть достаточно полным, чтобы определить границы проекта и ожидаемый результат, но не должно неоправданно ограничивать разработчика деталями, которые могут быть определены на стадии проектирования. Например, в техническом задании необходимо установить требуемую производительность, но не всегда требуется заранее назначать конкретную марку сервера. Действующий ГОСТ 34.602—2020 предусматривает обязательные разделы технического задания: общие сведения; цели и назначение создания системы; характеристику объектов автоматизации; требования к автоматизированной системе; состав и содержание работ; порядок разработки; порядок контроля и приемки; требования к подготовке объекта к вводу системы в действие; требования к документированию и источники разработки. В разделе требований могут фиксироваться требования к структуре и функционированию системы, численности и квалификации персонала, показателям назначения, надежности, безопасности, защите информации, эргономике, эксплуатации, техническому, информационному, программному, организационному и другим видам обеспечения. Техническое задание имеет не только техническое, но и договорное значение. Оно позволяет установить, что должен создать исполнитель и по каким критериям заказчик будет принимать результат. Неопределенные или непроверяемые формулировки в техническом задании становятся причиной конфликтов. Например, требование «система должна работать быстро» невозможно объективно проверить. Корректнее указать: «при штатной нагрузке время формирования карточки клиента не должно превышать двух секунд для 95 процентов запросов». Такое требование содержит условие, измеряемый показатель и допустимое значение.

3.1.4 Эскизный проект

Четвертая стадия называется «Эскизный проект». На ней разрабатываются предварительные проектные решения по системе и ее частям. Эскизный проект определяет основные функции системы и подсистем, укрупненную структуру информационной базы, состав вычислительных средств и основные параметры программного обеспечения. Он не содержит всей детализации, необходимой для непосредственной реализации, но позволяет сформировать общее представление о будущем решении. Например, на этой стадии может быть принято решение разделить систему на подсистемы управления пользователями, заявками, документами и отчетностью. Определяется, какие данные будут храниться централизованно, какие внешние системы необходимо подключить и какие пользовательские роли предусмотрены. Эскизный проект особенно полезен в крупных системах, когда требуется согласовать принципиальную структуру до разработки детальной документации. Он позволяет обнаружить неправильное понимание требований на сравнительно раннем этапе. Вместе с тем действующий стандарт допускает исключение стадии эскизного проекта. В этом случае соответствующие работы могут быть включены в технический проект.

3.1.5 Технический проект

Пятая стадия называется «Технический проект». На ней разрабатываются окончательные проектные решения по автоматизированной системе и ее частям. Технический проект раскрывает архитектуру системы, функции подсистем, организационную структуру, функции персонала, техническое, математическое, программное, информационное и лингвистическое обеспечение. На этой стадии требования преобразуются в конкретную архитектуру. Определяются состав программных компонентов, интерфейсы, модели данных, способы интеграции, механизмы разграничения доступа, структура технической инфраструктуры, способы резервного копирования и восстановления. Например, если техническое задание требует обмена дан-

ными с платежной системой, технический проект должен определить состав передаваемых данных, способ аутентификации, последовательность взаимодействия, обработку ошибок и защиту канала связи. Технический проект должен учитывать все виды обеспечения автоматизированной системы. Информационное обеспечение включает состав и структуру данных, классификаторы, справочники и информационные потоки. Программное обеспечение охватывает программы, модули, библиотеки и общесистемные средства. Техническое обеспечение включает серверы, рабочие станции, сети и периферийные устройства. Организационное обеспечение определяет роли, регламенты и взаимодействие персонала. На стадии технического проекта также разрабатывается документация на поставку изделий и технические задания на создание нестандартных средств, если они необходимы для системы.

3.1.6 Рабочая документация

Шестая стадия называется «Рабочая документация». Она включает разработку документации, необходимой для создания, ввода в действие, эксплуатации и сопровождения системы, а также разработку или адаптацию отдельных видов обеспечения. Рабочая документация должна содержать сведения, достаточные для реализации проектных решений. В нее могут входить описания программ, инструкции по установке, руководства пользователей, инструкции администраторов, схемы баз данных, настройки, эксплуатационные документы, программы и методики испытаний. На этой стадии создается или адаптируется программное обеспечение, настраиваются базы данных, приобретаются и конфигурируются технические средства, разрабатываются справочники и выполняются другие работы, обеспечивающие готовность системы. Действующий стандарт допускает объединение стадий «Технический проект» и «Рабочая документация» в стадию «Технорабочий проект». Это позволяет адаптировать нормативную модель к конкретному проекту.

3.1.7 Ввод в действие

Седьмая стадия называется «Ввод в действие». Она является одной из наиболее насыщенных и включает подготовку объекта автоматизации, подготовку персонала, комплектацию системы, строительно-монтажные и пусконаладочные работы, предварительные испытания, опытную эксплуатацию и приемочные испытания. Подготовка объекта автоматизации означает не только подготовку помещений и оборудования. Необходимо изменить организационные регламенты, определить ответственность сотрудников, подготовить исходные данные, справочники и документы. Подготовка персонала включает обучение пользователей, администраторов и других сотрудников. Проверяется их способность выполнять операции и поддерживать функционирование системы. Пусконаладочные работы охватывают установку и настройку технических и программных средств, загрузку данных, проверку информационной базы и комплексную наладку системы. На этапе предварительных испытаний проверяется работоспособность системы и ее соответствие техническому заданию. Выявленные недостатки устраняются, а документация уточняется. По результатам оформляется акт о приемке в опытную эксплуатацию. Опытная эксплуатация представляет собой использование системы в условиях, близких к реальной работе, но под усиленным наблюдением разработчика и заказчика. Она позволяет обнаружить недостатки, которые не проявились при лабораторном тестировании. Например, система может корректно работать на тестовых данных, но при реальном использовании выясняется, что пользователи вводят информацию в неожиданной последовательности, некоторые операции занимают слишком много времени, а отдельные справочники содержат ошибки. По результатам опытной эксплуатации выполняются анализ, доработка программного и информационного обеспечения, корректировка документации и дополнительная настройка технических средств. На этапе приемочных испытаний проверяется соответствие системы техническому заданию и готовность к постоянной эксплуатации. Результатом является акт о приемке автоматизированной системы в постоянную эксплуатацию.

3.1.8 Сопровождение автоматизированной системы

Восьмая стадия называется «Сопровождение автоматизированной системы». Она включает выполнение гарантийных обязательств и послегарантийное обслуживание. В гарантийный период устраняются недостатки, выявленные в ходе эксплуатации, и вносятся изменения в документацию. Послегарантийное сопровождение включает анализ функционирования системы, выявление отклонений фактических характеристик от проектных значений, установление их причин, устранение недостатков и поддержание стабильности эксплуатации. Сопровождение нельзя рассматривать только как исправление программных ошибок. Система может требовать адаптации к изменениям законодательства, технической инфраструктуры, организационной структуры и внешних информационных систем. Стандарт допускает исключение отдельных этапов, параллельное выполнение работ и включение новых этапов. Следовательно, отечественная стадийная модель не требует абсолютно жесткого каскадного движения. Конкретный состав работ определяется договорами, техническим заданием и организационно-распорядительными документами.

3.2 Первичная стандартизация процессов жизненного цикла программных средств

Под первичной стандартизацией процессов жизненного цикла программных средств обычно понимается формирование первоначальной структуры международного стандарта ISO/IEC 12207:1995 и ее отечественное воспроизведение в ГОСТ Р ИСО/МЭК 12207—99. До появления этого стандарта различные организации использовали собственные модели, термины и классификации процессов разработки программного обеспечения. Одни документы описывали стадии проекта, другие — программные работы, третьи — управление качеством. Отсутствовала единая международная структура, охватывающая приобретение, разработку, эксплуатацию и сопровождение программных средств. ISO/IEC 12207:1995 стал первым международным стандартом, предложившим целостный набор процессов жизненного цикла программного обеспечения. В нем процессы были разделены на пять основных, восемь вспомогательных и четыре организационных процесса. Важно различать стадии и процессы. Стадия представляет собой временной период жизненного цикла. Процесс является совокупностью взаимосвязанных действий, преобразующих входные данные в результаты. Один процесс может выполняться на нескольких стадиях. Например, документирование осуществляется при формировании требований, проектировании, программировании, испытаниях и сопровождении. Следовательно, документирование не является одной ограниченной стадией, а представляет собой процесс, сопровождающий значительную часть жизненного цикла.

3.2.1 Основные процессы первоначального стандарта

К основным относились процесс заказа, процесс поставки, процесс разработки, процесс эксплуатации и процесс сопровождения. Процесс заказа, также называемый процессом приобретения, описывал деятельность организации, которая приобретает программный продукт или программную услугу. Заказчиком может быть внешняя организация или подразделение одной организации. Процесс начинался с определения потребности. Заказчик должен был установить цели приобретения, требования к продукту, ограничения, критерии приемки и условия договора. Затем подготавливалась заявка или конкурсная документация, выбирался поставщик, контролировалось выполнение договора и проводилась приемка результата. Например, университет, приобретающий систему электронного обучения, должен определить количество пользователей, необходимые функции, требования к защите данных, интеграции и технической поддержке. Затем он сравнивает предложения поставщиков и принимает систему по установленным критериям. Процесс поставки описывал деятельность организации, предоставляющей заказчику программный продукт или услугу. Поставщик анализировал запрос, готовил предложение, заключал договор, планировал работы, создавал или адаптировал продукт и передавал его заказчику. Заказ и поставка являются взаимосвязанными процессами. Один и тот же договор рассматривается с позиции двух сторон. Заказчик формирует требования и принимает результат, а поставщик организует выполнение обязательств. Процесс разработки охватывал анализ требований, проектирование, программирование, интеграцию, тестирование, установку и поддержку приемки программного средства. На этапе анализа определялись требования к программному обеспечению. Затем разрабатывалась архитектура, выделялись программные компоненты, интерфейсы и структуры данных. После детального проектирования выполнялись программирование и тестирование отдельных компонентов. Далее компоненты объединялись, проводилось интеграционное и квалификационное тестирование, после чего программный продукт устанавливался в рабочей среде и передавался заказчику. Процесс эксплуатации описывал деятельность организации, использующей программное средство. В него входили подготовка эксплуатационной среды, выполнение операций, помощь пользователям, обработка обращений и контроль работы системы. Эксплуатация не является пассивным

использованием готовой программы. Она требует управления пользователями, настройки, резервного копирования, контроля доступности, регистрации инцидентов и выполнения регламентных работ. Процесс сопровождения охватывал изменение программного продукта после его передачи в эксплуатацию. Сопровождающая организация анализировала сообщения о проблемах и запросы на изменение, модифицировала программные компоненты, проводила тестирование и выпускала обновления. Сопровождение могло быть корректирующим, адаптивным, совершенствующим и профилактическим. Исправление ошибки относится к корректирующему сопровождению. Адаптация программы к новой операционной системе является адаптивным сопровождением. Добавление новой функции относится к совершенствующему сопровождению.

3.2.2 Вспомогательные процессы первоначального стандарта

Вспомогательные процессы не создавали основной продукт самостоятельно, но обеспечивали качество, управляемость и проверяемость основных процессов. К ним относились документирование, управление конфигурацией, обеспечение качества, верификация, валидация, совместный анализ, аудит и решение проблем. Процесс документирования обеспечивал планирование, разработку, оформление, распространение и сопровождение документов. Документами могли быть требования, проектные описания, руководства, планы испытаний и отчеты. Управление конфигурацией обеспечивало идентификацию программных компонентов и документов, контроль изменений и учет версий. Без управления конфигурацией невозможно точно установить, какие исходные файлы, библиотеки, настройки и документы входят в конкретную версию продукта. Например, если в промышленной эксплуатации возникла ошибка, необходимо определить, какая версия программы установлена, какой набор изменений она содержит и какие инструкции ей соответствуют. Обеспечение качества предоставляло уверенность в том, что процессы и результаты соответствуют установленным требованиям, планам и договорным условиям. Обеспечение качества не тождественно тестированию. Оно охватывает организацию работ, соблюдение процессов, квалификацию участников, документацию и контроль результатов. Верификация отвечала на вопрос, правильно ли создан результат относительно исходных спецификаций. Она могла применяться к требованиям, архитектуре, программному коду и документации. Например, верификация проектной модели определяет, соответствует ли она утвержденным системным требованиям. Валидация отвечала на вопрос, пригоден ли конечный продукт для предполагаемого использования и удовлетворяет ли реальные потребности пользователя. Система может быть правильно создана по техническому заданию, но оказаться неудобной для фактической работы. В таком случае она прошла формальную верификацию, но не полностью прошла валидацию. Совместный анализ предусматривал регулярную совместную оценку состояния проекта заказчиком и поставщиком. Рассматривались технические решения, ход выполнения работ, проблемы, риски и готовность результатов. Аудит представлял собой независимую проверку соответствия процессов или продуктов установленным требованиям. Аудит мог проверять документацию, выполнение процедур, результаты тестирования и управление конфигурацией. Процесс решения проблем обеспечивал регистрацию, анализ, устранение и контроль проблем, обнаруженных на протяжении жизненного цикла. Проблемой могла быть программная ошибка, несоответствие документации, нарушение процесса или отказ технического средства.

3.2.3 Организационные процессы первоначального стандарта

К организационным относились управление, создание инфраструктуры, усовершенствование процессов и обучение. Процесс управления включал планирование, организацию, контроль и оценку работ. Он применялся не только к разработке, но и к заказу, поставке, эксплуатации и сопровождению. Процесс создания инфраструктуры обеспечивал проект необходимыми средствами. К инфраструктуре относились оборудование, программные инструменты, сети, стандарты организации, помещения и средства связи. Процесс усовершенствова-

ния был направлен на анализ и развитие процессов организации. После завершения проектов организация должна была выявлять проблемы, сохранять успешные практики и изменять внутренние регламенты. Процесс обучения обеспечивал подготовку персонала. Новая технология или методология не может быть внедрена только распоряжением руководства. Сотрудники должны получить знания и практические навыки. Первоначальная модель ISO/IEC 12207 имела большое значение, поскольку отделила процессы от стадий и показала, что жизненный цикл включает не только программирование. Однако структура 1995 года преимущественно рассматривала программное обеспечение как самостоятельный объект и недостаточно полно объединяла программную и системную инженерию. Это стало одной из причин дальнейшей эволюции стандарта.

3.3 Глобальная унифицированная стандартизация процессов жизненного цикла информационных систем

Развитие информационных систем показало, что программное обеспечение невозможно полноценно проектировать вне контекста системы. Программа взаимодействует с оборудованием, пользователями, данными, организационными процессами и внешними системами. Поэтому потребовалось сближение стандартов программной и системной инженерии. Важным этапом стала разработка стандарта ISO/IEC 15288, определившего процессы жизненного цикла систем. В дальнейшем структуры ISO/IEC 12207 и ISO/IEC 15288 были гармонизированы. В России эта линия развития была отражена в ГОСТ Р ИСО/МЭК 12207—2010, созданном на основе ISO/IEC 12207:2008. Он заменил ГОСТ Р ИСО/МЭК 12207—99 и установил общую структуру процессов жизненного цикла программных средств, применимую к приобретению, поставке, разработке, эксплуатации, сопровождению и прекращению применения программных продуктов. Глобальная унификация заключалась в переходе от относительно изолированного описания разработки программ к согласованной системе процессов, применимой к системе в целом и к ее программным элементам. В структуре редакции 2008 года, отраженной в отечественном стандарте 2010 года, выделялись процессы соглашения, процессы организационного обеспечения проекта, процессы проекта, технические процессы, процессы реализации программных средств, процессы поддержки программных средств и процессы повторного применения программных средств.

3.3.1 Процессы соглашения

Процессы соглашения регулировали отношения между приобретателем и поставщиком. К ним относились приобретение и поставка. Процесс приобретения включал подготовку стратегии приобретения, формирование требований, выбор поставщика, заключение соглашения, контроль выполнения и приемку. Процесс поставки включал анализ запроса, подготовку предложения, планирование, выполнение соглашения, передачу результатов и поддержку приемки. Выделение процессов соглашения подчеркивало, что жизненный цикл информационной системы может выполняться несколькими организациями. Заказчик, разработчик, поставщик оборудования и сопровождающая организация могут быть различными юридическими лицами.

3.3.2 Процессы организационного обеспечения проекта

Эта группа создавала условия для выполнения проектов на уровне всей организации. Она включала управление моделью жизненного цикла, управление инфраструктурой, управление портфелем проектов, управление человеческими ресурсами, управление качеством и управление знаниями. Управление моделью жизненного цикла обеспечивало выбор, адаптацию и совершенствование процессов организации. Управление инфраструктурой обеспечивало наличие программных, технических и организационных средств. Управление портфелем проектов позволяло отбирать и координировать проекты с учетом стратегии и ресурсов организации. Управление человеческими ресурсами обеспечивало подбор, развитие и распределение специалистов. Управление качеством формировало политику и систему контроля качества. Управление знаниями обеспечивало сохранение и повторное использование опыта, решений, моделей и результатов проектов.

3.3.3 Процессы проекта

Процессы проекта обеспечивали непосредственное управление конкретной разработкой. К ним относились планирование, оценка и контроль проекта, управление решениями, рисками, конфигурацией, информацией и измерениями. Планирование проекта определяло содержание, сроки, ресурсы, роли, результаты и контрольные точки. Оценка и контроль проекта обеспечивали сравнение фактического состояния с планом. При выявлении отклонений

принимались корректирующие меры. Управление решениями обеспечивало формализованный выбор между альтернативами. Архитектурное решение должно приниматься не на основе личного предпочтения, а на основе критериев и анализа последствий. Управление рисками включало выявление, анализ, обработку и наблюдение за рисками. Управление конфигурацией контролировало состав и версии системы. Управление информацией регулировало создание, хранение, распространение и защиту проектной информации. Измерение обеспечивало получение количественных данных о процессе и продукте. Измеряться могут длительность операций, количество дефектов, производительность системы, объем изменений и другие показатели.

3.3.4 Технические процессы

Технические процессы связывали потребности заинтересованных сторон с созданием и эксплуатацией системы. Они включали определение требований заинтересованных сторон, определение системных требований, определение архитектуры, реализацию элементов, интеграцию, проверку, переход к эксплуатации, валидацию, эксплуатацию, сопровождение и прекращение применения. Процесс определения требований заинтересованных сторон выявлял потребности пользователей, заказчиков, операторов, владельцев и других участников. Процесс определения системных требований преобразовывал эти потребности в технически точное описание системы. Процесс определения архитектуры формировал структуру системы и распределял требования между ее элементами. Процесс реализации создавал системные элементы. Процесс интеграции объединял элементы в систему. Верификация подтверждала соответствие созданных результатов установленным требованиям и проектным решениям. Валидация подтверждала пригодность системы для предполагаемого использования. Переход обеспечивал передачу системы в эксплуатационную среду. Эксплуатация, сопровождение и прекращение применения охватывали последующие периоды жизненного цикла.

3.3.5 Процессы реализации программных средств

Поскольку программное обеспечение имеет собственные особенности, стандарт дополнительно выделял специализированные процессы его реализации. Они включали реализацию программного средства, анализ требований к программному обеспечению, проектирование программной архитектуры, детальное проектирование, конструирование, интеграцию и квалификационное тестирование. Анализ требований к программному обеспечению выполнялся после распределения системных требований между системными элементами. Не все системные требования являются программными. Например, часть требований может реализовываться оборудованием, персоналом или организационной процедурой. Проектирование программной архитектуры определяло крупные компоненты программного обеспечения, их обязанности, интерфейсы и зависимости. Детальное проектирование описывало внутреннее устройство компонентов, алгоритмы и структуры данных. Конструирование включало программирование, модульное тестирование и подготовку программных компонентов. Интеграция объединяла компоненты, а квалификационное тестирование подтверждало соответствие программного продукта требованиям.

3.3.6 Процессы поддержки программных средств

К процессам поддержки относились управление документацией, конфигурацией, качеством, верификация, валидация, совместные анализы, аудит и решение проблем. По сравнению с первоначальной редакцией их содержание было согласовано с системными и проектными процессами. Они могли применяться к различным программным работам и продуктам.

3.3.7 Процессы повторного применения программных средств

Отдельная группа была посвящена повторному использованию. Она включала инженерный анализ предметной области, управление повторно используемыми активами и управление программой повторного применения. Повторно используемым активом может быть программный компонент, библиотека, архитектурный образец, модель данных, тест, шаблон документа

или другой результат, пригодный для применения в нескольких проектах. Повторное использование требует управления. Нельзя просто копировать программный код из старого проекта. Необходимо оценивать его качество, совместимость, права использования, актуальность и пригодность.

3.3.8 Современное развитие унифицированной стандартизации

Следующая крупная международная редакция ISO/IEC/IEEE 12207:2017 еще сильнее гармонизировала процессы программного обеспечения с ISO/IEC/IEEE 15288. В ней использовались четыре основные группы: процессы соглашения, процессы организационного обеспечения проекта, процессы технического управления и технические процессы. В апреле 2026 года опубликован ISO/IEC/IEEE 12207:2026. Он устанавливает общую систему процессов полного жизненного цикла программных систем, продуктов и услуг, включая формирование замысла, разработку, эксплуатацию, поддержку и прекращение применения. Стандарт допускает одновременное, итерационное, рекурсивное и инкрементное выполнение процессов и не предписывает одну обязательную модель жизненного цикла. На системном уровне в России действует ГОСТ Р 57193—2025, разработанный с учетом ISO/IEC/IEEE 15288:2023 и заменивший ГОСТ Р 57193—2016. Он распространяется на процессы жизненного цикла систем и применяется к системам, создаваемым человеком. Смысл глобальной унификации заключается не в установлении одинаковой последовательности стадий для всех проектов. Унификация создает общий набор процессов, целей, результатов и терминов. Конкретная организация адаптирует их к каскадной, итерационной, спиральной, инкрементной или иной модели.

3.4 Понятие требования к информационной системе и программному обеспечению

Требование представляет собой документированное положение о потребности, возможности, свойстве, функции, характеристике или ограничении, которым должна соответствовать система, программный продукт, процесс или услуга. Требования являются связующим звеном между проблемой заказчика и техническим решением. Они объясняют, зачем создается система, что она должна делать, в каких условиях работать и каким критериям соответствовать. Требование не следует смешивать с пожеланием, идеей или проектным решением. Пожелание может быть неопределенным: «необходимо улучшить обслуживание клиентов». Требование должно уточнять, какой результат ожидается. Проектное решение указывает, каким способом требование будет реализовано. Например, положение «пользователь должен получать уведомление о подтверждении заказа» является функциональным требованием. Решение «уведомление отправляется через брокер сообщений» относится к архитектуре реализации. При формировании требований необходимо различать проблемную область и область решения. Проблемная область описывает деятельность, потребности и ограничения организации. Область решения охватывает программные и технические способы реализации. Преждевременное смешение этих областей приводит к тому, что пользователи формулируют не потребность, а привычный способ ее удовлетворения. Например, пользователь может потребовать «добавить кнопку экспорта в электронную таблицу». Его реальная потребность может заключаться в регулярном получении аналитического отчета. Возможно, автоматическая отправка отчета будет более подходящим решением. Инженерия требований представляет собой систематическую деятельность по выявлению, анализу, согласованию, документированию, проверке, валидации и управлению требованиями. Международный стандарт ISO/IEC/IEEE 29148:2018 регулирует процессы инженерии требований для систем и программных продуктов. Он определяет процессы, информационные результаты, их содержание и рекомендации по оформлению документации требований. Стандарт применяется совместно с процессами ISO/IEC/IEEE 15288 и ISO/IEC/IEEE 12207. Инженерия требований включает несколько взаимосвязанных видов деятельности. Сначала выявляются заинтересованные стороны и их потребности. Затем требования анализируются, устраняются противоречия и определяется приоритет. После этого требования документируются, проверяются, согласовываются и утверждаются. На протяжении жизненного цикла изменения требований регистрируются и контролируются. Требования образуют иерархию. На верхнем уровне находятся цели организации и бизнес-требования. Далее формулируются требования заинтересованных сторон и пользователей. Они преобразуются в системные требования. Системные требования распределяются между подсистемами, программным обеспечением, оборудованием и организационными процедурами. Затем формируются детальные требования к отдельным компонентам. Например, бизнес-цель может заключаться в сокращении времени обработки заявки клиента. Пользовательское требование устанавливает возможность подачи заявки через личный кабинет. Системное требование определяет состав операций, взаимодействие с внешними системами и требования к времени обработки. Программное требование описывает конкретную функцию проверки заявки. Разделение уровней необходимо для обеспечения прослеживаемости. Разработчик должен понимать, какой потребностью обусловлено конкретное программное требование. Заказчик должен видеть, каким системным решением удовлетворяется его потребность.

3.5 Пользовательские требования

Пользовательские требования описывают цели, задачи, потребности и ожидаемые возможности системы с точки зрения ее пользователей. Они отражают то, что пользователь должен иметь возможность выполнить при помощи системы. Пользователем может быть не только человек, непосредственно работающий с интерфейсом. В широком смысле к пользователям относятся операторы, администраторы, сотрудники сопровождения и организации, получающие результат работы системы. Пользовательские требования должны формулироваться на понятном предметном языке. Они не должны быть перегружены программными и инфраструктурными подробностями, если эти подробности не являются осознанным ограничением пользователя. Например, корректное пользовательское требование может звучать так: «Сотрудник отдела кадров должен иметь возможность сформировать справку о трудовой деятельности выбранного работника». Формулировка «система должна выполнять запрос к таблице Employee через хранимую процедуру» не является пользовательским требованием. Она описывает деталь реализации. Пользовательские требования отвечают на вопросы: кто выполняет действие, какую цель он преследует, какую информацию использует, какой результат получает и какие ограничения существуют. Например, для библиотечной системы можно определить требование: «Библиотекарь должен иметь возможность зарегистрировать выдачу экземпляра книги читателю с проверкой наличия задолженности и допустимого количества одновременно выданных изданий». В этом требовании указаны роль пользователя, действие, объект, условия и ожидаемый результат. Однако еще не определено, какие программные компоненты и таблицы будут использоваться. Пользовательские требования могут описываться в форме текстовых положений, пользовательских сценариев, моделей бизнес-процессов, вариантов использования, прототипов интерфейса и других моделей. Пользовательский сценарий описывает последовательность взаимодействия пользователя с системой для достижения цели. Он может включать основной поток, альтернативные варианты и исключительные ситуации. Например, сценарий оформления заказа начинается с выбора товара, продолжается вводом адреса, выбором способа оплаты и подтверждением. Альтернативный поток возникает при отсутствии товара. Исключительный поток связан с отказом платежной системы. Вариант использования описывает взаимодействие участника с системой. Он определяет начальные условия, инициирующее событие, последовательность шагов, альтернативы и конечный результат. Пользовательские требования должны учитывать различные категории пользователей. У администратора, руководителя и рядового сотрудника разные задачи и права. Например, сотрудник может просматривать собственные заявки, руководитель — согласовывать заявки подразделения, а администратор — управлять справочниками и учетными записями. Ошибкой является формирование требований только на основе мнения одного представителя заказчика. Руководитель может хорошо понимать цели организации, но недостаточно знать детали повседневной работы. Рядовой пользователь знает операции, но может не учитывать стратегические и нормативные ограничения. Поэтому требуется участие нескольких групп. Пользовательские требования должны описывать не только нормальный сценарий, но и исключения. Если система регистрирует платеж, необходимо определить поведение при отказе банка, повторной отправке запроса, недостатке средств и потере связи. Важно учитывать контекст использования: рабочее место, квалификацию, частоту выполнения операций, физические условия и ограничения пользователя. Например, интерфейс складского терминала используется сотрудником в движении и может требовать крупных элементов управления, минимального количества действий и работы со сканером. Интерфейс аналитической системы используется специалистом за рабочим столом и может содержать сложные таблицы и фильтры. Пользовательское требование должно быть проверяемым на уровне пользовательского результата. Если система должна

поддерживать регистрацию заявки, приемочное испытание должно продемонстрировать, что пользователь действительно может создать, сохранить, отправить и найти заявку.

3.6 Системные требования

Системные требования представляют собой формализованное техническое описание функций, характеристик, интерфейсов, условий и ограничений системы в целом. Они создаются путем преобразования потребностей заинтересованных сторон и пользовательских требований в требования, пригодные для проектирования и проверки. Пользовательское требование обычно описывает цель пользователя. Системное требование определяет, что должна обеспечить система для достижения этой цели. Например, пользовательское требование может звучать так: «Покупатель должен иметь возможность оплатить заказ банковской картой». Из него выводятся системные требования: система должна сформировать платежный запрос; передать его во внешнюю платежную систему; получить результат; связать платеж с заказом; зарегистрировать отказ; исключить повторное списание; уведомить покупателя; сохранить сведения для аудита. Системное требование рассматривает информационную систему как целое. Оно может распределяться между программным обеспечением, оборудованием, данными, персоналом и организационными процедурами. Например, требование обеспечить сохранность данных может реализовываться программными механизмами контроля доступа, резервным оборудованием, регламентом резервного копирования и действиями администратора. Системные требования включают требования к функциям, производительности, надежности, безопасности, пользовательскому взаимодействию, данным, интерфейсам, эксплуатации, сопровождению и ограничениям проектирования. Требования к внешним интерфейсам определяют взаимодействие с пользователями, другими системами, техническими устройствами и каналами связи. Например, требование к внешнему интерфейсу может устанавливать состав данных, передаваемых в государственную информационную систему, формат сообщения, частоту обмена, механизм подтверждения и обработку ошибок. Требования к данным определяют состав информационных объектов, правила качества, сроки хранения, владельцев, источники и ограничения доступа. Эксплуатационные требования устанавливают условия установки, настройки, администрирования, резервного копирования, обновления и мониторинга. Ограничения определяют обязательные условия, в пределах которых проектируется система. Ограничением может быть использование существующей инфраструктуры, определенной операционной системы, утвержденного протокола, конкретной территории хранения данных или установленного бюджета. Системные требования должны быть согласованы с архитектурой. Архитектура распределяет требования между элементами системы и определяет механизмы их выполнения. Например, требование выдерживать десять тысяч одновременных пользователей может привести к использованию балансировки нагрузки, нескольких экземпляров приложения, кэширования и масштабируемого хранилища. Требование сохранять работоспособность при отказе одного сервера влияет на резервирование компонентов и развертывание системы. Требование обеспечить конфиденциальность персональных данных влияет на аутентификацию, разграничение доступа, шифрование, журналирование и архитектуру информационных потоков. Системные требования должны описывать внешне наблюдаемые свойства системы и ограничения, но не всегда должны преждевременно фиксировать внутреннюю реализацию. Если технология не предопределена, требование должно определять результат, а выбор способа оставаться архитектурным решением. Например, требование «система должна обеспечивать восстановление работоспособности не более чем за тридцать минут» предпочтительнее, чем «система должна использовать определенную программу резервного копирования», если применение этой программы не является обязательным ограничением заказчика.

3.7 Функциональные требования

Функциональные требования определяют функции, операции и поведение, которые должна выполнять система. Они описывают преобразование входных данных в результаты, реакцию на события, бизнес-правила и взаимодействие с пользователями или внешними системами. Функциональное требование отвечает на вопрос: что должна делать система? Например, система должна регистрировать пользователя, рассчитывать стоимость заказа, проверять наличие товара, формировать счет, сохранять документ, отправлять уведомление или предоставлять отчет. Функция может инициироваться действием пользователя, поступлением сообщения, наступлением времени или изменением состояния данных. Например, функция формирования ежемесячного отчета инициируется календарным событием. Функция блокировки учетной записи запускается после определенного количества неудачных попыток входа. Функция обновления остатка выполняется после регистрации складской операции. Функциональное требование должно содержать условие выполнения, действие системы и ожидаемый результат. Формулировка «система должна поддерживать заказы» является слишком общей. Более точно: «После подтверждения покупателем состава корзины система должна создать заказ, присвоить ему уникальный номер, зафиксировать состав товаров, цены, адрес доставки, способ оплаты и состояние „ожидает оплаты“». Функциональные требования могут быть представлены на разных уровнях детализации. На верхнем уровне определяется функция «управление заказами». Она декомпозируется на создание, изменение, отмену, оплату, отгрузку и закрытие заказа. Каждая функция может дополнительно разделяться на отдельные операции. Например, создание заказа включает проверку клиента, получение товаров из корзины, расчет стоимости, выбор доставки, резервирование и сохранение. Функциональные требования должны учитывать бизнес-правила. Бизнес-правило представляет собой ограничение или условие предметной области. Например, «скидка предоставляется только зарегистрированным клиентам», «заявка стоимостью более установленного порога требует согласования руководителем», «студент не допускается к экзамену при отсутствии зачета». Бизнес-правило может использоваться несколькими функциями. Поэтому его целесообразно документировать отдельно и связывать с соответствующими требованиями. Функциональные требования должны учитывать права пользователей. Одна и та же операция может быть доступна разным ролям с различными ограничениями. Например, автор документа может редактировать черновик, руководитель — утверждать документ, а архивариус — переводить его на хранение. Необходимо описывать обработку ошибок. Требование не считается полным, если оно отражает только успешный вариант. Например, при загрузке файла система должна проверить формат, размер и наличие вредоносного содержимого. При нарушении условия операция отклоняется, а пользователю предоставляется понятное сообщение. Функциональные требования могут описываться с помощью моделей процессов, диаграмм состояний, таблиц решений и сценариев. Диаграмма состояний полезна, когда объект проходит через последовательность состояний. Например, заказ может иметь состояния «создан», «ожидает оплаты», «оплачен», «передан в доставку», «доставлен», «отменен». Требования должны определять допустимые переходы. Нельзя передать неоплаченный заказ в доставку, если бизнес-процесс не допускает оплату при получении. Таблица решений применяется, когда результат зависит от сочетания нескольких условий. Например, размер скидки определяется категорией клиента, суммой заказа и наличием акции. Функциональные требования образуют основу для функционального тестирования. Для каждого требования должны быть определены условия, входные данные и ожидаемый результат. При этом наличие большого количества функций не означает высокое качество системы. Система может реализовать все операции, но быть медленной, ненадежной

или неудобной. Поэтому функциональные требования должны дополняться нефункциональными.

3.8 Документирование требований

Документирование требований представляет собой систематическую фиксацию требований, их источников, характеристик, связей, состояния и критериев проверки. Основная задача документирования заключается не в создании большого текстового документа как такового, а в формировании общего и контролируемого понимания будущей системы. Требования могут храниться в техническом задании, спецификации системных требований, спецификации требований к программному обеспечению, реестре требований, моделях, сценариях, таблицах, специализированной информационной системе или их сочетании. ISO/IEC/IEEE 29148:2018 устанавливает процессы инженерии требований и требования к информационным результатам, создаваемым в ходе этих процессов. ГОСТ Р 56713—2015 определяет содержание информационных продуктов жизненного цикла систем и программного обеспечения, включая документацию. В отечественном проекте автоматизированной системы значительная часть требований фиксируется в техническом задании по ГОСТ 34.602—2020. Требования могут дополнительно уточняться в проектной и программной документации. Каждое требование целесообразно снабжать уникальным идентификатором. Например, функциональное требование может иметь обозначение ФТ-015, а требование безопасности — БЗ-007. Идентификатор позволяет ссылаться на требование в проектной документации, программном коде, тестах, протоколах и запросах на изменение. Требование должно иметь наименование и текст, описывающий обязательное свойство или поведение. Дополнительно могут указываться источник, обоснование, приоритет, статус, версия, ответственный, способ проверки и связанные требования. Источник требования показывает, кто или что стало основанием для его появления. Источником может быть пользователь, закон, договор, стандарт, бизнес-процесс или внешняя система. Обоснование объясняет, почему требование необходимо. Это особенно важно, если требование ограничивает архитектуру или повышает стоимость. Например, требование хранить журнал операций пять лет может быть обусловлено нормативным актом. Без обоснования разработчик может ошибочно считать срок произвольным и предложить его сократить. Приоритет определяет относительную важность. Требования могут быть обязательными, желательными и дополнительными. Однако приоритет не должен подменять договорную ясность. Если все требования объявлены максимальными, приоритизация теряет смысл. Статус отражает состояние требования: предложено, анализируется, согласовано, реализовано, проверено, отклонено или изменено. Критерий приемки определяет, каким наблюдаемым результатом подтверждается выполнение требования. Например, для требования о времени отклика критерием может быть результат нагрузочного испытания. Для требования о формировании отчета — совпадение результата с контрольным набором данных. Требования должны обладать рядом качеств. Необходимость означает, что требование связано с реальной потребностью или обязательным ограничением. Ненужные требования увеличивают стоимость системы. Однозначность означает, что требование допускает одно разумное толкование. Следует избегать слов «быстро», «удобно», «современно», «при необходимости» и «по возможности» без пояснения критериев. Полнота означает наличие всех необходимых условий, входных данных, результатов и исключений. Непротиворечивость требует отсутствия конфликтов между требованиями. Например, одно требование не может предписывать автоматическое удаление данных через год, если другое требует хранить их пять лет. Единичность означает, что одно требование описывает одно основное обязательство. Формулировка «система должна зарегистрировать пользователя, отправить письмо, создать отчет и обновить справочник» содержит несколько требований и должна быть разделена. Реализуемость означает возможность выполнения требования при существующих технологиях, ресурсах, сроках и ограничениях. Проверяемость означает возможность объективно подтвердить выполнение. Прослеживаемость означает нали-

чие связей требования с источником, более высокоуровневыми требованиями, проектными решениями, компонентами и тестами. Независимость от реализации означает, что требование по возможности описывает требуемый результат, а не преждевременно выбранный способ. Исключение составляют обязательные проектные ограничения. Для формулирования требований часто используется конструкция: «Система должна…». Она подчеркивает обязательность. Однако одного грамматического шаблона недостаточно. Требование должно содержать точный смысл. Плохая формулировка: «Система должна быстро сохранять документы». Улучшенная формулировка: «При размере документа до 20 мегабайт система должна завершать сохранение не более чем за три секунды для 95 процентов операций при одновременной работе до 500 пользователей». Второй вариант определяет объект, условия, показатель и допустимое значение. Документация требований должна различать обязательные требования и справочную информацию. Если пояснение, пример и требование оформлены одинаково, участники могут неправильно определить, что подлежит реализации и приемке. Необходимо документировать предположения. Например, проект может исходить из того, что внешняя система доступна круглосуточно или предоставляет определенный интерфейс. Если предположение окажется неверным, архитектуру придется изменить. Необходимо фиксировать ограничения. Ограничения могут быть нормативными, техническими, финансовыми, временными или организационными. Особое значение имеет матрица прослеживаемости требований. Она показывает связи между требованиями различных уровней и результатами разработки. Например, пользовательское требование ПТ-01 связано с системными требованиями СТ-11 и СТ-12. Они реализуются компонентами К-03 и К-07 и проверяются тестами Т-21, Т-22 и Т-25. Прослеживаемость позволяет определить, все ли потребности реализованы, почему существует определенный компонент, какие тесты необходимо повторить после изменения и какие части системы затрагиваются. Если изменяется требование безопасности, матрица позволяет найти связанные архитектурные решения, программные модули, инструкции и испытания. Документирование требований не является разовой работой. Требования изменяются, поэтому необходимо управление требованиями. Каждый запрос на изменение должен регистрироваться. Оцениваются причина, приоритет, влияние на стоимость, сроки, архитектуру, данные, документацию и испытания. После согласования обновляются связанные материалы. Неконтролируемое изменение приводит к расхождению между фактической системой и документацией. Например, программа может реализовать новую функцию, но техническое задание и руководство пользователя останутся прежними. Требования должны проходить проверку и валидацию. Проверка определяет качество формулировок: полноту, непротиворечивость и проверяемость. Валидация подтверждает, что требования отражают реальные потребности заинтересованных сторон. Для проверки используются рецензирование, совместные обсуждения, прототипы, моделирование, формальная проверка и подготовка тестов. Разработка тестов на этапе анализа требований помогает обнаружить непроверяемые формулировки. Если тестировщик не может определить ожидаемый результат, требование нуждается в уточнении. Прототип пользовательского интерфейса помогает проверить понимание сценариев. Однако прототип не заменяет требования. Он должен быть связан с текстовыми и модельными описаниями. Документация должна быть управляемой по версиям. Необходимо определять актуальную утвержденную редакцию, историю изменений и участников согласования.

3.9 Нефункциональные требования

Нефункциональные требования определяют характеристики качества системы, условия ее функционирования и ограничения реализации. Они отвечают не столько на вопрос, что делает система, сколько на вопросы: насколько хорошо она выполняет функции, в каких условиях работает, какие ограничения соблюдает и каким уровнем качества обладает. Термин «нефункциональные требования» условен. Эти требования не являются менее важными или необязательными. Во многих системах именно они определяют архитектуру и стоимость. Например, функция перевода денежных средств может быть реализована сравнительно просто. Однако требования к безопасности, целостности, доступности, производительности, аудиту и восстановлению превращают ее в сложную банковскую функцию. Нефункциональные требования часто называют требованиями к качеству, атрибутами качества или ограничениями. Эти понятия связаны, но не полностью тождественны. Требование «система должна обрабатывать тысячу запросов в секунду» относится к производительности. Требование «система должна использовать сертифицированное средство защиты» является ограничением. Требование «система должна хранить данные на территории определенного государства» является нормативно-организационным ограничением. Модель качества ISO/IEC 25010:2023 включает девять характеристик качества продукта: функциональную пригодность, эффективность функционирования, совместимость, способность к взаимодействию с пользователем, надежность, защищенность, сопровождаемость, гибкость и безопасность эксплуатации. Стандарт рассматривает эти характеристики как основу для определения, измерения и оценки качества информационно-коммуникационных и программных продуктов.

3.9.1 Функциональная пригодность

Функциональная пригодность показывает, насколько функции продукта удовлетворяют установленным и подразумеваемым потребностям при использовании в заданных условиях. Она включает полноту функций, правильность результатов и уместность функций для достижения целей. Функциональная пригодность связана с функциональными требованиями, но не совпадает с ними. Функциональные требования перечисляют конкретные операции. Функциональная пригодность оценивает совокупное качество этих операций. Например, система может формально содержать функцию расчета налога, но выдавать неточный результат. Функция существует, однако функциональная правильность нарушена.

3.9.2 Эффективность функционирования

Эффективность функционирования характеризует соотношение между производительностью и используемыми ресурсами. К ней относятся время отклика, пропускная способность, использование вычислительных ресурсов и предельная емкость. Требование производительности должно включать операцию, нагрузку, условия и измеряемый показатель. Плохая формулировка: «Система должна обладать высокой производительностью». Корректная формулировка: «При одновременной работе 1000 пользователей время ответа на операцию поиска не должно превышать двух секунд для 95 процентов запросов». Время отклика представляет собой период между запросом пользователя и предоставлением результата. Пропускная способность показывает количество операций, обрабатываемых за единицу времени. Емкость определяет предельный объем пользователей, данных, подключений или операций. Ресурсная эффективность отражает использование процессора, оперативной памяти, сети и хранилища. Производительность должна оцениваться в условиях, соответствующих реальной эксплуатации. Результат теста на пустой базе данных не подтверждает работу системы при многолетнем объеме информации.

3.9.3 Совместимость

Совместимость характеризует способность продукта работать совместно с другими системами и обмениваться информацией. Она включает сосуществование и функциональную совместимость. Сосуществование означает способность нескольких продуктов использовать общую среду и ресурсы без недопустимого взаимного влияния. Функциональная совместимость означает способность обмениваться информацией и использовать полученные данные. Например, медицинская система должна передавать результаты исследований в региональную систему в установленном формате и корректно обрабатывать подтверждения. Требования совместимости должны определять взаимодействующие системы, данные, протоколы, периодичность, безопасность, обработку ошибок и ответственность сторон.

3.9.4 Способность к взаимодействию с пользователем

Данная характеристика охватывает свойства, обеспечивающие результативное, эффективное и удовлетворяющее пользователя взаимодействие с системой. К ней относятся понятность назначения, обучаемость, управляемость, защита от пользовательских ошибок, вовлеченность, доступность для различных групп пользователей и помощь пользователю. Требование «интерфейс должен быть удобным» непроверяемо. Необходимо определять конкретные показатели. Например, новый сотрудник после двухчасового обучения должен выполнить основной сценарий без помощи инструктора. Пользователь должен иметь возможность отменить ошибочно начатую операцию до ее подтверждения. Все обязательные поля должны быть явно обозначены. Защита от пользовательских ошибок включает предотвращение неправильных действий и понятную обработку ошибок. Например, перед необратимым удалением система должна запросить подтверждение и указать последствия. Доступность означает возможность использования системы людьми с различными физическими и сенсорными особенностями. Могут устанавливаться требования к управлению с клавиатуры, контрастности, текстовым описаниям и масштабированию.

3.9.5 Надежность

Надежность представляет собой способность системы выполнять требуемые функции в заданных условиях в течение установленного времени. Она включает безошибочность, доступность, отказоустойчивость и восстанавливаемость. Доступность показывает долю времени, в течение которого система готова к использованию. Требование может устанавливать: «Доступность системы должна составлять не менее 99,9 процента в месяц, за исключением согласованных периодов обслуживания». Необходимо определить, как рассчитывается показатель. Включаются ли плановые работы, с какого момента начинается отказ, какие компоненты учитываются. Отказоустойчивость означает способность продолжать работу при отказе отдельных компонентов. Например, при недоступности сервиса рекомендаций интернет-магазин может продолжать оформление заказов без персональных предложений. Восстанавливаемость определяет способность восстановить работу и данные после отказа. Используются показатели допустимого времени восстановления и допустимой точки потери данных. Например, система должна быть восстановлена не более чем за тридцать минут, а потеря подтвержденных данных не должна превышать пять минут.

3.9.6 Защищенность

Защищенность характеризует способность системы обеспечивать конфиденциальность, целостность, подлинность, учет действий и другие свойства защиты информации. Конфиденциальность означает доступность информации только уполномоченным субъектам. Целостность означает защиту данных и программ от несанкционированного изменения. Подлинность

означает возможность подтвердить идентичность пользователя, системы или данных. Подотчетность обеспечивает связь действий с конкретным субъектом. Неотказуемость позволяет подтвердить совершение действия и препятствует необоснованному отказу от него. Требования безопасности должны основываться на анализе угроз и нормативных обязательствах. Например, система должна блокировать учетную запись после установленного количества неудачных попыток входа, регистрировать административные действия, шифровать конфиденциальные данные и ограничивать доступ по ролям. Формулировка «система должна быть безопасной» не имеет проверяемого смысла. Требование к журналированию должно определять регистрируемые события, состав записей, срок хранения, защиту журнала и права доступа.

3.9.7 Сопровождаемость

Сопровождаемость характеризует возможность анализа, изменения, тестирования и повторного использования системы. Она включает модульность, анализируемость, изменяемость, тестируемость и повторное использование. Модульность означает такое разделение системы, при котором изменение одного компонента оказывает ограниченное влияние на другие. Анализируемость отражает возможность определить причины ошибок и последствия изменения. Изменяемость характеризует затраты и риски внесения изменений. Тестируемость означает возможность установить критерии проверки и контролировать состояние системы. Сопровождаемость трудно оценить только по пользовательскому интерфейсу, но она существенно влияет на стоимость жизненного цикла. Требования могут устанавливать наличие автоматических тестов, документации интерфейсов, журналирования, модульной структуры и правил программирования. Например, «для всех внешних программных интерфейсов должна поддерживаться версионируемая документация» или «изменение настройки налоговой ставки не должно требовать изменения программного кода».

3.9.8 Гибкость

Гибкость в модели ISO/IEC 25010:2023 связана со способностью продукта адаптироваться, масштабироваться, устанавливаться и заменяться в различных средах. Она охватывает адаптируемость, масштабируемость, устанавливаемость и заменяемость. Адаптируемость показывает возможность приспособления к другим условиям. Масштабируемость означает способность увеличивать производительность или емкость при росте нагрузки. Устанавливаемость отражает возможность установки, удаления и обновления. Заменяемость характеризует возможность заменить другой продукт или быть замененным. Например, требование масштабируемости может устанавливать, что увеличение вычислительных ресурсов вдвое должно обеспечивать увеличение пропускной способности не менее чем на определенную величину. Требование адаптируемости может предусматривать настройку языка, часового пояса, справочников и бизнес-правил без изменения исходного кода.

3.9.9 Безопасность эксплуатации

Безопасность эксплуатации означает способность продукта не создавать недопустимого риска причинения вреда людям, имуществу или окружающей среде. Эта характеристика особенно важна для медицинских, транспортных, промышленных и энергетических систем. Например, отказ системы управления оборудованием не должен приводить к неконтролируемому опасному состоянию. При потере связи оборудование должно перейти в безопасный режим. Требования безопасности эксплуатации могут включать выявление опасностей, ограничения операций, предупреждения, аварийное отключение, резервные режимы и контроль критических действий. Необходимо различать защищенность информации и безопасность эксплуатации. Защищенность связана преимущественно с противодействием несанкционированным воздействиям. Безопасность эксплуатации связана с предотвращением физического и иного вреда.

3.9.10 Дополнительные группы нефункциональных требований

Практическая классификация нефункциональных требований может включать более широкий набор групп, чем модель качества продукта. Требования к данным определяют точность, полноту, актуальность, уникальность, согласованность, происхождение, сроки хранения, архивирование и удаление данных. Например, сведения о складских остатках должны обновляться после подтверждения каждой операции. Дублирующиеся записи клиентов должны выявляться по установленным правилам. Эксплуатационные требования определяют порядок установки, настройки, резервного копирования, мониторинга и обновления. Требования к наблюдаемости устанавливают состав журналов, показателей и средств диагностики. Например, администратор должен получать уведомление, если время ответа превышает допустимое значение или свободное место в хранилище становится меньше установленного порога. Требования к совместимости со средой определяют операционные системы, браузеры, устройства и техническую инфраструктуру. Правовые и нормативные требования возникают из законодательства, отраслевых правил, договоров и стандартов. Организационные требования определяют роли, ответственность, регламенты и порядок взаимодействия. Экономические ограничения устанавливают бюджет, стоимость владения и допустимые расходы на инфраструктуру. Временные ограничения определяют срок разработки, периоды внедрения и допустимые окна обслуживания.

3.10 Взаимосвязь требований и архитектуры

Требования являются основным входом для архитектурного проектирования. Однако не каждое требование одинаково влияет на архитектуру. Архитектурно значимое требование представляет собой требование, которое существенно влияет на структуру системы, выбор компонентов, технологии или способы взаимодействия. Например, требование изменить название поля обычно не является архитектурно значимым. Требование обеспечить работу в нескольких географических регионах влияет на размещение компонентов, данные, сетевое взаимодействие и отказоустойчивость. К архитектурно значимым обычно относятся требования к высокой производительности, доступности, безопасности, масштабируемости, интеграции, изменяемости и ограничениям инфраструктуры. Архитектор должен определить, какие решения обеспечивают выполнение требований. Затем решения документируются и связываются с требованиями. Например, требование высокой доступности может реализовываться через резервирование серверов, балансировку нагрузки, репликацию данных и автоматическое переключение. Одно архитектурное решение может удовлетворять несколько требований. Например, очередь сообщений может повышать устойчивость, снижать связанность компонентов и обеспечивать асинхронную обработку. Одновременно решение может создавать отрицательные последствия. Распределение системы повышает масштабируемость, но усложняет согласованность данных и эксплуатацию. Поэтому требования должны содержать приоритеты и количественные критерии. Невозможно одновременно максимизировать скорость, надежность, безопасность, простоту и минимальную стоимость. Архитектура является системой компромиссов. Требования позволяют определить, какие свойства имеют наивысший приоритет.

3.11 Проверка полноты системы требований

Полнота требований не означает максимально возможное количество положений. Она означает наличие требований, необходимых для определения и приемки системы. Необходимо проверить наличие целей, границ, пользователей, функций, данных, интерфейсов, характеристик качества, условий эксплуатации, безопасности, сопровождения и ограничений. Следует проверить, описаны ли основные и исключительные сценарии, обработка ошибок, права пользователей и взаимодействие с внешними системами. Необходимо проверить согласованность единиц измерения, терминов, ролей и состояний объектов. Если в одном разделе используется термин «клиент», а в другом — «покупатель», необходимо установить, являются ли они одним участником или различными ролями. Требования должны быть согласованы с моделями. Если диаграмма процесса предусматривает согласование руководителем, соответствующая функция и роль должны присутствовать в требованиях. Необходимо проверить реализуемость и стоимость. Некоторые требования могут быть технически возможны, но экономически неоправданны. Например, требование абсолютной доступности без перерывов невозможно выполнить буквально. Даже резервированные системы имеют вероятность отказа. Следует установить измеримый допустимый уровень. Необходимо проверить возможность приемки. Если критерий выполнения отсутствует, между заказчиком и разработчиком может возникнуть спор. Грамотно сформированная система требований обеспечивает основу архитектуры, разработки, тестирования, внедрения и сопровождения. Недостатки требований распространяются на все последующие стадии. Ошибка в программном коде обычно затрагивает отдельную функцию, тогда как ошибка в базовом требовании может привести к созданию системы, которая в целом не решает задачу пользователя. Стандартизация процессов и требований позволяет уменьшить этот риск. Отечественные стандарты определяют стадии создания автоматизированных систем, состав технического задания и содержание документов. Международные и гармонизированные стандарты формируют общую систему процессов жизненного цикла, инженерии требований, архитектурного описания и качества программного продукта. Пользовательские требования описывают цели и задачи пользователей. Системные требования преобразуют эти потребности в технически точное описание системы. Функциональные требования определяют операции и поведение. Нефункциональные требования устанавливают характеристики качества и ограничения. Документирование обеспечивает однозначность, проверяемость, прослеживаемость и управляемость изменений. Архитектура информационной системы должна формироваться не на основе случайного выбора технологий, а на основе согласованной системы требований. Именно требования объясняют, почему выделены определенные компоненты, каким образом организованы данные, зачем применяются механизмы резервирования, какие интерфейсы необходимы и какими характеристиками должна обладать система в процессе эксплуатации.

4. Методологии проектирования информационных систем

Методология проектирования информационной системы представляет собой организованную совокупность принципов, методов, моделей, процессов, ролей, правил, документов и инструментальных средств, применяемых для создания, развития и сопровождения информационной системы. Методология определяет не только последовательность выполнения работ, но и способы принятия решений, распределения ответственности, взаимодействия участников, управления требованиями, контроля качества и оценки полученных результатов. Проектирование информационной системы является сложной коллективной деятельностью. В нем участвуют заказчики, пользователи, владельцы бизнес-процессов, аналитики, архитекторы, разработчики, тестировщики, специалисты по данным, администраторы, инженеры по эксплуатации и информационной безопасности. Без согласованной методологии каждый участник может организовывать работу по собственному усмотрению. Аналитик будет формировать требования в одной форме, архитектор — разрабатывать модели в другой, разработчик — принимать самостоятельные технические решения, а тестировщик — проверять систему без ясных критериев приемки. В результате увеличиваются сроки, стоимость, количество дефектов и риск создания системы, не соответствующей потребностям организации. Методология устанавливает общий порядок перехода от потребности к работающей информационной системе. В ее рамках определяются способы обследования предметной области, выявления требований, моделирования процессов, проектирования архитектуры, выбора технологий, разработки программного обеспечения, проведения испытаний, внедрения и сопровождения. Необходимо различать понятия методологии, метода, процесса, практики, архитектурного подхода и инструмента. Методология охватывает систему организации деятельности в целом. Метод определяет способ решения определенного класса задач. Процесс представляет собой последовательность взаимосвязанных действий, преобразующих входные данные в результат. Практика является регулярно применяемым приемом работы. Архитектурный подход определяет принципы структурирования системы. Инструмент представляет собой программное или техническое средство, поддерживающее выполнение работы. Например, гибкая разработка, или Agile, является системой ценностей и принципов. Скрам, или Scrum, является каркасом организации работы над сложным продуктом. Канбан, или Kanban, является методом управления потоком работ и эволюционного совершенствования процессов. DevOps представляет собой культурный и инженерный подход, объединяющий разработку и эксплуатацию. Система управления задачами, среда моделирования или сервер непрерывной интеграции являются инструментами, но сами по себе не образуют методологию. Использование современной программы для управления задачами не делает процесс гибким. Команда может вести доску задач, но продолжать работать крупными партиями, редко показывать результат пользователям и запрещать изменение требований. Аналогично применение контейнеров и системы автоматизированного развертывания не означает полноценного внедрения DevOps, если разработчики не отвечают за эксплуатационные характеристики продукта, а обновления по-прежнему передаются между изолированными подразделениями вручную. Методология должна соответствовать особенностям проекта. Для информационной системы с устойчивыми требованиями и обязательной формальной приемкой может использоваться стадийная модель с подробной документацией. Для цифрового продукта, требования к которому меняются на основе поведения пользователей, необходимы короткие циклы, экспериментальная проверка гипотез и регулярная поставка новых версий. Для критической системы требуется сочетание гибкости с формальными проверками, управлением рисками, прослеживаемостью требова-

ний и подтверждением безопасности. Современные методологии развиваются в направлении сокращения длительности обратной связи. Чем быстрее участники получают информацию об ошибке требования, архитектурного решения, программного компонента или эксплуатационной настройки, тем дешевле исправление. Поэтому в современных подходах анализ, проектирование, разработка, тестирование, развертывание и наблюдение за работой системы стремятся объединить в непрерывный управляемый цикл.

4.1 Понятие гибкой разработки информационных систем

Гибкая разработка, обозначаемая международным термином Agile, представляет собой подход к созданию программных продуктов, основанный на итерационном развитии, тесном взаимодействии участников, регулярной поставке работающего результата и готовности изменять планы по мере появления новой информации. Agile не является одной конкретной методологией. Это система ценностей и принципов, на основе которой существуют различные методы и организационные каркасы. К гибким подходам относятся Scrum, Kanban, экстремальное программирование, разработка, управляемая функциональностью, семейство методов Crystal и другие подходы. Основой гибкой разработки является Манифест гибкой разработки программного обеспечения, сформулированный в 2001 году. Он устанавливает четыре ценностных приоритета: люди и взаимодействие важнее процессов и инструментов; работающий программный продукт важнее исчерпывающей документации; сотрудничество с заказчиком важнее формального согласования условий договора; готовность к изменениям важнее строгого следования первоначальному плану. Правая часть каждого сопоставления также признается важной, однако при конфликте приоритет отдается левой части. Данные ценности иногда толкуются неправильно. Agile не отрицает процессы, инструменты, документы, договоры и планирование. Он требует, чтобы они поддерживали создание полезного результата, а не становились самостоятельной целью. Документ, который необходим для эксплуатации, безопасности или передачи знаний, должен быть подготовлен. Однако создание сотен страниц документации, которая не используется и быстро устаревает, не создает ценности. Гибкая разработка основана на итерационности и инкрементности. Итерационность означает многократное уточнение продукта. Команда не пытается сразу создать окончательное решение, а последовательно развивает его на основе полученных результатов и обратной связи. Инкрементность означает создание системы функциональными приращениями. Каждое приращение добавляет определенную полезную возможность. Например, разработка медицинской информационной системы может быть разделена на несколько приращений. Сначала реализуется регистрация пациента и ведение расписания, затем электронная медицинская карта, после этого обмен результатами лабораторных исследований, электронные назначения и аналитическая отчетность. Каждое приращение должно быть интегрировано с уже созданной системой и пригодно для проверки. Итерационность не означает бесконечное переделывание. Каждая итерация должна давать новую информацию и приближать продукт к цели. Если команда постоянно изменяет одни и те же функции без оценки результата и ясной цели, это свидетельствует не о гибкости, а о слабом управлении требованиями. Одним из центральных понятий Agile является ценность продукта. Работа оценивается не только по количеству написанного программного кода, подготовленных документов или закрытых задач, но и по тому, насколько созданная возможность решает проблему пользователя и способствует достижению целей организации. Например, команда может реализовать сложный отчет, который формально соответствует требованиям, но практически не используется руководителями. С точки зрения объема выполненной работы функция завершена, однако ее полезная ценность ограничена. Гибкий подход требует регулярно проверять не только создание функции, но и ее фактическую востребованность. Следующим принципом является ранняя и регулярная поставка работающего результата. Чем дольше система существует только в виде требований, моделей и программного кода отдельных компонентов, тем выше риск позднего обнаружения неправильных решений. Работающая версия позволяет проверить интерфейс, бизнес-логику, производительность, интеграции и пользовательские сценарии. Работающий результат не обязательно сразу передается всем пользователям. Он может демонстрироваться представителям заказчика, проходить внутреннее тестирование, использоваться ограниченной пилотной группой или размещаться в опытной среде.

Главное заключается в том, что результат должен быть интегрированным и проверяемым. Гибкая разработка предусматривает непрерывное уточнение требований. В традиционной модели требования стараются полностью определить до начала реализации. В Agile признается, что часть требований становится понятной только в процессе взаимодействия с продуктом. Это не означает отсутствия исходного анализа. До начала разработки необходимо определить проблему, цели, границы продукта, ключевых пользователей, основные риски и архитектурные ограничения. Однако детальная проработка всех функций на несколько лет вперед может оказаться неэффективной, поскольку часть предположений изменится раньше, чем функции будут реализованы. Одним из инструментов организации требований является перечень работ продукта, или бэклог продукта. Он содержит упорядоченные функции, изменения, исправления, исследования и технические работы, необходимые для развития продукта. Порядок элементов отражает их относительную важность и очередность рассмотрения. Бэклог не должен превращаться в неконтролируемое хранилище пожеланий. Его элементы должны регулярно анализироваться, объединяться, разделяться, уточняться, переоцениваться и удаляться, если они больше не соответствуют целям продукта. Для описания пользовательской потребности часто применяется пользовательская история. Она может быть сформулирована по схеме: «Как определенный пользователь, я хочу выполнить определенное действие, чтобы получить определенную пользу». Например: «Как заведующий складом, я хочу видеть товары с критически низким остатком, чтобы своевременно сформировать заказ поставщику». Пользовательская история не является полной спецификацией. Она представляет краткое описание потребности и основание для дальнейшего обсуждения. К ней добавляются критерии приемки, бизнес-правила, модели данных, макеты интерфейса и другие сведения, необходимые для реализации и проверки. Критерии приемки устанавливают условия, при выполнении которых функция считается соответствующей ожиданиям. Например, для истории о низком остатке критерии могут определять способ расчета порога, перечень отображаемых полей, правила фильтрации и частоту обновления сведений. Гибкая разработка использует приоритизацию требований. Не все функции имеют одинаковую ценность и срочность. Приоритет может определяться влиянием на цели продукта, риском, стоимостью задержки, обязательностью, зависимостями и трудоемкостью. Распространенной ошибкой является объявление всех задач критически важными. Если каждый элемент имеет высший приоритет, реальная приоритизация отсутствует. Необходимо принимать решения о том, какие функции создаются раньше, а какие могут быть отложены или исключены. Для оценки объема работы могут использоваться относительные единицы. Команда сравнивает задачи между собой по сложности, неопределенности и объему. Одним из методов является планирование с использованием карточек оценивания, при котором участники независимо предлагают оценки, а затем обсуждают различия. Относительная оценка не является точным измерением времени. Она помогает сравнивать задачи и планировать доступный объем работы, но не должна использоваться для оценки индивидуальной производительности сотрудников. В гибких командах применяется понятие скорости выполнения, показывающее объем работы, завершенный командой за определенный период. Скорость может использоваться для внутреннего прогнозирования, но сравнивать по ней разные команды некорректно. Различные команды используют разные шкалы, имеют разный состав работ и различную техническую среду. Важным инструментом является минимально жизнеспособный продукт. Он представляет собой минимальный вариант решения, достаточный для проверки ключевого предположения и получения содержательной обратной связи. Минимально жизнеспособный продукт не должен быть некачественным или небезопасным. Минимальность относится к объему функций, а не к обязательным характеристикам качества. Например, перед разработкой полной системы электронного согласования документов организация может создать решение для одного типа заявления и одного подразделения. Его использование покажет, соответствует ли выбранная схема согласования реальному процессу.

Другим инструментом является техническое исследование, иногда называемое исследовательской задачей. Оно используется, когда команда не может достоверно оценить технологическое решение. В ходе ограниченного по времени исследования создается прототип, выполняется эксперимент или анализируется документация. Например, до выбора технологии распознавания документов можно проверить точность нескольких решений на реальном наборе материалов. Результатом исследования являются знания и решение, а не обязательно готовая функция. Гибкая разработка основывается на коротких циклах обратной связи. Обратная связь поступает от пользователей, тестов, эксплуатации, аналитики, мониторинга и других участников. Чем быстрее команда узнает о проблеме, тем меньше объем работы, построенной на неверном предположении. К гибким инструментам относятся демонстрации продукта, ретроспективы, ежедневная координация, автоматические тесты, анализ пользовательского поведения, ограниченные эксперименты и регулярное уточнение бэклога. Ретроспектива направлена на улучшение способа работы команды. Участники анализируют, что помогало достижению результата, какие препятствия возникли и какое изменение процесса следует проверить в следующем периоде. Ретроспектива не должна становиться формальным обсуждением без последующих действий. Ее результатом должно быть одно или несколько конкретных улучшений. Например, команда может изменить порядок проверки кода, сократить размер задач, автоматизировать тестирование или уточнить правила взаимодействия с заказчиком. Гибкость требует дисциплины. Чем чаще изменяется продукт, тем важнее автоматизация тестирования, управление версиями, модульная архитектура, качественный программный код и актуальная документация. Без этих условий каждое изменение становится опасным и дорогостоящим.

4.1.1 Бережливый подход Lean

Lean, или бережливый подход, представляет собой систему мышления и управления, направленную на создание необходимой потребителю ценности с использованием меньшего количества ресурсов и с сокращением потерь. Lean рассматривается не как разовый проект оптимизации, а как постоянная практика экспериментального улучшения процессов. Исторически идеи Lean развивались в промышленном производстве, однако впоследствии были адаптированы для управления услугами, разработкой продуктов и информационными технологиями. В проектировании информационных систем Lean помогает рассматривать весь поток создания ценности — от появления потребности до предоставления работающей функции пользователю. Центральным понятием Lean является ценность. Ценность определяется не разработчиком и не количеством выполненных действий, а потребностью конечного пользователя или заказчика. Работа, которая не способствует достижению требуемого результата и не является необходимой для обеспечения качества, безопасности или нормативного соответствия, должна рассматриваться как потенциальная потеря. Например, ручное копирование данных между несколькими системами не создает самостоятельной ценности для клиента. Оно может быть временно необходимым из-за отсутствия интеграции, но должно рассматриваться как потеря, подлежащая устранению. Следующим понятием является поток создания ценности. Он включает все действия от формирования потребности до получения пользователем результата. При анализе потока учитываются не только непосредственные операции разработки, но и ожидание, согласования, передачи между подразделениями, повторная работа и исправление дефектов. Например, создание небольшой функции может занимать два дня программирования, но три месяца проходить через согласование, распределение ресурсов, тестирование и ожидание выпуска. Оптимизация только скорости программирования почти не изменит общий срок. Lean требует анализировать весь поток. Для изображения потока применяется карта потока создания ценности. На ней отражаются стадии прохождения работы, длительность активной обработки, время ожидания, очереди, передачи и возвраты. Карта позволяет определить, где возникает основная задержка. Одним из принципов Lean является обеспече-

ние непрерывного потока. Работа должна по возможности равномерно перемещаться через систему без длительных остановок, больших очередей и повторных передач. В программной разработке поток нарушается, когда аналитики создают огромный пакет требований, затем передают его проектировщикам, проектировщики формируют крупный пакет документации, а разработчики получают результаты через несколько месяцев. Ошибки и вопросы накапливаются, а обратная связь появляется поздно. Другим принципом является вытягивающая система. Новая работа начинается тогда, когда у исполнителя имеется возможность ее выполнить, а не просто потому, что задача была поставлена. Это позволяет ограничивать объем одновременно начатой работы. Если каждый сотрудник начинает множество задач, формально все задачи находятся «в работе», но ни одна не завершается. Ограничение незавершенной работы способствует концентрации и сокращает время прохождения задачи. Lean использует идею непрерывного совершенствования, часто обозначаемую термином кайдзен. Улучшение рассматривается как регулярная деятельность всех участников, а не исключительно управленческая инициатива. Небольшое изменение, сокращающее ручную операцию на несколько минут, может показаться незначительным. Однако если операция выполняется сотни раз, эффект становится существенным. Непрерывное совершенствование складывается из множества таких изменений. Важным принципом является уважение к людям. Исполнители, непосредственно выполняющие работу, обладают знаниями о реальных проблемах процесса. Поэтому изменения не должны полностью разрабатываться внешней группой и навязываться сверху. Сотрудники должны участвовать в анализе и улучшении собственного процесса. В программной разработке к потерям относят незавершенную работу, создание ненужных функций, ожидание, лишние передачи между участниками, повторное получение уже имевшихся знаний, постоянное переключение между задачами и дефекты. Незавершенная работа включает требования, модели, программные компоненты и функции, которые были начаты, но еще не приносят пользу. Она требует учета, устаревает и создает дополнительные зависимости. Избыточные функции являются возможностями, созданными без подтвержденной потребности. Каждая такая функция требует проектирования, тестирования, документирования, поддержки и обеспечения безопасности. Ожидание возникает при согласовании, передаче задачи, недоступности тестовой среды, длительной проверке кода или ожидании решения заказчика. Передачи создают риск потери контекста. Если аналитик передает документ архитектору, архитектор — разработчику, разработчик — тестировщику, а тестировщик — эксплуатационной группе, каждая передача требует объяснения и может привести к искажению смысла. Переключение между задачами снижает концентрацию и увеличивает время завершения. Сотрудник вынужден восстанавливать контекст каждой задачи. Дефекты требуют повторной работы. Особенно дорого обходятся дефекты требований и архитектуры, обнаруженные на поздней стадии. Lean не требует полного устранения всех вспомогательных действий. Некоторые из них не создают прямой пользовательской ценности, но необходимы. Например, резервное копирование, проверка безопасности и документирование могут быть обязательными для надежной эксплуатации. Задача заключается в том, чтобы выполнять необходимые действия эффективно и исключать необоснованные. Одним из инструментов Lean является метод «пять почему». При возникновении проблемы последовательно задается вопрос о ее причине, чтобы перейти от непосредственного проявления к корневому фактору. Например, выпуск системы задержался, потому что интеграционное тестирование началось поздно. Тестирование началось поздно, потому что среда не была подготовлена. Среда не была подготовлена, потому что ее создание выполнялось вручную. Ручной порядок сохранялся, потому что инфраструктура не была описана программно. В результате улучшение должно быть направлено не только на ускорение тестировщиков, но и на автоматизацию подготовки среды. Другим инструментом является стандартизированная работа. Для повторяющихся действий определяется лучший известный на текущий момент способ выполнения. Стандарт не является неизменным правилом: он слу-

жит исходной точкой для дальнейшего улучшения. В информационных технологиях стандартизироваться могут правила проверки кода, выпуск версии, обработка инцидента, создание нового сервиса, оформление архитектурного решения и настройка мониторинга. Lean тесно связан с уменьшением размера партии. Небольшие изменения проще анализировать, тестировать и выпускать. При возникновении ошибки легче определить ее источник и выполнить откат. Например, выпуск одной версии, содержащей двести изменений, создает высокий риск и требует сложного тестирования. Регулярные небольшие выпуски позволяют быстрее получать обратную связь и уменьшают объем потенциально ошибочной работы. Вережливый подход не следует смешивать с простым сокращением затрат. Увольнение сотрудников или отказ от тестирования может временно уменьшить расходы, но увеличить потери, дефекты и длительность работы. Lean направлен на устранение причин неэффективности, а не на механическое сокращение ресурсов.

4.1.2 Соотношение Agile и Lean

Agile и Lean имеют общие идеи: ориентацию на ценность, сокращение времени обратной связи, небольшие партии работы, постоянное совершенствование и участие команды. Однако они не являются тождественными. Agile возник прежде всего как ответ на проблемы разработки программного обеспечения в условиях изменяющихся требований. Lean представляет более широкую систему управления потоками создания ценности и устранения потерь. Гибкая команда может работать короткими итерациями, но сохранять значительные потери. Например, она регулярно планирует спринты, однако задачи долго ожидают проверки, сотрудники перегружены параллельной работой, а новые функции создаются без анализа их ценности. Lean позволяет увидеть проблемы всего потока. Одновременно Lean-процесс может не использовать фиксированные итерации. Работа может перемещаться непрерывно по мере появления и освобождения мощности. Такой подход характерен для Kanban. В практической деятельности Agile и Lean часто дополняют друг друга. Agile определяет ценности работы в условиях изменений, а Lean помогает оптимизировать поток и устранять потери.

4.1.3 Каркас Scrum

Scrum, в русскоязычном тексте часто называемый Скрамом, представляет собой легковесный каркас организации работы над сложными продуктами. Он основан на эмпирическом управлении, при котором решения принимаются на основе наблюдаемого результата и полученного опыта. Scrum не является подробной технологической инструкцией. Он не определяет язык программирования, структуру архитектуры, метод оценки задач, правила тестирования или инструмент управления проектом. Эти практики выбираются командой с учетом контекста. Актуальная официальная версия руководства Scrum была опубликована в ноябре 2020 года и по состоянию на 2026 год остается текущей официальной редакцией. Руководство определяет Scrum через команду, события, артефакты, обязательства и связывающие их правила. Теоретической основой Scrum является эмпиризм, то есть получение знаний из опыта и принятие решений на основе наблюдений. Эмпиризм поддерживается тремя опорами: прозрачностью, инспекцией и адаптацией. Прозрачность означает, что состояние работы и продукта должно быть понятно участникам. Если требования, качество, прогресс или проблемы скрыты, решения принимаются на основе неправильной информации. Инспекция представляет собой регулярную проверку продукта и способа работы. Она не должна превращаться в тотальный контроль сотрудников. Инспекция направлена на своевременное обнаружение отклонений. Адаптация означает изменение продукта или процесса, если проверка выявила проблему или новую информацию. Инспекция без последующей адаптации не создает пользы. Scrum поддерживается пятью ценностями: обязательностью, сосредоточенностью, открытостью, уважением и смелостью. Эти ценности определяют ожидаемый характер взаимодействия команды. Основ-

ной организационной единицей является Scrum-команда. Она включает владельца продукта, Scrum-мастера и разработчиков. Внутри Scrum-команды отсутствуют отдельные подкоманды и внутренняя иерархия. Команда является межфункциональной и самоуправляемой: она располагает навыками, необходимыми для создания ценного результата, и самостоятельно определяет внутреннюю организацию работы. Официальное руководство указывает, что Scrum-команда обычно состоит из десяти или меньшего числа участников. Термин «разработчики» в Scrum имеет расширенное значение. Он относится ко всем участникам, непосредственно создающим пригодное приращение продукта. В зависимости от предметной области к ним могут относиться программисты, аналитики, тестировщики, проектировщики, специалисты по данным и другие исполнители. Владелец продукта отвечает за максимизацию ценности продукта и результативное управление бэклогом продукта. Он определяет цель продукта, обеспечивает понятность элементов бэклога и упорядочивает их. Владелец продукта не является простым секретарем, записывающим пожелания пользователей. Он должен принимать решения о приоритетах, согласовывать интересы заинтересованных сторон и обеспечивать ориентацию команды на ценность. Ответственность владельца продукта должна быть закреплена за одним человеком, а не комитетом. При этом он может консультироваться с экспертами и представителями пользователей. Scrum-мастер отвечает за правильное понимание и применение Scrum. Он помогает команде развивать самоуправление и межфункциональность, устранять препятствия, улучшать практики и взаимодействовать с организацией. Scrum-мастер не является традиционным начальником команды и не распределяет задачи между разработчиками. Он создает условия, в которых команда может эффективно управлять собственной работой. Scrum организует работу в рамках спринтов. Спринт представляет собой фиксированный период продолжительностью не более одного месяца, в течение которого создается полезное и пригодное приращение продукта. Новый спринт начинается сразу после завершения предыдущего. Внутри спринта проводятся планирование спринта, ежедневный Scrum, обзор спринта и ретроспектива. Короткая продолжительность спринта ограничивает риск. Если решение оказалось ошибочным, команда получает обратную связь раньше. Слишком длинный спринт увеличивает объем работы, выполненной без проверки. Во время спринта не должны вноситься изменения, ставящие под угрозу его цель. Однако содержание работы может уточняться по мере получения новых знаний. Это отличает Scrum от полностью жесткого плана. Планирование спринта определяет, почему предстоящий спринт имеет ценность, что может быть выполнено и каким образом будет создан результат. В ходе планирования формируется цель спринта, выбираются элементы бэклога продукта и составляется план работы. Цель спринта должна объединять работу команды. Если участники выполняют несвязанные задачи без общей цели, спринт превращается в календарный контейнер для набора поручений. Ежедневный Scrum представляет собой короткое ежедневное событие для разработчиков. Его назначение состоит в проверке продвижения к цели спринта и адаптации плана. Ежедневный Scrum не должен превращаться в отчет каждого сотрудника руководителю. Команда координирует собственную работу, выявляет препятствия и принимает решения о следующем шаге. Обзор спринта проводится для анализа результата вместе с заинтересованными сторонами и определения дальнейших изменений. Команда демонстрирует созданное приращение, рассматривает состояние продукта и уточняет будущие приоритеты. Обзор не является только праздничной демонстрацией. Он должен обеспечивать содержательную обратную связь. Если заинтересованные стороны не участвуют или решение уже невозможно изменить, ценность обзора уменьшается. Ретроспектива спринта направлена на повышение качества и эффективности. Команда анализирует людей, взаимодействия, процессы, инструменты и определение готовности, а затем выбирает полезные улучшения. Scrum определяет три основных артефакта: бэклог продукта, бэклог спринта и приращение. Каждому артефакту соответствует обязательство, усиливающее прозрачность. Бэклог продукта является возникающим и упорядоченным перечнем всего, что

необходимо для улучшения продукта. Его обязательством является цель продукта, описывающая будущее состояние, к которому стремится команда. Бэклог называется возникающим, потому что он изменяется по мере получения новых знаний. Он не является раз и навсегда утвержденным списком. Бэклог спринта включает цель спринта, выбранные элементы бэклога продукта и план их реализации. Он создается разработчиками и обновляется в течение спринта. Приращение представляет собой конкретный шаг к цели продукта. Оно должно быть совместимо с предыдущими приращениями, проверено и пригодно для использования. Обязательством приращения является определение готовности. Оно устанавливает формальное описание состояния, при котором работа соответствует требованиям качества и может считаться частью приращения. Официальное руководство связывает бэклог продукта с целью продукта, бэклог спринта — с целью спринта, а приращение — с определением готовности. Определение готовности может включать прохождение автоматических тестов, проверку кода, обновление документации, соответствие требованиям безопасности, развертывание в тестовой среде и отсутствие критических дефектов. Не следует смешивать определение готовности с критериями приемки отдельной функции. Критерии приемки относятся к конкретному требованию, а определение готовности устанавливает общий уровень качества для всех элементов. В Scrum часто используются дополнительные инструменты, не входящие в обязательное ядро каркаса. К ним относятся пользовательские истории, оценка в условных единицах, диаграмма сгорания работы, планирование с карточками, карта историй и методы приоритизации. Например, диаграмма сгорания показывает изменение оставшегося объема работы во времени. Она помогает увидеть, успевает ли команда завершить запланированное. Однако сама диаграмма не гарантирует создание ценности и не должна использоваться как единственный показатель эффективности. Scrum эффективен в условиях сложной работы, когда невозможно заранее определить все решения и требуется регулярная проверка результата. Он обеспечивает ритм, прозрачность и взаимодействие. Ограничением Scrum является риск формального применения. Организация может проводить все события, но сохранять командно-административное управление, передавать задачи крупными партиями и не выпускать продукт месяцами. Другой проблемой является использование спринта как жесткого обязательства выполнить заранее назначенный объем. Цель спринта является обязательством, но отдельные элементы могут уточняться. Попытка любой ценой выполнить первоначальный перечень приводит к снижению качества и скрытию проблем. Scrum не решает автоматически архитектурные и инженерные задачи. Без автоматического тестирования, управления конфигурацией, качественной архитектуры и технического совершенствования команда может быстро накапливать технический долг.

4.1.4 Метод Kanban

Kanban, или Канбан, представляет собой метод управления потоком знаний и услуг, основанный на визуализации работы, ограничении незавершенной работы, управлении потоком и эволюционном совершенствовании существующего процесса. Канбан необходимо отличать от простой доски с карточками. Доска является одним из инструментов визуализации, но полноценный метод включает принципы, правила, показатели, ограничения и циклы обратной связи. Официальное руководство Kanban University выделяет принципы управления изменениями: начинать с существующего способа работы, стремиться к улучшению через эволюционные изменения и поддерживать лидерские действия на всех уровнях. Метод не требует одномоментной полной перестройки организации, а развивается на основе наблюдения за текущим процессом. Первой общей практикой Kanban является визуализация работы. Для этого создается доска, отражающая стадии процесса. Простейшая доска может включать состояния «Запланировано», «Анализ», «Разработка», «Проверка», «Готово». Карточка представляет единицу работы: пользовательскую функцию, дефект, аналитическую задачу, запрос на изме-

нение или техническую работу. Ее движение по доске показывает состояние. Визуализация должна отражать реальный процесс, а не желаемую упрощенную картину. Если задача ожидает архитектурного согласования или развертывания, соответствующая стадия должна быть видна. Иначе время ожидания остается скрытым. На карточке могут указываться тип работы, ответственный, дата начала, срок, приоритет, класс обслуживания, блокировка и связанные зависимости. Второй практикой является ограничение незавершенной работы. Для отдельных стадий устанавливается максимальное количество одновременно находящихся в них элементов. Например, если для стадии разработки установлен предел пять задач, шестая задача не должна начинаться до завершения одной из текущих. Команда сосредотачивается на завершении начатого, а не на постоянном открытии новых задач. Ограничения делают узкие места видимыми. Если очередь перед тестированием постоянно достигает предела, проблема может заключаться в недостаточной автоматизации тестов, неравномерном распределении компетенций или низком качестве разработки. Без ограничения работы доска только отображает перегрузку, но не управляет ею. Третьей практикой является управление потоком. Команда стремится обеспечить предсказуемое и равномерное движение задач. Анализируются задержки, блокировки, возвраты и колебания нагрузки. Поток не означает максимальную загрузку каждого сотрудника. Система, в которой все участники загружены на сто процентов, может работать медленно, поскольку отсутствует резерв для обработки проблем и взаимной помощи. Четвертой практикой является явное определение правил. Участники должны понимать условия перехода карточки между стадиями, правила приоритета, критерии готовности, ограничения и порядок обработки срочных работ. Например, задача может переходить в состояние «Готова к тестированию» только после прохождения модульных тестов, проверки кода и обновления описания программного интерфейса. Неявные правила приводят к конфликтам. Один разработчик считает задачу готовой после написания кода, другой — после развертывания, тестировщик — после устранения всех дефектов. Явная политика устраняет неоднозначность. Пятой практикой является использование циклов обратной связи. В Kanban могут проводиться ежедневные совещания, обзоры поставки, анализ рисков, пополнение очереди, обзоры операций и стратегические обзоры. Разные циклы решают различные задачи. Ежедневная встреча координирует текущий поток, пополнение определяет, какие задачи допускаются к выполнению, а обзор поставки анализирует предсказуемость и качество обслуживания. Шестой практикой является совместное улучшение и экспериментальное развитие. Изменения процесса формулируются как гипотезы и проверяются по показателям. Например, команда предполагает, что автоматизация подготовки тестовой среды уменьшит среднее время прохождения задачи. После внедрения сравниваются показатели до и после изменения. К основным показателям Kanban относятся время выполнения запроса, время цикла, пропускная способность и объем незавершенной работы. Время выполнения запроса измеряет период от появления потребности до предоставления результата. Время цикла обычно измеряет период от начала активной работы до завершения. Границы показателей должны быть явно определены. Пропускная способность показывает количество элементов, завершенных за определенный период. Объем незавершенной работы показывает количество элементов, находящихся в процессе. Важным средством анализа является накопительная диаграмма потока. Она отображает количество элементов в каждом состоянии во времени. Расширение области определенной стадии указывает на накопление очереди. Другим инструментом является диаграмма времени цикла, показывающая длительность выполнения отдельных задач. Она позволяет оценивать вариативность и формировать прогнозы. Kanban стремится к прогнозированию на основе исторических данных. Вместо обещания завершить каждую задачу к точной дате может использоваться вероятностное ожидание, например определенная доля задач данного типа завершается за установленный срок. Для разных типов работ могут применяться классы обслуживания. Срочная работа получает особые правила, но число срочных задач должно быть ограничено. Если каждая задача объяв-

ляется срочной, система теряет управляемость. Kanban хорошо подходит для сопровождения информационных систем, обработки заявок, эксплуатации, поддержки пользователей и других процессов с непрерывным поступлением работы. Он также может использоваться в проектной разработке. В отличие от Scrum, Kanban не требует фиксированных спринтов и определенных ролей. Работа может поставляться по мере готовности. При этом метод не запрещает использование итераций. Scrum и Kanban могут сочетаться. Scrum задает ритм планирования и обзора, а Kanban помогает управлять потоком внутри спринта. Такое сочетание иногда называется Scrumban, или Скрамбаном, однако конкретные правила должны быть определены организацией. Основным риском Kanban является ограничение применения только визуальной доской. Если отсутствуют ограничения незавершенной работы, показатели, явные правила и улучшение процесса, организация не получает основных преимуществ метода.

4.1.5 Сравнение Scrum и Kanban

Scrum и Kanban относятся к гибким способам организации работы, но используют различные механизмы. Scrum организует работу фиксированными периодами. Kanban ориентирован на непрерывный поток. Scrum определяет конкретные ответственности, события и артефакты. Kanban начинается с существующих ролей и процессов. В Scrum объем работы выбирается на спринт и уточняется с учетом цели. В Kanban новая работа принимается при наличии свободной мощности и соблюдении ограничений. Scrum подходит для разработки продукта, когда полезно иметь регулярный ритм планирования, обзора и ретроспективы. Kanban особенно удобен для потока неоднородных запросов, которые трудно заранее сгруппировать в спринты. В Scrum часто применяется скорость команды. В Kanban основное внимание уделяется времени цикла, пропускной способности и объему незавершенной работы. Нельзя утверждать, что один подход всегда лучше другого. Выбор определяется характером деятельности. Команда разработки новой системы может использовать Scrum, а подразделение технической поддержки — Kanban. Одна организация может применять оба подхода.

4.1.6 Типичные ошибки гибкой разработки

Первой ошибкой является представление Agile как отсутствия планирования. Гибкий подход предполагает постоянное планирование на разных уровнях: цели продукта, выпусков, итераций и ежедневной работы. Отличие заключается в возможности корректировать план на основе новой информации. Второй ошибкой является отказ от документации. Гибкость требует создавать документацию в объеме, необходимом для понимания, эксплуатации, безопасности и сопровождения. Особенно важны архитектурные решения, требования к интерфейсам, модели данных и инструкции эксплуатации. Третьей ошибкой является подмена результата количеством выполненных задач. Закрытие карточки не гарантирует получения ценности. Необходимо оценивать использование функции, качество и влияние на цели. Четвертой ошибкой является разделение команды на последовательные мини-подразделения. Аналитик завершает требования, передает разработчику, разработчик — тестировщику. В результате внутри короткого спринта воспроизводится каскадная модель. Пятой ошибкой является отсутствие технического совершенствования. Если команда постоянно добавляет функции, но не улучшает архитектуру, тесты и средства развертывания, скорость разработки постепенно снижается. Шестой ошибкой является использование оценки как средства давления. Условные единицы и скорость предназначены для прогнозирования команды, а не для индивидуального контроля. Седьмой ошибкой является формальная ретроспектива. Если одни и те же проблемы обсуждаются, но не устраняются, доверие к процессу снижается.

4.2 Методология DevOps

DevOps представляет собой культурный, организационный и инженерный подход, объединяющий разработку программного обеспечения и его эксплуатацию в единый поток создания и предоставления ценности. Название образовано от английских слов development и operations, однако в русскоязычной лекции целесообразно понимать DevOps как совместную организацию разработки и эксплуатации. Традиционно разработчики могли отвечать за создание функций, а эксплуатационное подразделение — за стабильность работающей системы. Интересы подразделений вступали в противоречие. Разработчики стремились чаще выпускать изменения, эксплуатация стремилась уменьшить число изменений как источник риска.DevOps исходит из того, что скорость развития и надежность не должны достигаться разными изолированными группами. Команда должна учитывать эксплуатационные характеристики с начала проектирования, а эксплуатационная информация должна использоваться для улучшения продукта. DevOps не является отдельной должностью, конкретным инструментом или обязательной архитектурой. Наличие инженера с названием должности «DevOps» не гарантирует внедрения подхода. Основное значение имеют совместная ответственность, автоматизация, быстрые обратные связи, управляемые изменения и обучение. В DevOps особое значение имеет управление всем потоком поставки. Он начинается с изменения программного кода или конфигурации и завершается безопасным предоставлением результата пользователю и наблюдением за работой. Основой является управление версиями. В системе контроля версий должны храниться программный код, сценарии сборки, конфигурации, тесты, описание инфраструктуры и, по возможности, архитектурная документация. Это обеспечивает историю, проверку изменений и возможность воспроизведения. DORA указывает, что комплексное использование управления версиями является важной способностью непрерывной поставки. Следующей практикой является непрерывная интеграция. Разработчики часто объединяют небольшие изменения в общей основной ветви. Каждое изменение автоматически собирается и проверяется. Назначение непрерывной интеграции заключается не только в запуске автоматических тестов. Она сокращает длительность изоляции изменений и позволяет раньше выявлять конфликты. DORA связывает непрерывную интеграцию с небольшими партиями работы и быстрыми циклами обратной связи. Автоматизированный процесс может включать получение исходного кода, установку зависимостей, компиляцию, модульные тесты, статический анализ, проверку безопасности, сборку пакета и публикацию артефакта. Если сборка или обязательная проверка завершается ошибкой, исправление должно иметь высокий приоритет. Длительное существование неработающей основной ветви лишает команду надежной интеграционной основы. Непрерывная поставка означает способность быстро, безопасно и устойчиво выпускать изменения по требованию. Программный продукт поддерживается в состоянии, пригодном для развертывания. DORA отличает непрерывную поставку от непрерывного развертывания: при поставке выпуск возможен в любой момент, а при непрерывном развертывании каждое успешно прошедшее изменение автоматически передается в промышленную среду. Непрерывное развертывание подходит не для каждой системы. Мобильное приложение требует публикации через внешнюю площадку, а критическая система может требовать официального разрешения. Однако даже при ручном решении о выпуске сборка, тестирование и подготовка должны быть автоматизированы. Последовательность автоматизированных действий называется конвейером поставки. Он может включать стадии сборки, тестирования, проверки безопасности, упаковки, развертывания и контроля после выпуска. Описание конвейера целесообразно хранить в виде кода. Jenkins, например, поддерживает определение конвейера в файле Jenkinsfile, который хранится вместе с проектом и проходит управление версиями. Официальная документация определяет такой конвейер как автоматизированное

представление процесса движения программного обеспечения от системы контроля версий до пользователей. Системы GitHub Actions также позволяют автоматизировать сборку, тестирование и развертывание на основе событий в репозитории. Ключевой практикой DevOps является инфраструктура как код. Серверы, сети, хранилища, политики и другие ресурсы описываются декларативными конфигурациями, которые можно хранить в системе контроля версий, проверять и повторно применять. Terraform позволяет описывать облачные и локальные ресурсы в конфигурационных файлах и управлять их жизненным циклом через последовательность записи, предварительного плана и применения изменений. Инфраструктура как код снижает зависимость от ручной настройки. Если тестовая среда создается вручную, она может незаметно отличаться от промышленной. Автоматизированное описание обеспечивает повторяемость. Управление конфигурацией автоматизирует настройку операционных систем, программ, пользователей и сервисов. Ansible использует человекочитаемые сценарии, называемые плейбуками, и стремится приводить управляемую систему к описанному состоянию. Повторное выполнение не должно изменять уже правильно настроенную систему. Контейнеризация позволяет упаковать приложение вместе с необходимыми зависимостями в переносимый образ. Контейнеры запускаются как изолированные процессы и помогают обеспечить единообразие среды. Контейнер не решает автоматически проблемы архитектуры. Если приложение плохо разделено, не имеет мониторинга или хранит состояние неправильно, упаковка в контейнер не устранит недостатки. Для управления большим числом контейнеров применяется оркестрация. Kubernetes является открытой платформой управления контейнеризированными рабочими нагрузками и поддерживает декларативную конфигурацию, автоматизированное развертывание, масштабирование и управление. Оркестрация оправданна при соответствующем масштабе и сложности. Для небольшой системы Kubernetes может создать больше эксплуатационной нагрузки, чем пользы. DevOps требует автоматизированного тестирования. Проверки должны выполняться на разных уровнях: модульном, интеграционном, системном, пользовательском, нагрузочном и безопасностном. Автоматизация не отменяет исследовательское тестирование. Она берет на себя повторяющиеся проверки и освобождает специалистов для анализа сложных сценариев. Важной практикой является статический анализ программного кода. Он позволяет обнаруживать потенциальные дефекты, нарушения правил, уязвимости и проблемы сопровождаемости без выполнения программы. SonarQube, например, применяет профили правил и пороги качества, определяющие возможность принятия изменения. Официальная документация связывает анализ с защищенностью, надежностью и сопровождаемостью программного обеспечения. DevOps охватывает наблюдаемость системы. Она включает журналы событий, показатели, распределенную трассировку и профилирование. Мониторинг показывает известные показатели и состояния, например загрузку процессора, количество ошибок или время отклика. Наблюдаемость позволяет исследовать внутреннее состояние системы по внешним данным и отвечать на заранее не предусмотренные вопросы. Логи должны содержать достаточный контекст, но не раскрывать пароли, персональные данные и другие секреты. Метрики используются для анализа тенденций и уведомлений. Трассировка показывает прохождение запроса через распределенные компоненты. DevOps использует управление инцидентами. При отказе команда должна обнаружить проблему, ограничить влияние, восстановить сервис, установить причины и реализовать улучшения. Анализ инцидента должен быть направлен не на поиск виновного, а на понимание системных причин. Ошибка человека часто становится катастрофической только при отсутствии защитных механизмов, проверок и возможности безопасного отката. Для уменьшения риска применяются стратегии постепенного выпуска. Сине-зеленое развертывание использует две среды, между которыми переключается трафик. Канареечный выпуск сначала предоставляет новую версию небольшой части пользователей. Функциональные переключатели позволяют включать и выключать функции независимо от развертывания кода. Автоматизация выпуска должна

сопровождаться возможностью отката или быстрого исправления. При изменении структуры базы данных откат может быть сложным, поэтому миграции проектируются с учетом совместимости версий.

4.2.1 Метрики DevOps

Для оценки способности поставлять изменения применяются показатели DORA. В актуальной модели 2026 года используются пять показателей: время прохождения изменения от фиксации в системе контроля версий до промышленного развертывания; частота развертываний; время восстановления после неудачного развертывания; доля изменений, потребовавших немедленного вмешательства; доля незапланированных развертываний, выполняемых из-за промышленного инцидента. Первые три показателя характеризуют пропускную способность изменений, последние два — нестабильность. Метрики не должны использоваться для ранжирования отдельных сотрудников. Они описывают способность команды и системы поставки. Попытка механически увеличить частоту развертывания может привести к искусственному дроблению изменений без повышения ценности. Показатели необходимо интерпретировать в контексте конкретного приложения. Сравнение мобильного приложения, банковской системы и небольшого веб-сервиса без учета различий не дает объективного результата. DORA рекомендует применять показатели на уровне отдельного приложения или сервиса и использовать их для постоянного улучшения.

4.2.2 Расширение DevSecOps

DevSecOps представляет собой развитие DevOps, при котором безопасность интегрируется во весь поток разработки и эксплуатации, а не проверяется только перед выпуском. Традиционная модель могла предусматривать передачу почти готового продукта службе безопасности. Если обнаруживалась архитектурная уязвимость, исправление требовало значительной переработки. DevSecOps применяет принцип раннего включения безопасности. Моделирование угроз выполняется при проектировании, зависимости проверяются при сборке, программный код анализируется автоматически, контейнерные образы сканируются, а конфигурации инфраструктуры проверяются до развертывания. В конвейер могут включаться статический анализ безопасности, динамическое тестирование, анализ состава программного обеспечения, проверка секретов, сканирование инфраструктурных конфигураций и формирование перечня компонентов программного продукта. NIST рассматривает DevSecOps через интеграцию мер безопасности цепочки поставки программного обеспечения в процессы непрерывной интеграции и развертывания, охватывающие сборку, тестирование, упаковку и выпуск. Анализ состава программного обеспечения выявляет сторонние библиотеки, их версии, лицензии и известные уязвимости. Современная система может содержать значительно больше внешних компонентов, чем собственного кода, поэтому управление зависимостями является важной частью безопасности. Перечень программных компонентов, или SBOM, фиксирует состав программного продукта. Он помогает установить, какие системы затронуты обнаруженной уязвимостью библиотеки. Необходимо защищать сам конвейер поставки. Если злоумышленник получает возможность изменить сценарий сборки, подменить зависимость или опубликовать артефакт, проверка исходного кода может оказаться недостаточной. DevSecOps не должен превращаться в добавление большого количества блокирующих проверок без обратной связи. Проверки должны быть быстрыми, понятными и по возможности выполняться на рабочем месте разработчика до передачи изменения.

4.2.3 GitOps

GitOps является подходом к управлению системами, при котором желаемое состояние приложения и инфраструктуры описывается декларативно, хранится в репозитории Git, а автоматизированные контроллеры приводят фактическую среду в соответствие с этим состоянием. OpenGitOps развивает нейтральные к поставщикам принципы и открытые практики применения GitOps. В традиционном конвейере система сборки может напрямую выполнять команды в промышленной среде. В GitOps изменение сначала фиксируется в репозитории конфигурации, проходит проверку и согласование, после чего контроллер обнаруживает изменение и применяет его. Репозиторий становится источником желаемого состояния. История Git позволяет увидеть, кто и почему внес изменение, провести проверку и вернуться к предыдущей конфигурации. GitOps особенно тесно связан с декларативными платформами, например Kubernetes, но его принципы могут применяться шире. Основными преимуществами являются прослеживаемость, воспроизводимость, разделение полномочий и автоматическое устранение расхождения между описанным и фактическим состоянием. Риском является неправильное управление секретами. Пароли и ключи нельзя хранить в открытом виде в репозитории. Необходимы специализированные механизмы шифрования и управления секретами.

4.2.4 DataOps

DataOps переносит идеи гибкой разработки, автоматизации и DevOps в область управления данными. Его назначение состоит в обеспечении надежного и быстрого движения данных от источников к пользователям, аналитическим системам и моделям. В традиционном проекте аналитическое хранилище может обновляться редко, а изменение структуры источника обнаруживаться после формирования ошибочного отчета. DataOps предусматривает управление версиями схем, автоматизированное тестирование данных, наблюдение за конвейерами и совместную работу инженеров данных, аналитиков и владельцев предметной области. Проверки данных могут включать контроль полноты, уникальности, допустимых диапазонов, ссылочной целостности, свежести и распределения значений. Конвейер данных должен быть воспроизводимым. Необходимо понимать происхождение показателя: из какого источника он получен, какие преобразования выполнены и какая версия правил использовалась. DataOps не является простым использованием инструмента загрузки данных. Он требует управления качеством, ответственностью, обратной связью и жизненным циклом информационного продукта.

4.2.5 MLOps

MLOps представляет собой применение принципов DevOps к системам машинного обучения. Такие системы включают не только программный код, но и данные, признаки, обученную модель, параметры, вычислительную среду и процесс обучения. Обычный программный продукт изменяет поведение преимущественно после изменения кода или конфигурации. Модель машинного обучения может ухудшаться из-за изменения реальных данных даже без изменения программы. MLOps включает управление версиями данных и моделей, автоматизированное обучение, проверку качества, развертывание, мониторинг и повторное обучение. Google Cloud определяет MLOps как культуру и инженерную практику, объединяющую разработку и эксплуатацию систем машинного обучения, с автоматизацией интеграции, тестирования, выпуска, развертывания, инфраструктуры и непрерывного обучения. Конвейер машинного обучения может включать получение данных, проверку качества, подготовку признаков, обучение, оценку, сравнение с действующей моделью, регистрацию, развертывание и мониторинг. Необходимо контролировать дрейф данных, когда распределение входных сведений изменяется, и дрейф концепции, когда меняется зависимость между входными данными и целевым результатом. Например, модель прогнозирования спроса, обученная на исторических данных, может ухудшиться после изменения поведения покупателей или появления нового

канала продаж. Нельзя автоматически развертывать модель только потому, что процесс обучения завершился. Она должна пройти проверку точности, устойчивости, справедливости, безопасности и соответствия бизнес-ограничениям.

4.2.6 AIOps

AIOps представляет собой применение аналитики и методов искусственного интеллекта к эксплуатации информационных систем. Он используется для обработки большого объема событий, обнаружения аномалий, корреляции сигналов, прогнозирования отказов и поддержки анализа причин. Например, распределенная система может формировать миллионы записей журналов. AIOps помогает объединить взаимосвязанные события и выделить вероятный инцидент. AIOps не заменяет качественное журналирование, мониторинг и архитектуру. Алгоритм не сможет достоверно анализировать систему, если исходные эксплуатационные данные неполны или противоречивы. Автоматическое реагирование должно применяться осторожно. Ошибочная классификация может привести к ненужному перезапуску сервисов или масштабированию. Для критических действий могут требоваться подтверждение специалиста и ограничение полномочий.

4.2.7 Platform Engineering

Платформенная инженерия представляет собой создание внутренней технологической платформы, предоставляющей командам разработки стандартизированные и самообслуживаемые возможности для сборки, тестирования, развертывания и наблюдения. При масштабировании DevOps каждая команда может самостоятельно собирать сложный набор инструментов. Это приводит к дублированию, различиям в безопасности и высокой когнитивной нагрузке. Платформенная команда создает типовые пути: шаблон нового сервиса, стандартный конвейер, автоматизированное окружение, готовый мониторинг, управление секретами и каталог внутренних сервисов. Платформа должна рассматриваться как продукт. Ее пользователями являются команды разработки, а качество оценивается удобством, надежностью и способностью ускорять поставку. Платформа не должна полностью лишать команды выбора. Целесообразно предоставлять безопасный стандартный путь и возможность обоснованного отклонения для особых случаев.

4.2.8 FinOps

FinOps представляет собой практику совместного управления экономической эффективностью облачных и цифровых ресурсов. Она объединяет технических специалистов, финансовые подразделения и владельцев продукта. В традиционной инфраструктуре затраты часто определялись заранее приобретенным оборудованием. В облачной среде ресурсы создаются и масштабируются программно, поэтому расходы изменяются динамически. FinOps предполагает прозрачность стоимости, распределение расходов по продуктам, планирование, контроль неиспользуемых ресурсов и учет экономических последствий архитектурных решений. Например, использование высокопроизводительного хранилища может улучшить скорость, но значительно увеличить стоимость. Архитектор должен понимать не только технические, но и экономические последствия. Оптимизация стоимости не означает выбор самого дешевого решения. Недостаточная надежность или производительность может привести к большим потерям. Необходимо искать баланс между ценностью, качеством и стоимостью.

4.2.9 Расширения DevOps как система взаимосвязанных практик

Обозначения DevSecOps, GitOps, DataOps, MLOps, AIOps и FinOps отражают расширение принципов совместной ответственности и автоматизации на разные области. Они не должны превращаться в изолированные подразделения с новыми барьерами. Если организа-

ция создает отдельную команду DevSecOps, которая только принимает запросы на проверку безопасности, она воспроизводит прежний барьер под новым названием. Цель заключается в интеграции компетенций и автоматизированных механизмов в общий поток. Аналогично MLOps не должен отделять специалистов по моделям от инженеров данных и эксплуатации. Результат зависит от совместной ответственности за всю систему.

4.3 Инструменты автоматизации проектирования информационных систем

Автоматизация проектирования информационных систем включает применение программных средств для моделирования, управления требованиями, разработки архитектуры, генерации программного кода, проектирования данных, тестирования, развертывания и документирования. Традиционно такие средства назывались CASE-средствами, то есть средствами автоматизированной поддержки программной инженерии. CASE-система может поддерживать одну или несколько стадий жизненного цикла. Верхнеуровневые CASE-средства ориентированы на обследование, анализ требований, моделирование процессов и концептуальное проектирование. Нижнеуровневые CASE-средства поддерживают детальное проектирование, программирование, тестирование и сопровождение. Интегрированные CASE-средства объединяют несколько стадий и используют общий репозиторий моделей. Современная граница между CASE-средствами, средами разработки, системами управления требованиями и DevOps-платформами становится условной. Один продукт может одновременно хранить требования, модели, исходный код, тесты и конвейеры.

4.3.1 Средства управления требованиями

Инструменты управления требованиями обеспечивают создание, классификацию, согласование, версионирование и прослеживаемость требований. Требование должно иметь идентификатор, состояние, приоритет, источник, критерии приемки и связи. Система позволяет установить, какое требование реализуется определенным компонентом и каким тестом проверяется. Для гибкой разработки требования часто хранятся в системах управления бэклогом и задачами. Примерами являются Jira, Azure DevOps, GitLab, GitHub Projects и YouTrack. Название инструмента не определяет качество требований. Карточка с формулировкой «сделать отчет удобным» остается непроверяемой независимо от используемой системы. Инструмент должен поддерживать иерархию: цели, инициативы, крупные функции, пользовательские истории, задачи и дефекты. Полезны связи между требованиями, история изменений, права доступа и отчеты. В критических проектах применяются специализированные средства управления требованиями, поддерживающие базовые линии, формальное согласование и матрицы прослеживаемости.

4.3.2 Инструменты моделирования бизнес-процессов

Для описания деятельности организации применяются средства моделирования бизнес-процессов. Одной из распространенных нотаций является BPMN, предоставляющая графические обозначения событий, действий, потоков, участников и сообщений. Спецификация BPMN предназначена для описания бизнес-процессов и не зависит от конкретной среды реализации. Модель процесса помогает определить последовательность действий, ответственность, условия ветвления, исключения и взаимодействие организаций. Например, процесс обработки заявки может включать регистрацию, автоматическую проверку, согласование руководителем, возврат на исправление и окончательное утверждение. Инструменты могут проверять синтаксическую корректность модели, выполнять имитацию, рассчитывать показатели и генерировать документацию. Имитация позволяет оценить очереди, загрузку ресурсов и длительность процесса до его автоматизации. Однако точность результата зависит от качества исходных данных.

4.3.3 Инструменты системного и программного моделирования

Для моделирования программных систем применяется UML, или унифицированный язык моделирования. Он включает диаграммы классов, компонентов, последовательностей,

состояний, деятельности, вариантов использования и развертывания. Официальная спецификация UML публикуется Object Management Group. Диаграмма вариантов использования показывает цели внешних участников. Диаграмма классов отражает структуру объектов и отношений. Диаграмма последовательности показывает обмен сообщениями во времени. Диаграмма компонентов описывает крупные программные элементы. Диаграмма развертывания связывает программные компоненты с техническими узлами. UML не требует создавать все возможные диаграммы. Следует моделировать только те аспекты, которые помогают принимать решения и передавать знания. Для систем, включающих программные и физические компоненты, применяется SysML, или язык системного моделирования. Он поддерживает требования, структуру, поведение, параметры и связи элементов системы. Для корпоративной архитектуры используется ArchiMate. Этот язык позволяет описывать и анализировать связи между бизнес-процессами, организацией, информационными системами и технологической инфраструктурой. Официальная спецификация позиционирует ArchiMate как открытый независимый язык моделирования корпоративной архитектуры. В корпоративной модели можно показать, какой бизнес-процесс поддерживается каким приложением, какие данные используются и на какой инфраструктуре размещаются компоненты. К инструментам моделирования относятся Enterprise Architect, Visual Paradigm, Cameo Systems Modeler, Archi и другие средства. Enterprise Architect поддерживает моделирование требований, UML, BPMN, проектирование кода, прямую и обратную генерацию, общий репозиторий и прослеживаемость. Преимуществом репозиторного средства является возможность хранить не отдельные несвязанные рисунки, а единую модель. Если один компонент используется на нескольких диаграммах, он остается одним объектом с общими свойствами.

4.3.4 Модель C4

Для визуализации программной архитектуры применяется модель C4. Она использует последовательное увеличение детализации: контекст системы, контейнеры, компоненты и код. Дополнительно могут применяться диаграммы системного ландшафта, динамики и развертывания. Диаграмма контекста показывает пользователей, систему и внешнее окружение. Она отвечает на вопрос о границах решения. Диаграмма контейнеров показывает крупные исполняемые приложения и хранилища данных. В терминологии C4 контейнером является приложение или хранилище, а не обязательно контейнер Docker. Диаграмма компонентов раскрывает внутреннее устройство отдельного контейнера. Диаграмма кода показывает классы, интерфейсы или иные элементы реализации и обычно создается только для сложных частей. Модель C4 не привязана к конкретному графическому обозначению или инструменту. Она требует ясного уровня абстракции и понятных подписей.

4.3.5 Документация как код

Современный подход «документация как код» предполагает хранение архитектурных описаний в текстовом формате вместе с программным кодом. Для диаграмм могут применяться PlantUML, Mermaid, Structurizr DSL и Graphviz. Текстовое описание удобно хранить в Git, проверять при рецензировании и автоматически преобразовывать в изображения или веб-документацию. Преимуществом является синхронизация с разработкой. Изменение архитектуры может включаться в тот же запрос на объединение, что и программный код. Однако текстовая модель не всегда удобна для сложного корпоративного моделирования. Репозиторные CASE-средства предоставляют более развитую семантику, анализ и управление связями. Выбор зависит от масштаба, аудитории и необходимости формального моделирования.

4.3.6 Инструменты проектирования данных

Инструменты моделирования данных поддерживают концептуальные, логические и физические модели. Они позволяют определять сущности, атрибуты, ключи, связи, ограничения и индексы. К таким средствам относятся специализированные моделировщики, а также функции в системах управления базами данных и интегрированных средах. Прямое проек-

тирование преобразует модель в сценарии создания базы. Обратное проектирование строит модель на основе существующей схемы. Автоматическая генерация не освобождает архитектора данных от принятия решений. Необходимо учитывать объем, характер запросов, целостность, безопасность, архивирование и масштабирование.

4.3.7 Инструменты проектирования программных интерфейсов

Для описания интерфейсов HTTP применяется OpenAPI. Спецификация предоставляет независимый от языка программирования формат, позволяющий людям и программным средствам понимать возможности сервиса без доступа к его исходному коду. По описанию OpenAPI можно автоматически формировать документацию, клиентские библиотеки, серверные заготовки, проверочные тесты и имитаторы сервиса. Подход «сначала контракт» предполагает согласование интерфейса до реализации. Разработчики клиента и сервера могут работать параллельно, используя утвержденное описание. Подход «сначала код» генерирует спецификацию из программных аннотаций. Он быстрее на начальном этапе, но существует риск, что описание будет отражать техническую реализацию, а не согласованный контракт. Кроме структуры запросов и ответов необходимо документировать ошибки, аутентификацию, ограничения частоты, идемпотентность, версии и правила совместимости.

4.3.8 Интегрированные среды разработки

Интегрированная среда разработки объединяет редактор, навигацию по коду, отладчик, средства сборки, рефакторинг, тестирование и анализ. К таким средам относятся IntelliJ IDEA, Visual Studio, Eclipse и другие продукты. Рефакторинг позволяет изменять внутреннюю структуру программы без изменения наблюдаемого поведения. Среда может безопасно переименовывать элементы, выделять методы, перемещать классы и обновлять ссылки. Средства статического анализа обнаруживают ошибки до выполнения. Интеграция с системой контроля версий позволяет сравнивать и рецензировать изменения. Инструмент может автоматически генерировать часть кода, но не способен самостоятельно определить правильную архитектуру. Автоматизация повышает скорость как хороших, так и плохих решений.

4.3.9 Средства управления версиями и совместной разработки

Git является основой совместной работы над программным кодом и конфигурациями. Репозиторий хранит историю, ветви и связи изменений. Платформы GitHub, GitLab, Bitbucket и Azure DevOps добавляют запросы на объединение, проверку кода, задачи, конвейеры и хранение артефактов. Проверка кода помогает выявлять дефекты, распространять знания и поддерживать архитектурные правила. Она не должна ограничиваться проверкой стиля. Необходимо анализировать корректность, безопасность, тестируемость, влияние на архитектуру и соответствие требованиям. Большие запросы на объединение сложно проверять. Небольшие изменения сокращают время обратной связи и вероятность скрытых ошибок.

4.3.10 Инструменты сборки и управления зависимостями

Системы сборки автоматизируют компиляцию, тестирование, упаковку и публикацию. Примерами являются Maven, Gradle, MSBuild, npm и другие средства. Управление зависимостями определяет версии внешних библиотек и источники получения. Для воспроизводимости необходимо фиксировать версии и защищать репозитории пакетов. Использование новейшей версии каждой библиотеки не всегда оправданно. Обновление должно сопровождаться анализом совместимости, безопасности и лицензий.

4.3.11 Средства автоматизированного тестирования

Инструменты тестирования поддерживают модульные, интеграционные, пользовательские, нагрузочные и безопасностные проверки. Модульный тест проверяет отдельный компонент. Интеграционный тест проверяет взаимодействие с базой данных, очередью или внешним сервисом. Пользовательский тест воспроизводит действия через интерфейс. Контрактный тест проверяет совместимость сервисов. Для веб-интерфейсов используются Selenium, Playwright, Cypress и другие средства. Для нагрузочного тестирования применяются JMeter, k6 и Gatling.

Автоматический тест должен быть надежным, быстрым и понятным. Нестабильные тесты, случайно завершающиеся ошибкой, снижают доверие и начинают игнорироваться. Тестовая пирамида предполагает большое число быстрых нижнеуровневых тестов и меньшее число дорогих сквозных проверок. Конкретное соотношение зависит от архитектуры.

4.3.12 Средства непрерывной интеграции и поставки

К платформам автоматизации относятся Jenkins, GitHub Actions, GitLab CI/CD, Azure Pipelines, TeamCity и другие средства. Они запускают действия по событию: фиксации изменения, созданию запроса на объединение, выпуску версии или расписанию. Конвейер должен быть воспроизводимым, наблюдаемым и защищенным. Ручные скрытые действия создают зависимость от конкретного сотрудника. Секреты не должны находиться в открытом тексте сценария. Необходимо использовать хранилища секретов и ограниченные учетные данные. Артефакт, прошедший тестирование, должен продвигаться между средами без повторной непроверенной сборки. Иначе промышленная версия может отличаться от протестированной.

4.3.13 Контейнеризация и оркестрация

Docker поддерживает создание и запуск контейнерных образов. Образ должен быть воспроизводимым, минимальным и не содержать секретов. Многоэтапная сборка позволяет отделить инструменты компиляции от конечного образа. Фиксация базовой версии снижает риск непредсказуемого изменения. Kubernetes автоматизирует развертывание, масштабирование и восстановление контейнерных приложений. Он использует декларативные объекты, описывающие желаемое состояние. Вместе с Kubernetes часто применяются Helm для шаблонизации конфигураций, Argo CD или Flux для GitOps, а также системы сервисной сетки для управления взаимодействием. Сложность оркестрации должна соответствовать реальной задаче. Для небольшого монолитного приложения может быть достаточно виртуальной машины или управляемой платформы.

4.3.14 Инструменты наблюдаемости

К системам сбора метрик относятся Prometheus и облачные платформы мониторинга. Для визуализации применяется Grafana. Для журналов используются Elasticsearch, OpenSearch, Loki и другие хранилища. Для распределенной трассировки применяется OpenTelemetry и совместимые системы. Инструмент наблюдаемости должен быть связан с целями системы. Сбор огромного количества показателей без ясного применения увеличивает стоимость и усложняет анализ. Полезно определять показатели уровня сервиса и цели уровня сервиса. Например, доля успешных запросов или допустимое время отклика. Уведомление должно сигнализировать о проблеме, требующей действия. Если система генерирует сотни несущественных предупреждений, сотрудники перестают на них реагировать.

4.3.15 Инструменты управления архитектурными решениями

Для фиксации архитектурных решений применяются записи архитектурных решений. Они содержат контекст, рассматриваемые варианты, выбранное решение и последствия. Такие записи можно хранить в репозитории в текстовом формате. Каждое решение получает номер и состояние: предложено, принято, заменено или отменено. Например, решение о выборе асинхронного обмена должно объяснять, какие требования его вызвали, какие альтернативы анализировались и какие новые сложности возникнут. Инструментальная автоматизация может проверять архитектурные ограничения. Тесты могут запрещать определенные зависимости между модулями, контролировать направление слоев и проверять использование интерфейсов. Такие проверки называются архитектурными функциями пригодности. Они превращают часть архитектурных правил из документации в автоматически проверяемые условия.

4.3.16 Преимущества и ограничения автоматизации проектирования

Автоматизация повышает скорость, повторяемость, прослеживаемость и качество. Модель можно автоматически проверить, документацию — сгенерировать, инфраструктуру — воспроизвести, а тесты — запускать после каждого изменения. Однако инструмент не

заменяет методологию и профессиональное мышление. Неправильное требование, внесенное в специализированную систему, остается неправильным. Неподходящая архитектура, красиво изображенная в UML, остается неподходящей. Избыточное количество инструментов может увеличить сложность. Информация дублируется между системой требований, доской задач, репозиторием, средой моделирования и документацией. Необходимо определить источник истины для каждого вида данных. Интеграция инструментов должна поддерживать поток работы. Например, требование связывается с изменением кода, изменение — со сборкой, сборка — с тестами и выпуском. При выборе инструмента учитываются поддерживаемые процессы и нотации, интеграции, масштаб, права доступа, стоимость, безопасность, возможность экспорта данных, обучение и долгосрочная поддержка. Особенно важно избегать полной зависимости от закрытого формата. Организация должна иметь возможность получить собственные требования, модели, документы и историю изменений.

4.4 Понятие архитектуры и технологического стека

Архитектура информационной системы определяет фундаментальную организацию системы, ее основные элементы, обязанности, связи, принципы и значимые решения. Технологический стек представляет собой совокупность языков программирования, платформ, библиотек, систем управления базами данных, протоколов, инфраструктурных и эксплуатационных средств, используемых для реализации архитектуры. Архитектура и стек тесно связаны, но не тождественны. Архитектура отвечает на вопрос, как организована система, а технологический стек — какими средствами она реализована. Например, система может иметь слоистую архитектуру независимо от того, реализована ли она на Java,.NET или Python. Микросервисная архитектура может использовать разные языки и базы данных. Технологии не должны определять архитектуру без учета требований. Выбор Kubernetes не является основанием автоматически делить приложение на десятки сервисов. Сначала определяются задачи и архитектурные границы, затем выбираются подходящие средства. Архитектурный стиль описывает семейство систем с общими характеристиками. К распространенным стилям относятся многоуровневая архитектура, модульный монолит, микросервисы, событийно-ориентированная архитектура, клиент-серверная система, обработка через очередь и работника, бессерверная архитектура и архитектура обработки больших данных. Microsoft подчеркивает, что архитектурный стиль не требует определенной технологии, хотя некоторые технологии лучше соответствуют конкретному стилю.

4.4.1 Уровни технологического стека

В технологическом стеке можно выделить несколько условных уровней. Клиентский уровень включает веб-интерфейс, мобильное приложение, настольную программу или специализированное устройство. Здесь выбираются язык, пользовательская библиотека, средства управления состоянием и взаимодействия с сервером. Для веб-клиента могут использоваться HTML, CSS, JavaScript или TypeScript и библиотеки React, Angular или Vue. Выбор должен учитывать сложность интерфейса, компетенции команды, доступность компонентов и срок поддержки. Не каждое приложение требует тяжелой клиентской платформы. Для простой административной формы может быть достаточно серверной генерации страниц. Серверный уровень реализует прикладную логику, обработку запросов и правила предметной области. Возможны платформы Java и Spring,.NET, Node. js, Python, Go и другие технологии. Выбор языка определяется не только производительностью. Необходимо учитывать экосистему, библиотеки, специалистов, безопасность, тестируемость и сопровождение. Интеграционный уровень включает программные интерфейсы, брокеры сообщений, шлюзы и средства преобразования данных. Синхронное взаимодействие удобно, когда результат необходим немедленно. Асинхронное взаимодействие снижает временную связанность, но требует обработки повторов, порядка сообщений и временной несогласованности. Уровень данных включает операционные базы, кэш, поисковые индексы, файловые хранилища, аналитические платформы и средства потоковой обработки. Реляционная база подходит для структурированных данных и транзакций. Документное хранилище удобно для гибких структур. Графовая база используется при сложных отношениях. Поисковый индекс оптимизирован для текстового поиска. Не следует выбирать нереляционную базу только из-за ее популярности. Для финансовой операции строгая транзакционность может быть важнее горизонтального масштабирования. Инфраструктурный уровень включает физические серверы, виртуальные машины, контейнеры, облачные сервисы, сети и системы хранения. Модель развертывания может быть локальной, облачной или гибридной. Выбор определяется законодательством, безопасностью, стоимостью, доступностью и компетенциями. Уровень поставки включает систему контроля версий, сборку, тестирование, хранилище артефактов, развертывание и управление инфра-

структурой. Уровень эксплуатации включает мониторинг, журналы, трассировку, управление инцидентами, резервное копирование и восстановление. Уровень безопасности проходит через весь стек. Он включает идентификацию, управление доступом, шифрование, секреты, анализ зависимостей, защиту сети и аудит. Безопасность нельзя реализовать одним отдельным продуктом. Она должна быть свойством архитектуры и процесса.

4.4.2 Критерии выбора технологического стека

Главным основанием являются требования. Стек должен обеспечивать функции, производительность, надежность, безопасность, масштабируемость и условия эксплуатации. Если система должна работать без подключения к сети на слабом мобильном устройстве, это существенно влияет на клиентскую архитектуру и технологии хранения. Если данные должны размещаться в изолированной инфраструктуре, ряд облачных сервисов исключается. Вторым критерием является зрелость технологии. Зрелая технология имеет устойчивую документацию, инструменты, сообщество, специалистов и известные ограничения. Новая технология может предоставлять полезные возможности, но создавать риск отсутствия опыта и долгосрочной поддержки. Ее целесообразно проверять прототипом. Третьим критерием является компетенция команды. Технологически идеальное решение может быть неудачным, если организация не способна его разработать и сопровождать. Однако нельзя всегда выбирать только уже знакомые технологии. Если существующий стек объективно ограничивает развитие, необходимо планировать обучение и постепенный переход. Четвертым критерием является экосистема. Оцениваются библиотеки, средства тестирования, мониторинга, безопасности, интеграции и документации. Пятым критерием является совместимость с существующими системами. Новая система редко создается изолированно. Необходимо учитывать протоколы, форматы, каталоги пользователей и инфраструктуру. Шестым критерием является стоимость владения. Она включает лицензии, оборудование, облачные ресурсы, разработку, обучение, эксплуатацию и обновление. Бесплатный продукт может иметь высокую стоимость сопровождения. Коммерческий продукт может уменьшить трудозатраты, но создать лицензионную зависимость. Седьмым критерием является лицензирование. Необходимо понимать условия использования, распространения, модификации и облачного предоставления. Восьмым критерием является безопасность и поддержка. Важно оценивать частоту обновлений, процесс устранения уязвимостей и срок поддержки версий. Девятым критерием является переносимость и зависимость от поставщика. Использование управляемого сервиса сокращает эксплуатационную нагрузку, но может затруднить перенос. Зависимость не всегда является недопустимой. Она может быть осознанным компромиссом ради скорости и качества. Необходимо понимать стоимость выхода. Десятым критерием является наблюдаемость и управляемость. Технология должна позволять контролировать производительность, ошибки и использование ресурсов. Одиннадцатым критерием является простота. Следует выбирать минимальный стек, достаточный для требований. Каждый дополнительный язык, хранилище и платформа увеличивают нагрузку на разработку, безопасность и сопровождение.

4.4.3 Пример технологического стека

Для корпоративного веб-приложения может быть выбрана архитектура модульного монолита. Пользовательский интерфейс создается на TypeScript и React, серверная часть — на Java и Spring Boot, данные хранятся в PostgreSQL, для кэширования используется Redis, а интеграция с внешними системами выполняется через HTTP-интерфейсы и брокер сообщений. Приложение упаковывается в контейнерный образ. Для небольшой нагрузки оно может развертываться на нескольких виртуальных машинах без сложной оркестрации. Git используется для управления версиями, GitLab CI/CD — для сборки и тестирования, Terraform — для инфраструктуры, Ansible — для конфигурации, Prometheus и Grafana — для мониторинга. Этот пример не является универсальным рекомендуемым стеком. В другом проекте. NET и SQL Server могут лучше соответствовать существующей инфраструктуре. Для небольшой внутрен-

ней системы часть компонентов будет лишней. Важно, чтобы каждый элемент решал конкретную задачу. Если Redis не требуется для достижения показателей производительности, его добавление только усложнит систему.

4.4.4 Архитектура в гибкой разработке

Распространено ошибочное мнение, что Agile исключает предварительное архитектурное проектирование. Гибкая разработка отказывается не от архитектуры, а от попытки заранее детально спроектировать всю систему без обратной связи. На начальном этапе необходимо определить минимально достаточную архитектуру: границы системы, основные компоненты, критические данные, интеграции, безопасность и решения по самым опасным рискам. Некоторые решения должны приниматься заранее. Например, требования к размещению персональных данных, отказоустойчивости и взаимодействию с внешней системой могут определять базовую структуру. Другие решения можно отложить до появления информации. Преждевременный выбор технологии для гипотетической нагрузки создает лишнюю сложность. В гибкой архитектуре применяется понятие архитектурной основы, иногда называемой архитектурным заделом. Она включает технические возможности, необходимые для будущего развития. Например, если в следующих итерациях планируется подключение нескольких каналов уведомлений, архитектура должна предусмотреть расширяемый интерфейс, но не обязательно сразу реализовывать все каналы. Для проверки архитектурных гипотез применяются технические исследования и прототипы. Критический риск должен проверяться как можно раньше. Архитектурные решения фиксируются в записях решений. Документация развивается вместе с системой. Автоматические архитектурные проверки предотвращают постепенное разрушение структуры. Например, модуль предметной области не должен зависеть от пользовательского интерфейса.

4.4.5 Эволюционная архитектура

Эволюционная архитектура рассчитана на управляемое изменение. Она не означает отсутствие общей структуры. Напротив, изменение возможно только при ясных границах, тестах и автоматизации. К основным свойствам относятся модульность, слабая внешняя связанность, высокая внутренняя целостность компонентов, стабильные интерфейсы и автоматические проверки. Модульный монолит может быть эволюционной архитектурой, если модули изолированы. Микросервисная система может быть неэволюционной, если сервисы тесно связаны общими данными и требуют совместного выпуска. Архитектура должна оцениваться по стоимости изменения. Если добавление нового способа оплаты требует исправления десятков несвязанных модулей, границы ответственности выбраны неправильно. Применение функциональных переключателей позволяет отделить развертывание кода от включения функции. Новая реализация может присутствовать в системе, но оставаться выключенной до готовности. Для безопасной модернизации старых систем применяется постепенная замена. Новый компонент перехватывает часть функций, а старый постепенно сокращается. Такой подход уменьшает риск одномоментного переписывания.

4.4.6 Взаимосвязь архитектуры и DevOps

Архитектура определяет возможность независимой разработки, тестирования и выпуска. Если изменение одного модуля требует полной пересборки и длительной проверки всей системы, непрерывная поставка затрудняется. DORA отмечает, что технические и организационные структуры с более слабой связанностью поддерживают способность к непрерывной поставке. Микросервисы могут обеспечить независимый выпуск, но увеличивают сложность сети, данных и наблюдаемости. Успешная микросервисная архитектура требует зрелых DevOps-практик. Microsoft также указывает, что микросервисы требуют развитых инженерных и эксплуатационных возможностей. Модульный монолит часто позволяет начать с более простой эксплуатации и сохранить внутренние границы. Сервисы следует выделять при наличии конкретной причины: независимого масштабирования, отдельного жизненного цикла, тре-

бований безопасности или организационной автономии. Событийно-ориентированная архитектура ослабляет прямую связанность отправителей и получателей, но создает проблемы доставки, порядка и согласованности. DevOps должен учитываться при проектировании архитектуры. Необходимо заранее определить развертывание, конфигурацию, мониторинг, отказоустойчивость, обновление и восстановление. Система, которую невозможно удобно эксплуатировать, не является завершенным архитектурным решением.

4.4.7 Интегрированный пример применения методологий

Рассматривается проект создания информационной системы управления заявками технического обслуживания предприятия. На первом этапе проводится Lean-анализ потока. Устанавливается, что заявка несколько дней ожидает ручного распределения, информация дублируется, а исполнители не видят приоритет. Команда создает карту потока и определяет основную потерю — ожидание назначения. Цель продукта заключается в сокращении времени от регистрации заявки до начала работы. Для организации разработки используется Scrum. Владелец продукта формирует бэклог, включающий регистрацию заявки, классификацию, назначение, уведомление, контроль срока и отчетность. Первый спринт создает минимальную функцию регистрации и просмотра. На обзоре пользователи показывают, что необходима возможность приложить фотографию оборудования. Бэклог уточняется. Внутри спринта используется Kanban-доска со стадиями анализа, разработки, проверки и готовности. Для разработки и проверки устанавливаются ограничения незавершенной работы. Архитектура первоначально строится как модульный монолит. Выделяются модули заявок, пользователей, оборудования и уведомлений. Используется реляционная база данных. Интерфейс описывается в OpenAPI. Архитектурные схемы создаются по модели C4. Решения фиксируются в записях архитектурных решений. Исходный код и инфраструктура хранятся в Git. После каждого изменения запускаются сборка, модульные и интеграционные тесты, анализ качества и проверка зависимостей. Контейнерный образ публикуется в реестре и автоматически развертывается в тестовой среде. После подтверждения версия передается в промышленную среду. Инфраструктура описывается Terraform, конфигурация — Ansible. Метрики и журналы передаются в систему наблюдаемости. После внедрения Kanban используется для сопровождения. Анализ времени цикла показывает, что задачи долго ожидают проверки. Команда автоматизирует часть тестов и меняет ограничение незавершенной работы. DevSecOps-проверки контролируют зависимости, секреты и контейнерные образы. Эксплуатационная информация поступает в бэклог продукта. Таким образом, Lean определяет ценность и потери, Scrum организует итерационную разработку, Kanban управляет потоком, DevOps обеспечивает автоматизированную поставку и эксплуатацию, а архитектура и технологический стек создают техническую основу системы.

4.4.8 Основные принципы комплексного применения методологий

Методологии не должны применяться механически. Необходимо понимать проблему, которую решает каждый подход. Lean используется для анализа ценности и потока. Agile обеспечивает адаптацию к изменениям. Scrum создает ритм эмпирического управления продуктом. Kanban управляет непрерывным потоком и ограничивает перегрузку. DevOps объединяет создание и эксплуатацию. DevSecOps интегрирует безопасность. GitOps обеспечивает декларативное управление средой. MLOps охватывает жизненный цикл моделей. Архитектура должна поддерживать выбранный процесс. Частые изменения требуют модульности, автоматических тестов и воспроизводимой инфраструктуры. Технологический стек должен выбираться по требованиям и способности организации сопровождать решение, а не по моде. Инструменты должны поддерживать поток и прослеживаемость. Их количество не является показателем зрелости. Хорошо организованный простой конвейер полезнее большого набора плохо интегрированных продуктов. Метрики должны использоваться для улучшения системы, а не для давления на отдельных сотрудников. Документация должна быть достаточной, актуальной и связанной с реальной системой. Архитектурная схема, не обновлявшаяся несколько лет,

создает ложное представление. Гибкость не означает отказ от качества. Чем быстрее выполняются изменения, тем выше требования к автоматизации, безопасности, архитектуре и дисциплине. Качественная методология проектирования информационной системы формирует непрерывный цикл: выявление потребности, определение ценности, проектирование, реализация, автоматическая проверка, выпуск, наблюдение за эксплуатацией и последующее улучшение. Такой цикл позволяет системе развиваться вместе с потребностями пользователей, сохраняя управляемость, надежность и архитектурную целостность.

Конец ознакомительного фрагмента.