Управление ИТ-проектами

к.ф.-м.н., доц. Журенков Олег Викторович, zhur@pie-aael.ru

кафедра цифровых технологий и бизнес-аналитики (305С), тел. 296–547

Оглавление

Глава 1

Введение

1. Основные понятия

> ***Проект (project)*** *— ограниченная временными рамками деятельность, цель которой состоит в создании уникального продукта или услуги.*

Управление программным проектом является защитной деятельностью программной инженерии, пронизывающей все виды основной деятельности — подготовку, планирование, моделирование, конструирование и развёртывание. Именно эта деятельность интегрирует в жизненный цикл ПО все остальные виды защитной деятельности.

Руководство применяется ко всем четырём «П» разработки: персоналу (тем, кто делает продукт), процессу (работе, в результате которой делается продукт), проекту (среде, в которой делается продукт), продукту (результату всей деятельности).

> ***Управление проектом (project management)*** *— применение знаний, навыков, инструментов и методов для планирования и реализации действий, направленных на достижение поставленной цели в рамках проектных требований.*

Управление проектами — область деятельности, в ходе которой определяются и достигаются чёткие цели при балансировании между объёмом работ, ресурсами (такими как деньги, труд, материалы, энергия, пространство и др.), временем, качеством и рисками в рамках некоторой деятельности, направленной на достижение определённого результата при указанных ограничениях. В основе современных методов управления проектами лежат методики структуризации работ и сетевого планирования, разработанные в конце 50-х годов XX века в США.

Эти предложения зафиксированы в **PMBoK**, стандарте управления проектами, который удобно применять для проектов разработки ПО.

Методология проектирования ИС описывает процесс создания и сопровождения программных систем в виде *жизненного цикла (ЖЦ)*, представляя его как некоторую последовательность стадий и выполняемых на них процессов. **Жизненный**

цикл системы можно представить как ряд событий, происходящих с системой в процессе её создания и использования.

Для каждого этапа определяются состав и последовательность выполняемых работ, получаемые результаты, методы и средства, необходимые для выполнения работ, роли и ответственность участников и т. д. Такое формальное описание ЖЦ системы позволяет спланировать и организовать процессы коллективной разработки и обеспечить управление этим процессом.

Процессы состоят из деятельностей, в свою очередь деятельности состоят из действий, а действия — из задач.

> ***Деятельность*** *— элемент процесса, ориентированный на достижение весомой цели.*

> ***Действие*** *— набор задач, которые создают рабочий продукт на этапе ЖЦ.*

> ***Задача*** *служит для достижения маленькой, но конкретной цели.*

2. Основные стандарты управления ИТ-проектами

Национальные стандарты по проектному управлению давно существуют в США, Великобритании, странах ЕС и не только.

Наиболее известными стандартами являются:

NASA Project Management — США, известен своим сводом из 100 правил руководителей проектов NASA.

BSI BS 6079 — Великобритания (в 2010 году вышла третья редакция).

В конце 2011 года Федеральное агентство по техническому регулированию и метрологии РФ утвердило серию национальных стандартов:

- ГОСТ 54869-2011 Проектный менеджмент.
- ГОСТ 54871-2011 Управление программой проектов.
- ГОСТ 54870-2011 Управление портфелем проектов.

Эти стандарты введены в действие 1 сентября 2012 года.

2.1. PMBoK

PMBoK (Project Management Body of Knowledge) — *«Свод знаний по управлению проектами»*, который представляет собой сумму профессиональных знаний по управлению проектами. Руководство PMBoK фиксирует части Свода знаний по управлению проектами, которая обычно считается хорошей практикой. Сейчас используется 6-я редакция PMBoK, выпущенная в 2017 году.

PMBoK является стандартом Международного института управления проектами (Project Management Instituite, PMI). PMI — некоммерческая организация, которая поставила себе цель — развивать профессионализм в управлении проектами, проводить сертификацию менеджеров проектов, добиваться признания профессии «руководитель проекта». Организация имеет представительства в более чем 170 странах. В странах СНГ PMI имеет представительства в России, Украине, Казахстане.

Шестое издание отражает опыт и знания руководителей-практиков и освещает фундаментальные основы управления широким кругом проектов. Этот признанный международный стандарт предоставляет руководителям проектов важнейшие инструменты для управления проектами и достижения результатов в организации работы.

Была добавлена 10-я область знаний; в которой управление заинтересованными сторонами проекта расширяет важность привлечения соответствующих заинтересованных сторон проекта в принятие ключевых решений и мероприятия.

Был пересмотрен поток данных и информации проекта для обеспечения согласованности и соответствия модели информационной иерархии, включающей данные, информацию, знания и мудрость (Data, Information, Knowledge and Wisdom, DIKW), которая используется в области управления знаниями.

Были добавлены четыре новых процесса планирования: планирование управления содержанием, планирование управления расписанием, планирование управления стоимостью и планирование управления заинтересованными сторонами. Эти процессы были созданы, чтобы укрепить концепцию, в которой каждый из вспомогательных планов интегрирован через общий план управления проектом.

PMBoK — энциклопедия, объединяющая все знания в области управления проектами. Его можно назвать своеобразной энциклопедией, однако в качестве практического руководства PMBoK рассматривать не следует. В нём содержится лишь информация для изучения, описания методов и рекомендации к работе.

Так что называть PMBoK методологией неверно. Информация в руководстве ещё не адаптирована к нуждам проектного окружения и организации и подходит для большинства проектов, как универсальная заготовка, а не готовая методология.

Однако её следует изучить каждому менеджеру проектов. В руководстве описана суть процессов управления проектами и цели, которым они служат.

2.2. PRINCE2

PRINCE2 (PRojects IN Controlled Environments 2) — один из наиболее известных и популярных стандартов ведения ИТ проектов. Он был разработан в 1989 году Central Computer and Telecommunications Agency (CCTA) Великобритании как стандарт для руководства проектами в области ИТ. В настоящее время он широко используется и в других областях и является «de facto» стандартом для руководства

проектами в Великобритании. Стандарт описывает принципы организации программы проектов и проектов в рамках программы.

PRINCE2 популярен в Великобритании, США, Канаде, странах Восточной и Северной Европы, отчего и получил свое название.

На западе его используют многие крупные компании (например, Siemens, Bank of New York, Philips, Microsoft, Unilever, Tesco, Philip Morris UK, Tesco, Shell, Nokia, Hitachi), крупнейшие банки и телекоммуникационные операторы Великобритании, и другие.

В России принципы PRINCE2 применяют в международных британских фирмах, как например, British American Tobacco и British Telecom, а также в крупнейших международных американских корпорациях (IBM и Hewlett Packard Enterprise).

PRINCE2 позволяет вести проекты системно, при этом легко масштабируется и интегрируется с другими методологиями, как например, Agile или CMMI. У **PRINCE2** нет своего регламента по методам и подходам, подразумевается, что каждый менеджер использует те, которыми привык пользоваться.

В нём выделены 7 принципов управления проектами:

1. *Постоянная оценка* соответствия проекта потребностям бизнеса.
2. *Учёт уроков проекта*, «самообучение» проекта.
3. *Выделение определённых ролей* с ответственностью и правами.
4. *Управление по фазам.* Текущая фаза завершается планированием следующей.
5. *Управление по отклонениям.* Каждая роль имеет полномочия на утверждение конкретных отклонений.
6. *Ориентация на результат.* Главное — именно результат, а не процессы его достижения (в отличие, например, от PMBoK).
7. *Адаптация стандарта* под конкретную организацию и внешнюю среду.

Проекты в PRINCE2 делятся на фазы с чётко заданным временем начала и окончания.

1. *Начало проекта (Starting Up a Project)* определение того, надо ли выполнять проект и выбор подхода к его реализации.
2. *Инициация проекта (Initiating a Project)* — создание бизнес-кейса проекта, формирование оргструктуры, плана и бюджета проекта.
3. *Планирование (Planning)* — детальное планирование и перепланирование проекта, выполняется в течение всего проекта.
4. *Руководство проектом (Directing a Project)* — принятие Комитетом проекта решений по контрольным точкам и ситуационное управление по значительным проблемам и отклонениям. Выполняется в течение всего проекта.

5. *Систематический контроль стадий (Controlling a Stage)* — непосредственная работа менеджера проекта по ежедневному управлению проектом (выдача и приёмка заданий, фиксация сложностей и рисков, принятие решения об эскалации и отчётность).

6. *Управление поставкой продукта (Managing Product Delivery)* — действия по созданию нужных для проекта продуктов, выполняемые менеджером проекта.

7. *Управление границами стадий (Managing Stage Boundaries)* — анализ исполнения плана стадии, промежуточное планирование следующей стадии, обзор рисков и предоставление информации Комитету проекта.

8. *Завершение проекта (Closing a Project)* — действия по закрытию проекта.

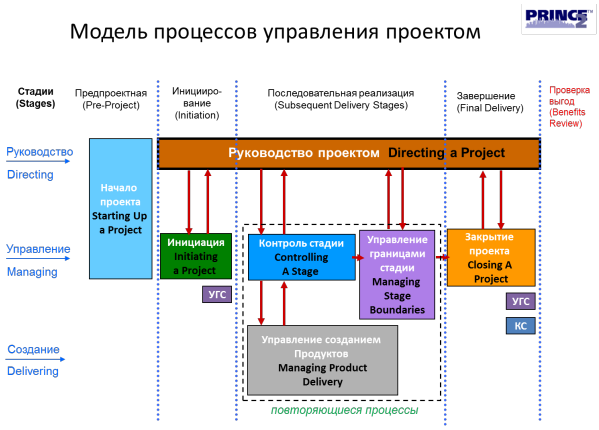


Рис. 1.1. Процессы PRINCE2

Стадии в стандарте не являются последовательными, зависимости между ними более сложные. Каждая стадия детализирована по активностям (всего их 45), входящим и исходящим артефактам, критериям, позволяющим начать активность.

PRINCE2 выделяет 7 артефактов (документов и тем), на которых надо фокусироваться при выполнении проекта:

1. Соответствие потребностям бизнеса (бизнес-кейс).

2. Оргструктура проекта.

3. Обеспечение качества (по трём направлениям: финансовое состояние проекта, выполнение требований пользователей, технологическое качество результата — продукта проекта).

4. Планирование.

5. Управление рисками.

6. Управление изменениями.

7. Отчётность (в частности, план/факт).

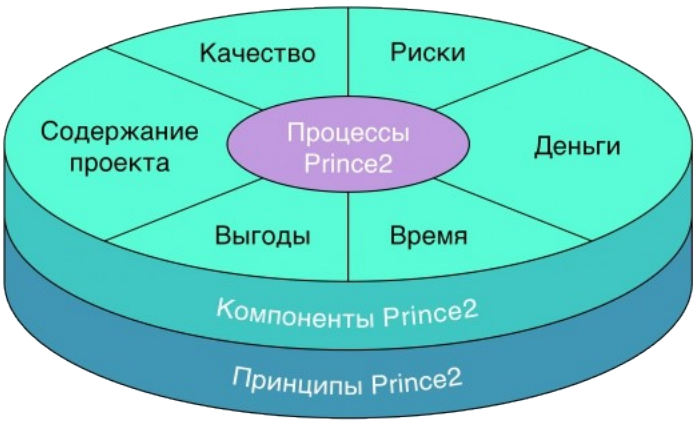


Рис. 1.2. Структура PRINCE2

Основной принцип — планирование от продукта (product-based planning). Это значит, что работа над проектом начинается с обсуждения того, как должен выглядеть финальный продукт. Затем его разбивают на более мелкие субпродукты (элементы функционала).

Следующий этап — построение карты последовательности продуктов (product flowchart), которая описывает все взаимозависимости субпродуктов и последовательности при реализации. Фактически проектный план выглядит не как дерево задач, а дерево продуктов.

Все результаты делятся на две категории — продукты управления (management products) и специализированные продукты (specialist products).

Продукты управления — это все документы, подготовленные в ходе работы. Обязательный — **PID**.

> ***Project Initiation Document (PID)*** *— документ, который включает в себя Project Brief с бизнес-кейсом, описание продуктов (Product Description), карты последовательности продуктов (Product Flowchart), и детальный проектный план.*

PID — достаточно объёмный документ, включающий Project Brief с бизнес-кейсом (то, что некоторые называют Project Vision), описание продуктов (нечто наподобие User Stories из Scrum), диаграммы производства продукта, и, соответственно детальный проектный план.

Большим преимуществом стандарта является его глубокая проработанность и универсальность. В целом **PRINCE2** является методологией, готовой к использованию для выполнения ИТ-проектов и проектов по разработке ПО.

Главным недостатком **PRINCE2** является высокая сложность применения. Распространение его в России сдерживает также отсутствие официального перевода на русский язык.

PMBoK Guide и **PRINCE2** работают на одну цель и сильно пересекаются. На их стыке образовался **P3.express**.

2.3. P3.express

P3.express взял лучшие черты из фундаментальных и гибких методов, попытавшись преодолеть многие их недостатки. По сути, это такой гибрид.

P3.express не использует итеративный подход к проекту.

P3.express, как и PRINCE2, предписывает разбивать конечный продукт на субпродукты. При этом каждый субпродукт получает свой product description (описание).

Дальше выстраивают product workflow diagram — очерёдность, по которой субпродукты должны быть готовы.

PRINCE2 завязан на бесконечных отчётах и отписках, поэтому система недостаточно гибка и медленно реагирует на изменения.

PRINCE2 часто критикуют из-за того, что там отсутствует ретроспектива и общение с командой, так называемый «мягкий менеджмент». Из-за чего возникают недопонимания, конфликты в команде, и проект стопорится. Людям, как минимум, нужно выговориться и выпустить пар, а как максимум — подвести промежуточные или окончательные итоги по проекту и поделиться впечатлениями.

Всё это учёл P3.express.

P3.express — это фреймворк, — полностью интегрированный пакет из методологий, практик и инструментов. Больше не нужно беспокоиться о том, как выбрать нужные техники для проекта, или выяснять, как использовать уже привычные программы.

P3.express работает по принципу Парето — 20% усилий дают 80% результата.

Фазы P3.express

Каждая фаза предполагает активную коммуникацию между всеми участниками процесса, чтобы убедиться, что каждый работает на общую цель проекта, и не будет конфликтов между командами и внешними исполнителями.

Подготовка проекта

1. Назначить спонсора
2. Подготовить резюме проекта
3. Назначить руководителя проекта
4. Развернуть информационную систему
5. Назначить команду
6. Планирование проекта
7. [Выбрать внешних исполнителей]
8. Провести аудит подготовки
9. Да/Нет (принятие решения)
10. Провести стартовую встречу

11. Фокусированная коммуникация

Планирование цикла

12. Обновить планы

13. [Выбрать внешних исполнителей]

14. Да/Нет

15. Провести стартовую встречу цикла

16. Фокусированная коммуникация

17. Измерить и задокументировать ход проекта

18. Работать с отклонениями

19. Еженедельная встреча

20. Провести еженедельный аудит

21. Фокусированная коммуникация

Ежедневные действия

22. Зафиксировать RICs

23. Реагировать на RICs на основе делегированных полномочий

24. Принять готовые продукты от Тимлидов и исполнителей

25. Оценить удовлетворённость заказчика и команды

26. Запланировать улучшения

27. Фокусированная коммуникация

Закрытие проекта

28. Получить одобрение и передать продукт

29. Передать Бизнес-кейс ответственному лицу

30. Оценить удовлетворённость заказчика и команды

31. Провести аудит проекта

32. Извлечь уроки и заархивировать проект

33. Объявить и отметить окончание проекта

34. Фокусированная коммуникация

35. Свериться с Бизнес-кейсом и оценить выгоды

36. Спланировать дополнительные действия, если необходимо

37. Фокусированная коммуникация

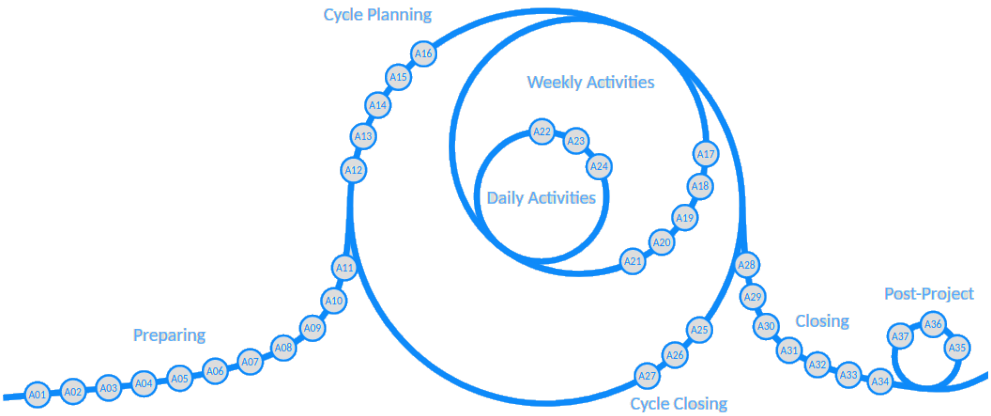


Рис. 1.3. Фазы P3.express

P3.express — пока что самое новое и современное, что нам предлагает проектный менеджмент. Ожидается, что скоро **P3.express** начнёт активно продвигаться в России. Однако никакая методология — не панацея.

2.4. РИМ-III

> ***РИМ-III*** *(Результат, Инструмент, Метод) — российская инструментальная модель управления проектами.*

Модель анонсирована в декабре 2014 г. С 1 января 2015 г. открыта для обсуждения (rim-iii.postach.io). Принята на вооружение в федеральных органах исполнительной власти и органах госвласти субъектов РФ при Минэкономразвития, начинает набирать популярность в коммерческих компаниях.

РИМ-III — это трёхуровневая система управления проектами. Компания развивается в трёх направлениях:

- непосредственное управление проектами — реализация задач высокой сложности;
- управление информацией и знаниями — построение комплексной системы;
- цифровая трансформация бизнеса — внедрение ИТ-систем.

Эта модель ориентирована на российский рынок.

3. Процессы управления проектом

Исходной идеей для взаимодействия между процессами управления проектом является цикл «планирование-исполнение-проверка-воздействие» (предложенный Уолтером А. Шьюартом и доработанный У. Эдвардсом Демингом). Этот цикл связан результатами — результат одной части цикла становится входом другой части (см. рис. 1.4).

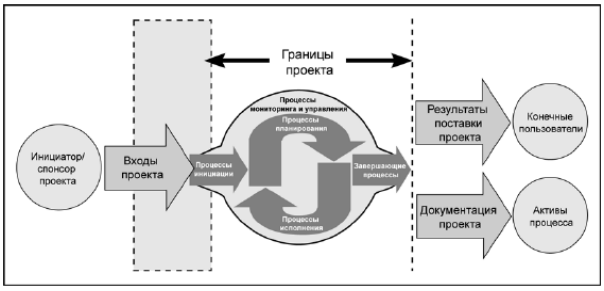


Рис. 1.4. Границы проекта

Управление проектами включает следующие этапы:

1. *Группа процессов инициации.* В группу процессов инициации входят следующие процессы управления проектами:

   - *Разработка Устава проекта* — процесс, связанный прежде всего с авторизацией проекта или фазы проекта (в многофазном проекте). Это процесс, необходимый для формулирования практических нужд и документального оформления нового продукта, услуги или иного результата, который должен удовлетворять этим требованиям. С помощью Устава проект привязывается к текущей работе организации, а также осуществляется авторизация проекта. Составление Устава проекта и авторизация проекта проводятся вне рамок проекта подразделением, управляющим организацией, программой или портфелем. В многофазных проектах в ходе этого процесса оцениваются или исправляются решения, принятые в предыдущем процессе разработки Устава проекта на предыдущей фазе.
   - *Разработка предварительного описания содержания проекта* — процесс, необходимый для предварительного общего описания проекта с использованием Устава проекта и других входов процессов инициации. Этот процесс направляет и документирует требования к проекту и результатам поставки, требования к продукту, границы проекта, методы приёмки и общее управление содержанием. В многофазных проектах этот процесс оценивает или уточняет содержание проекта для каждой фазы.

2. *Группа процессов планирования* способствует планированию проекта. В нижеприведённом списке указываются процессы, к которым команда проекта должна обратиться, чтобы решить нужно ли их выполнять, и если да, то кем. В

группу процессов планирования входят следующие процессы управления проектами:

- *Разработка плана управления проектом* — процесс, необходимый для определения, подготовки, координации и интеграции всех вспомогательных планов в план управления проектом. План управления проектом становится первичным источником информации по планированию, исполнению, мониторингу и управлению, а также закрытию проекта.
- *Планирование содержания* — процесс, необходимый для создания плана управления содержанием проекта, который описывает, как будет определяться, проверяться и управляться содержание проекта и как будет создана и определена иерархическая структура работ.
- *Определение содержания* — процесс, необходимый для разработки подробного описания содержания проекта, на основании которого будут впоследствии приниматься решения по проекту.
- *Создание иерархической структуры работ* — процесс, необходимый для разделения основных результатов поставки проекта и работ проекта на меньшие элементы, которыми легче управлять.
- *Определение состава операций* — процесс, необходимый для идентификации конкретных операций, которые следует выполнить для получения различных результатов поставки проекта.
- *Определение взаимосвязей операций* — процесс, необходимый для определения и документирования взаимосвязей между операциями.
- *Оценка ресурсов операций* — процесс, необходимый для оценки типа и количества ресурсов, необходимых для выполнения каждой плановой операции.
- *Оценка длительности операций* — процесс, необходимый для оценки количества рабочих периодов, которые потребуются для завершения отдельных плановых операций.
- *Разработка расписания* — процесс, необходимый для анализа последовательности операций, длительности операций, требований к ресурсам и ограничений на сроки с целью создания расписания проекта.
- *Стоимостная оценка* — процесс, необходимый для разработки приблизительных значений стоимости ресурсов, необходимых для выполнения операций проекта.
- *Разработка бюджета расходов* — процесс, необходимый для суммирования оценок стоимости отдельных операций или пакетов работ для оценки базового плана по стоимости.
- *Планирование качества* — процесс, необходимый для определения стандартов качества, которые соответствуют проекту, и средств достижения этих стандартов.
- *Планирование человеческих ресурсов* — процесс, необходимый для определения и документирования ролей в проекте, ответственности и отчётности, а также создания плана управления обеспечением проекта персоналом.

- *Планирование коммуникаций* — процесс, необходимый для определения потребностей участников проекта в информации и коммуникациях.

- *Планирование управления рисками* — процесс, необходимый для определения подходов к планированию и выполнению операций по управлению рисками проекта.
- *Идентификация рисков* — процесс, необходимый для определения того, какие именно риски могут повлиять на проект, а также для документирования их характеристик.
- *Качественный анализ рисков* — процесс, необходимый для установления приоритетов рисков с целью их дальнейшего анализа или действий путём оценки и совмещения их вероятности и воздействия.
- *Количественный анализ рисков* — процесс, необходимый для количественной оценки риска, влияющего на общие цели проекта.
- *Планирование реагирования на риски* — процесс, необходимый для разработки вариантов и операций для повышения возможностей и снижения угроз целям проекта.
- *Планирование покупок* — процесс, необходимый для определения, что, как и когда следует приобрести.
- *Планирование контрактов* — процесс, необходимый для документирования требований к продуктам, услугам и результатам, а также для поиска потенциальных продавцов.

Задачи планирования и контроля развития проекта рассматриваются в качестве основы производства программной продукции. Они важны при любой методологии, но каждая из них понимает планирование и контроль по-своему.

3. *Группа процессов исполнения.* Обычно при исполнении имеют место отклонения, приводящие к корректировке планов. Эти отклонения могут затрагивать длительность операций, наличие и эффективность ресурсов, а также непредусмотренные риски. Независимо от того, повлияют такие отклонения на план управления проектом или нет, они могут потребовать анализа. Результаты этого анализа могут повлечь за собой запрос на изменение. Если этот запрос будет одобрен, то это может привести к изменению плана управления проектом и, возможно, утверждению нового базового плана. Подавляющая часть бюджета проекта пойдет на выполнение группы процессов исполнения. В группу процессов исполнения входят следующие процессы управления проектами:

- *Руководство и управление исполнением проекта* — процесс, необходимый для управления различными организационными и техническими интерфейсами, имеющимися в проекте, для выполнения работ, предусмотренных в плане управления проектом. Результаты поставки представляются как выходы выполненных процессов, указанных в плане управления

проектом. По мере выполнения проекта собирается информация о завершении подготовки результатов поставки и о том, какие именно работы завершены. Эта информация становится входом для процесса отчётности по исполнению.

- *Процесс обеспечения качества* — процесс, необходимый для применения плановых систематических операций по проверке качества (например, аудит или независимая экспертиза), чтобы удостовериться, что в проекте используются все необходимые процессы для выполнения требований.
- *Набор команды проекта* — процесс, необходимый для получения человеческих ресурсов, нужных для выполнения проекта.
- *Развитие команды проекта* — процесс, необходимый для повышения компетенции и взаимодействия членов команды для улучшения исполнения проекта.
- *Распространение информации* — процесс, необходимый для обеспечения участников проекта нужной им информацией.
- *Запрос информации у продавцов* — процесс, необходимый для получения информации, расценок или предложений.
- *Выбор продавцов* — процесс, необходимый для изучения предложений, выбора из потенциальных продавцов и заключения письменного контракта с продавцом.

4. *Группа процессов мониторинга и управления.* В эту группу входят следующие процессы управления проектами:

- *Мониторинг и управление работами проекта* — процесс, необходимый для сбора, измерения и распространения информации об исполнении проекта и оценки измерений и тенденций для влияния на улучшение процессов. Этот процесс включает в себя мониторинг рисков, что позволяет обеспечить выявление рисков на ранних стадиях, после чего составляется отчёт об их состоянии и приводятся в исполнение соответствующие планы реагирования на риски. Мониторинг включает в себя отчёты о текущем состоянии, оценку прогресса и прогнозирование. Отчёты об исполнении предоставляют информацию об исполнении проекта по таким показателям, как содержание, расписание, стоимость, ресурсы, качество и риски.
- *Общее управление изменениями* — процесс, необходимый для управления факторами, создающими изменения, чтобы эти изменения были благотворными. Он необходим также для отслеживания внесения изменений и для управления одобренными изменениями, в том числе временем их обработки. Этот процесс выполняется в течение всего проекта, от инициации до закрытия проекта.
- *Подтверждение содержания* — процесс, необходимый для формализации приёмки завершённых результатов поставки проекта.

- *Управление содержанием* — процесс, необходимый для управления изменениями в содержании проекта.
- *Управление расписанием* — процесс, необходимый для управления изменениями в расписании проекта.
- *Управление стоимостью* — процесс влияния на факторы, создающие отклонения, и управление изменениями бюджета проекта. Это процессы, касающиеся планирования, оценки, разработки бюджета и контролирования затрат, так чтобы проект был завершён в пределах одобренного бюджета.
- *Управление командой проекта* — процесс, необходимый для отслеживания деятельности членов команды, обеспечения обратной связи, решения проблем и координации изменений с целью улучшения исполнения проекта.
- *Отчётность по исполнению* — процесс, необходимый для сбора и распространения информации об исполнении. Эта информация включает в себя отчёты о текущем состоянии, оценку прогресса, а также прогнозирование.
- *Управление участниками проекта* — процесс, необходимый для управления коммуникациями с целью удовлетворения требований участников проекта и решения вместе с ними возникающих проблем.
- *Наблюдение и управление рисками* — процесс, необходимый для отслеживания выявленных рисков, мониторинга остаточных рисков, выявления новых рисков, выполнения планов реагирования на риски и оценки их эффективности в течение жизненного цикла проекта.
- *Процесс контроля качества* — процесс, необходимый для мониторинга определённых результатов проекта с целью определения их соответствия принятым стандартам качества и выработки путей устранения причин неудовлетворительного исполнения.
- *Администрирование контрактов* — процесс, необходимый для управления контрактом и взаимоотношениями между продавцом и покупателем, для изучения и документирования действий продавца и, в соответствующих случаях, для управления контрактными отношениями с внешним покупателем проекта.

5. *Группа завершающих процессов.* В группу завершающих процессов входят следующие процессы управления проектами:

- *Закрытие проекта* — процесс, необходимый для завершения всех операций всех групп процессов, чтобы формально закрыть проект или фазу проекта.
- *Закрытие контрактов* — процесс, необходимый для завершения и урегулирования каждого контракта, в том числе завершение действующих контрактов и закрытия каждого контракта, затрагивающего проект или фазу проекта.

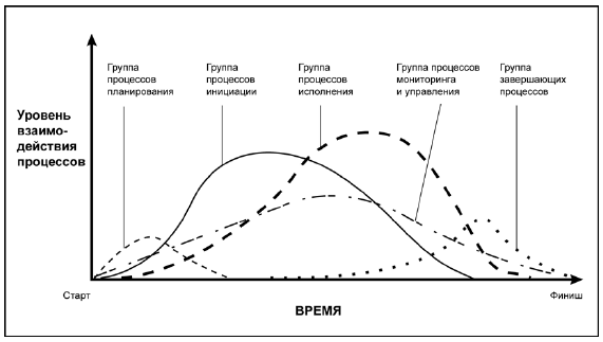


Рис. 1.5. Взаимодействие групп процессов в проекте

Группы процессов управления проектом связаны целями, которые перед ними поставлены. Выход одного процесса обычно является входом для другого процесса или является результатом поставки проекта. Группа процессов планирования предоставляет группе процессов исполнения документированный план управления проектом и описание содержания проекта, а также часто вносит изменения в план управления проектом по ходу проекта.

Необходимо также отметить, что группы процессов редко являются дискретными или однократными событиями; они являются накладывающимися друг на друга действиями, осуществляемыми с той или иной интенсивностью в течение жизненного цикла проекта. На рис. 1.5 изображено, как взаимодействуют группы процессов, а также показан уровень наложения в различные периоды осуществления проекта. Если проект разделён на фазы, группы процессов взаимодействуют в рамках фазы проекта и могут также пересекать границы фаз.

Результаты процессов связаны с другими группами процессов и воздействуют на них. Например, для завершения фазы проектирования необходима приёмка проекта заказчиком. После этого проектный документ определяет описание продукта для последующей группы процессов исполнения. Когда процесс разделяется на фазы, группы процессов обычно связаны с каждой фазой на протяжении существования проекта, чтобы способствовать успешному завершению проекта.

Однако не все процессы могут понадобиться в каждом конкретном выполняемом проекте или его фазе, и не все взаимодействия могут быть к ним применимы.

3.1. Участники процесса разработки ИС

Основные участники процесса разработки ИС и ПО следующие:

Пользователь — лицо или организация, которое использует ПО для выполнения конкретной функции.

Заказчик — лицо (физическое или юридическое), заинтересованное в выполнении исполнителем разработки ПО, оказании им услуг (связанным с ПО) или приобретении у него ПО. Иногда при этом предполагается оформление заказа, но не обязательно.

Разработчик — специалист, занимающийся написанием и корректировкой программ для ЭВМ (программированием).

Руководитель проекта — человек, задачей которого является организация всего процесса разработки ПО.

Аналитик — специалист по предметной области, для которой разрабатывается ПО. Также в его задачи входит подготовка документации по программным требованиям и спецификациям (Software Requirements Specification). Аналитик ПО — промежуточное звено между пользователями и разработчиками. Благодаря этому они могут предъявлять разработчикам требования пользователей разрабатываемых программ. Аналитик ПО должен обладать следующими навыками: знание технологий, связанных с программным обеспечением; опыт программирования; знание в области бизнеса и маркетинга; навыки решения проблем; умение налаживать отношения с людьми; иметь гибкий и адаптирующийся склад ума.

Тестировщик — специалист, занимающийся тестированием ПО.

Поставщик — лицо, которое занимается поставкой, установкой, настройкой ПО и обучением пользователей.

Глава 2

Управление человеческими ресурсами проекта

1. Основные понятия

Самым ценным ресурсом программного проекта являются, конечно, люди. Вне сомнений, в первую очередь важны технические навыки инженеров-разработчиков. Однако эти навыки необходимо применять для решения проблем в нужное время и в нужном месте. Следовательно, предполагается комбинация двух стилей: работа в команде и лидерство.

Организация команды, обеспечивающей эффективную работу, является весьма сложной задачей для *менеджера — руководителя проекта*. Хорошая команда должна демонстрировать сплав самых разнообразных качеств: *профессиональные навыки*, *опыт*, *сплочённость*, *дух товарищества*. Структура команды должна стимулировать творческую работу всех и каждого. Выражаясь языком известнейших в этой области авторов Тома Демарко и Тимоти Листера, «команда должна пройти *кристаллизацию*». Они пишут:

> «Команда, прошедшая кристаллизацию, — это группа людей, столь сильно связанных, что целое становится больше суммы составляющих его частей. Производительность этой команды выше, чем производительность тех же людей, не перешедших порог кристаллизации. И, что столь же важно, удовольствие от работы также выше, чем можно было бы ожидать, учитывая природу работы. В некоторых случаях кристаллизованная команда может замечательно себя чувствовать, работая над задачей, которую другие посчитали бы откровенно скучной.
>
> Как только начинается кристаллизация команды, вероятность успеха очень резко возрастает. Команда может стать неумолимой силой, стремящейся к успеху. Управлять этой стихией — одно удовольствие. Управление в традиционном смысле этого слова им не нужно, и уж точно не нужны дополнительные стимулы. Они уже обладают собственным импульсом.
>
> Причины такого явления не столь сложны: команды по природе своей склонны формироваться при наличии целей. До кристаллизации команды её участники, возможно, имели различные цели. Однако в процессе

кристаллизации каждый поверил в общую цель. Эта корпоративная цель обретает особую важность по причине её значимости для группы. И хотя сама по себе цель может казаться участникам команды выбранной наудачу, они стремятся к её достижению с невероятным напором».

Обеспечить такую *кристаллизацию* — главная задача руководителя проекта.

2. Подбор членов команды

Прежде всего, руководитель должен организовать правильный подбор членов команды: они могут дополнять друг друга по навыкам и опыту и должны быть совместимы друг с другом психологически.

При работе с кандидатом в команду менеджер должен учитывать следующие аспекты:

1. Опыт работы во многих аппаратно-программных средах.
2. Знание языков программирования.
3. Образование и опыт работы по специальности.

   Образование является комплексным показателем начальных знаний и навыков кандидата, а также его способности к обучению. Опыт же характеризует конечные знания и навыки специалиста.
4. Коммуникабельность.

   Коммуникабельность характеризует возможности общения с коллегами, руководителями и другими заинтересованными в проекте лицами.
5. Способность адаптироваться.

   Способность к адаптации может пояснять «послужной список» — имеющийся рабочий стаж.
6. Жизненная позиция.

   О жизненной позиции судить трудно, но важно. Ведь она говорит о любви к профессии и стремлении развивать знания и навыки.
7. Личностные качества.

   Личностные качества оценить, пожалуй, труднее всего. Здесь и психологический портрет, и темперамент, и инициативность, и целеустремленность, и многое другое. Именно эти качества определяют совместимость кандидата с коллективом.

Конечно, руководители должны не на словах, а на деле учитывать типичные *человеческие факторы* сотрудников. Люди хотят иметь интересную работу, хотят иметь возможность проявить себя, хотят, чтобы их заметили и наградили, и хотят тёплых дружеских отношений в коллективе. Здоровое самоуважение является предпосылкой этих желаний.

Один из источников самоуважения — качественная работа. Следовательно, сотрудники должны точно знать, что означает *качество*. Например, сотрудники должны знать, как оценить трудозатраты на создание хорошего продукта, как доказать себе и другим, что работа сделана корректно, и как измерить качество сделанной работы.

Важно также правильно выбрать *лидера* команды. Он отвечает за *техническое руководство* или за *административное управление* (возможно и совмещение этих обязанностей). Лидеры должны быть в курсе повседневной деятельности команды, обеспечивая её эффективную работу и сотрудничество с руководством проекта. Они должны ладить со всеми членами коллектива, смягчая напряжённости и неприятности.

Лидеры во многом обеспечивают сплочённость команды, чувствуя самые тонкие нюансы профессиональных и личностных отношений и помогая отдельным сотрудникам преодолевать трудности. Они «несут знамя командного духа», воспитывают чувство единения, ответственности за работу всей команды.

Как отмечает Б. Шнейдерман (ссылаясь на Г. Вейнберга) в сплочённой команде возможно преобладание *стиля «программирования без персонализации»*.

При программировании без персонализации все рабочие продукты (модели, код, документация) считаются собственностью всей команды, а не отдельного сотрудника, который занимался их созданием.

Преимущества *разработки без персонализации:*

1. упрощение процедур проверки, критики недостатков, повышение их объективности;
2. поощрение стиля непринуждённого обсуждения рабочих заданий, достоинств и недостатков отдельных решений;
3. активизация дружеских отношений, повышение уровня искренности;
4. быстрый рост мастерства (благодаря работе бок о бок);
5. улучшение качества, совершенствование результатов работы.

Несомненно, опытные руководители способствуют сплочённости команды, регулярно проводя в этих целях различные совещания, собрания, дискуссии, диспуты, устраивая совместные празднования и другие неофициальные мероприятия.

3. Взаимодействия в команде

Для команды ИТ проекта необходима развитая система взаимодействия, иными словами, общение и хорошие средства связи между сотрудниками. Сотрудники должны информировать друг друга о ходе работы, принимаемых решениях, а также об изменениях, которые внесены в предыдущие решения. Постоянное взаимодействие тоже способствует сплочённости и повышению качества работы, поскольку сотрудники совместно обсуждают решения, начинают лучше понимать мотивацию своих коллег.

На эффективность взаимодействия влияют следующие параметры:

- *Размер/структура команды. С ростом числа участников количество связей по взаимодействию растёт квадратично.*

  Например, между тремя участниками есть три связи, четыре участника имеют шесть связей, пять человек — десять связей, то есть n человек имеют $(n-1)+(n-2)+\cdots+1 = n(n-1)/2$ связей (каждый с каждым). Следовательно, 50 человек должны участвовать в 1225 взаимодействиях! Для больших команд альтернативой является их разделение на группы. Каждая группа отвечает за определённую часть проекта и работает над ней. Обычно численность группы не превышает восьми человек. В таких группах проблемы взаимодействия исчезают. Для взаимодействия с другими группами в каждой группе выделяется один сотрудник. Такая структура сохраняет преимущества небольших команд, но позволяет большому количеству людей создавать большие программные продукты.

- *Иерархия команды.* Сотрудники в команде с горизонтальной организацией (один уровень, все сотрудники равны) легче общаются между собой, чем в командах с многоуровневой организацией и иерархией отношений (начальники/подчиненные). В последних взаимодействие происходит между уровнями, в иерархической последовательности. Сотрудники одного уровня могут вовсе не общаться между собой. Если в такой команде сотрудники разных групп общаются только через своих менеджеров, возможны запаздывания в разработке и проблемы недопонимания.

- *Рабочее окружение.* Организация рабочего места оказывает существенное влияние на производительность труда. Психологи доказали, что человек всегда предпочитает работать в отдельном помещении, которое он может оформить по своему вкусу и в котором может сосредоточиться. Исследования подтвердили — в открытом помещении сотруднику сконцентрироваться труднее, следствием чего становится снижение рабочей активности (производительность труда может упасть в два раза).

4. Состав команды

Увы, не существует универсального рецепта для определения оптимального состава группы разработчиков. Состав зависит от большого числа разнообразных факторов: стиля менеджмента, принятого в организации, предметной области и размера проекта, профессиональных возможностей сотрудников организации и т. д.

Типовые роли:

Руководитель проекта (project manager) — отвечает за работу над проектом, управляет командой.

Аналитик — отвечает за развитие и интерпретацию требований заказчика; должен быть экспертом в предметной области, но работать в тесном контакте с остальными сотрудниками.

Архитектор — отвечает за проектирование и развитие архитектуры продукта, является одним из наиболее квалифицированных специалистов, имеющих опыт принятия стратегических решений; кроме опыта проектирования, архитектор должен уметь программировать, поскольку его решения воплощаются в программном коде.

Конструктор компонентов — главный создатель компонентов (строительных кирпичиков, из которых компонуется продукт).

Программист — программирует компоненты и отвечает за их сборку (в веб-разработке разделяется на бэк-енд и фронт-енд разработчиков).

Специалист по повторному использованию — внедряет в продукт готовые компоненты из сторонних коммерческих библиотек.

Специалист по интеграции — отвечает за сборку версий компонентов и проверку правильности их совместной работы, поддерживает выпуск версий продукта.

Тестировщик — специалист по составлению тестов, тестированию и анализу результатов тестирования. В его обязанность входит поиск вероятных ошибок и сбоев в функционировании программы. Тестировщик моделирует различные ситуации, которые могут возникнуть в процессе использования программы, чтобы разработчики смогли исправить обнаруженные ошибки.

Тестировщик также создаёт и использует разнообразные входные данные, предусмотренные и не предусмотренные авторами программы. Его деятельность обычно подразумевает как минимум три модели поведения:

1. Пользователь не читал инструкций или неспособен их прочитать. Находится несоответствие интерфейса программы существующим стереотипам.
2. Добросовестный пользователь действует в строгом соответствии с инструкциями. Поиск ошибок как в логике работы программы, так и в документации на программу.
3. Злонамеренный пользователь стремится использовать программу непредусмотренным способом.

В случае создания программ с различными уровнями защиты и прав доступа для пользователя количество возможных моделей поведения («начальник», «подчинённый») существенно увеличивается.

Альфа-тестер — пользователь программы, находящейся на стадии разработки («Альфа-версия», как правило не полнофункциональная), взявший на себя в какой-либо форме обязательства по полному или частичному тестированию программы, а также, возможно, об особых условиях её копирования и использования.

Бета-тестер — пользователь программы, взявший на себя обязательства по тестированию программы («Бета-версия»), в том числе опубликованных официально версий и так называемых «релиз-кандидатов» программы.

В разных случаях отношения альфа- и бета-тестеров с разработчиками могут оформляться или не оформляться. Ряд пользователей добровольно участвует в бета-тестировании программного обеспечения.

Технический писатель — специалист по документации, документирует все реализованные решения, готовит документацию для пользователя.

Системный программист — отвечает за создание и адаптацию программных утилит, облегчающих работу над проектом.

Системный администратор — управляет физическими компьютерными ресурсами в проекте, а также средами и платформами (ОС, СУБД).

Разумеется, не каждый проект требует исполнения всех этих ролей. В небольших проектах сотрудники могут играть сразу несколько ролей.

Группа разработчиков обычно пользуется услугами вспомогательного технического персонала и *программной библиотеки*.

Библиотека обслуживает многие группы, она *выполняет сопровождение и контроль всех элементов программной конфигурации* (документации, листингов, данных, моделей, результатов измерений, проверки, оценки), причём хранит весь архив данных по прошлым, уже выполненным проектам.

Глава 3

Управление конфигурацией

1. Основные понятия

> ***Управление конфигурацией ПО*** *(Software Configuration Management,* ***SCM****) — это комплекс методов, направленных на:*
>
> - *систематический учёт изменений, вносимых разработчиками в ПО в процессе его разработки и сопровождения;*
> - *сохранение целостности системы после изменений;*
> - *предотвращение нежелательных и непредсказуемых эффектов;*
> - *формализацию процесса внесения изменений.*

В ряде источников можно увидеть аббревиатуру **SCCM** (Software Configuration and Change Management) — управление конфигурацией и изменениями ПО. При том, что в понимании SWEBOK и соответствующих стандартов, содержание SCM и SCCM тождественно, термин SCCM иногда используется для того, чтобы подчеркнуть принципиальную значимость управления изменениями как составной части конфигурационного управления.

> ***Управление конфигурацией*** *— защитная деятельность по координации различных версий и частей документации и программного кода, применяемая на всех этапах ЖЦ ПО.*

Управление конфигурацией начинается с началом проекта и заканчивается с прекращением использования ПО.

За время жизни программный код продукта претерпевает изменения двух категорий:

- добавление новых частей,
- подключение новых версий существующих частей.

Следует учитывать обе категории изменений.

Конфигурация *(Configuration) — совокупность функциональных, эксплуатационных и физических характеристик изделия.*

Конфигурация представляется древовидной (иерархической) структурой, элементами которой являются объекты управления конфигурацией.

Информация на выходе процесса разработки ПО условно делится на три категории:

1. Компьютерные *программы* (в виде исполняемых кодов).
2. *Документы*, описывающие программы (как для технического персонала, так и для пользователей).
3. *Структуры данных* (как внешних, так и внутренних).

Совокупность всех элементов информации, вырабатываемых как часть процесса разработки ПО, называют *конфигурацией ПО*.

С развитием процесса разработки ПО количество элементов конфигурации стремительно растёт.

Важным результатом SCM является тот факт, что потребителю поставляется не только само изделие, но и документированные доказательства того, что изделие и все его компоненты соответствуют заданным требованиям. Это, с одной стороны, служит основой гарантии качества, а с другой — защищает поставщика от необоснованных претензий.

Минимальная конфигурация ПО включает следующие базовые элементы:

1. Системная спецификация.
2. План программного проекта.
3. Спецификация требований к ПО, работающий или бумажный макет.
4. Предварительное руководство пользователя.
5. Спецификация проектирования.
6. Листинги исходных текстов программ.
7. План и методика тестирования. Тестовые варианты и полученные результаты.
8. Руководства по работе и инсталляции.
9. Исполняемый код программ.
10. Описание базы данных.
11. Руководство пользователя по настройке.
12. Документы сопровождения. Отчёты о проблемах ПО. Запросы сопровождения. Отчёты об изменениях.
13. Стандарты и методики разработки ПО.

Цели конфигурационного управления:

- *Контроль*: SCM позволяет отслеживать изменения в контролируемых объектах, обеспечивает соблюдение процесса разработки.
- *Управление*: SCM диктует процесс автоматической идентификации в ходе всего жизненного цикла ПО, обеспечивает простоту модификации и сопровождения ПО.
- *Экономия*: снижается риск потерь от ротации кадров в организации, предоставляет возможность сменить организацию-разработчика без перепроектирования.
- *Качество*: контролируется полнота и соответствие показателей ПО предъявляемым требованиям заказчика.

Задачи конфигурационного управления:

- идентификация объектов конфигурации;
- контроль изменений конфигурации: контроль над изменениями материалов (элементов, активов);
- учёт текущего состояния: состояние документов, состояние кода, состояние отдельных задач и всего проекта в целом;
- управление процессом разработки;
- управление сборкой;
- управление окружением;
- отслеживание задач и проблем (в частности, отслеживание ошибок).

2. Процедуры управления конфигурацией

К конфигурации ПО применяется техника управления конфигурацией.

Управление конфигурацией состоит в применении действий для управления изменениями в течение всего жизненного цикла ПО.

Изменение — непременный факт жизненного цикла ПО. Заказчики хотят изменить требования. Разработчики хотят модифицировать технический подход. Руководство хочет улучшить подход к проектированию. Почему так происходит? С течением времени все участники узнают что-то новое (о том, в чём они нуждаются; какой подход лучше; как увеличить прибыль).

Эти дополнительные знания и приводят к утверждению, которое плохо воспринимается практиками разработки: *Большинство изменений обоснованно!*

2.1. Идентификация объектов в конфигурации ПО

Для контроля и управления должны быть определены объекты конфигурации. Идентифицируются два типа объектов: **базисные объекты** и **составные объекты**.

> ***Базисный объект конфигурации*** *— артефакт, создаваемый в ходе анализа, проектирования, кодирования или тестирования.*

Например, базисный объект может быть секцией спецификации требований, частью проектной модели, исходным кодом для компонента или набором тестов для проверки программного кода.

> ***Составной объект конфигурации*** *является коллекцией базисных объектов и других составных объектов.*

Пример .1 (объекты конфигурации). `ПроектнаяСпецификация` является составным объектом. Он может рассматриваться как поименованный (идентифицированный) список указателей, которые определяют такие составные объекты, как `АрхитектурнаяМодель` и `МодельДанных`, а также базисные объекты `Компонент_N` и `ДиаграммаКлассов_N`.

Каждый объект имеет набор характеристик, которые определяют его уникальность: *имя*, *описание*, *список ресурсов*, *реализация*.

Имя объекта — строка символов.

Описание объекта — перечень элементов данных, определяющий:

1. тип элемента (элемент модели, программа, данные);
2. идентификатор проекта;
3. информацию изменения и/или версии.

> ***Ресурсы*** *— это элементы, которые предоставляются, обрабатываются, указываются или как-то иначе запрашиваются объектом.*

Например, в качестве ресурсов объектов могут рассматриваться типы данных, конкретные функции и даже имена переменных.

> ***Реализация*** *— указатель на элемент текста для базисного объекта и null для составного объекта.*

При идентификации объектов могут также рассматриваться отношения, которые существуют между именованными объектами.

Пример .2 (иерархия объектов конфигурации). Используя простую нотацию `ДиаграммаUseCase <part-of> МодельТребований; МодельТребований <part-of> СпецификацияТребований`, можно создать иерархию объектов конфигурации. Отношение `<part-of>` (часть) определяет иерархию объектов.

Во многих случаях существуют внутренние отношения, пересекающие ветви в иерархии объектов.

Пример .3 (пересекающие отношения). Такие пересекающие отношения представляются в следующем виде:

```
МодельДанных <interrelated> МодельПотоковДанных;
МодельДанных <interrelated> ТестовыйВариантКласса_М.
```

В первом случае рассматривается внутреннее отношение между составными объектами, во втором — между составным объектом `МодельДанных` и базисным объектом `ТестовыйВариантКласса_М`.

Схема идентификации должна выявлять программные объекты, эволюционирующие в процессе разработки.

2.2. Ревизия конфигурации

> ***Ревизия конфигурации*** *— процесс проверки того, что артефакт нижнего уровня соответствует всем требованиям артефакта верхнего уровня.*

2.3. Аудит конфигурации

> ***Аудит конфигурации*** *— процесс проверки того, что готовый продукт или его часть соответствуют документации.*

2.4. Учёт состояния конфигурации

> ***Учёт состояния конфигурации*** *— процесс подготовки отчётов о текущем состоянии продукта и состоянии утверждённых изменений.*

Отчётность о состоянии конфигурации позволяет зафиксировать ответы на следующие вопросы:

- Что случилось?
- Кто это сделал?
- Когда это произошло?
- Что ещё будет затронуто?

Отчётность гарантирует, что разработчики не попадут в ситуацию, когда «левая рука не знает, что делает правая».

2.5. Контроль конфигурации

> ***Контроль конфигурации (управление изменениями** в ПО) — процесс, при котором все предлагаемые изменения продукта проходят одобрение специальной группы (или отдельного человека).*

Одна из функций такой группы — контроль актуальности всех имеющихся документов, а также контроль того что все изменения сначала вносятся в документацию, а уже затем в объект изменения.

Эта процедура включает следующие действия:

1. Идентификация изменения.
2. Контроль изменения.
3. Гарантия правильной реализации изменения.
4. Формирование сообщения об изменениях.

При разработке больших систем неконтролируемые изменения быстро приводят к хаосу. Контроль изменений обеспечивается механизмом, включающим человеческие действия и автоматические средства.

Пример .4 (процесс контроля и изменения конфигурации). Допустим, поступил запрос на изменение конфигурации.

1. Запрос оценивается по следующим параметрам: техническое качество, эффект, воздействие на другие объекты конфигурации и функции системы, стоимость. Формируется донесение об изменении.
2. Специалист по контролю принимает решение: отвергнуть или утвердить. В случае отказа оповещается инициатор изменения и прекращается процесс. В случае утверждения создаётся заказ на изменение (описываются само изменение, ограничения, критерии проверки).
3. Для обработки заказа объект конфигурации выбирается из базы данных.
4. Выполняется «выходная» проверка объекта.

Пример .5 (процесс контроля и изменения конфигурации).

5. Выполняется изменение объекта.
6. Проверяется правильность и качество проведённого изменения.
7. Выполняется «входная» проверка объекта, после чего он заносится в базу данных.
8. После утверждения изменения обновлённый объект включается в следующую реализацию.
9. Перестраивается соответствующая версия ПО.

10. Проверяются изменения всех элементов конфигурации.

11. Изменения включаются в новую версию ИС.

12. Выпускается новая версия ИС.

2.6. Контроль версий

Контроль версий объединяет процедуры и средства для управления различными версиями объектов конфигурации, которые создаются в ходе разработки.

Система контроля версий обычно состоит из следующих элементов:

1. репозиторий проекта, хранящий все значимые объекты конфигурации;

2. средство управления версиями, сохраняющее все версии объектов конфигурации (или создающее любую версию на основе различий предыдущих версий);

3. устройство генерации, позволяющее собирать все значимые объекты конфигурации и создавать определённую версию ПО.

Некоторые системы контроля версий определяют набор изменений — коллекцию всех изменений (по отношению к базовой конфигурации), которые требуются для создания определённой версии ПО.

Кроме того, системы контроля версий и контроля изменений часто предлагают возможности отслеживания результатов (часто называемые отслеживанием ошибок), которые позволяют записать и отследить состояние всех нереализованных результатов, связанных с каждым объектом конфигурации.

3. План управления конфигурацией

Пример типового плана управления конфигурацией:

1. Введение.

2. Управление конфигурациями.

   2.1. Организация.

   2.2. Ответственность за управление конфигурациями.

   2.3. Применяемые методики, директивы, процедуры.

3. Виды деятельности.

   3.1. Определение элементов конфигурации.

      3.1.1. Именование элементов конфигурации.

      3.1.2. Получение элементов конфигурации.

   3.2. Контроль конфигурации.

      3.2.1. Запрос на изменения.

      3.2.2. Оценка изменений.

      3.2.3. Одобрение или неодобрение изменений.

3.2.4. Реализация изменений.

3.3. Определение статуса конфигурации.

3.4. Аудиты и проверки конфигурации.

3.5. Контроль интерфейса.

3.6. Контроль поставщиков и субподрядчиков.

4. Расписание.

5. Ресурсы.

6. Сопровождение.

В этом плане приводится описание всех узловых моментов процесса управления конфигурацией, фиксируются временные ограничения, периодичность выполнения действий.

Глава 4

Организация процесса разработки информационных систем

1. Понятие жизненного цикла ИС

1.1. Классификация процессов жизненного цикла

В 1987 году был создан объединённый технический комитет № 1, — ISO/IEC JTC 1 (ISO/IEC Joint Technical Committee 1). Это подразделение Международной организации по стандартизации (International Organization for Standardization, ISO) и Международной электротехнической комиссии (International Electrotechnical Commission, IEC), которое занимается всеми вопросами, связанными со стандартами в области информационных технологий. В 1989 году этот комитет инициировал разработку стандарта ISO/IEC 12207. Соответствующий стандарт впервые был опубликован 1-го августа 1995 года под заголовком «Software Life Cycle Processes» — «Процессы жизненного цикла программного обеспечения».

Государственный стандарт РФ, созданный на его основе, получил название ГОСТ Р ИСО/МЭК 12207-99 «Информационная технология. Процессы жизненного цикла программных средств». Он был принят в 1999 г. Этот стандарт предназначен для использования как в случае отдельно поставляемых программных средств, так и для программных средств, встраиваемых или интегрируемых в общую систему.

Цель разработки стандарта ISO/IEC 12207 была определена как создание общего фреймворка по организации жизненного цикла программного обеспечения для формирования общего понимания жизненного цикла ИС всеми заинтересованными сторонами и участниками процесса разработки, приобретения, поставки, эксплуатации, поддержки и сопровождения программных систем, а также возможности управления, контроля и совершенствования процессов жизненного цикла.

Данный стандарт устанавливает, используя чётко определённую терминологию, общую структуру процессов жизненного цикла программных средств, на которую можно ориентироваться в программной индустрии. Он определяет процессы, работы и задачи, которые используются: при приобретении системы, содержащей программные средства, или отдельно поставляемого программного продукта; при оказании программной услуги, а также при поставке, разработке, эксплуатации и сопровож-

дении программных продуктов.

В соответствии с базовым международным стандартом ISO/IEC 12207 [**?**] все процессы ЖЦ ИС делятся на три группы:

1. **Основные процессы** (выполняются под управлением одного из участников):
   - заказ/приобретение (acqusition), определяет работы заказчика;
   - поставка (supply), определяет работы поставщика;
   - разработка (development), определяет работы разработчика;
   - эксплуатация (operation), определяет работы оператора;
   - сопровождение (maintenance), определяет работы сопровождающей организации.

2. **Вспомогательные процессы** (для поддержки выполнения основных процессов, обеспечения качества проекта):
   - документирование (documentation);
   - управление конфигурацией (configuration management);

     > ***Конфигурация ИС*** *— совокупность функциональных, эксплуатационных и физических характеристик, установленных в технической документации и реализованных в ИС.*

     Управление конфигурацией позволяет организовать, систематически учитывать и контролировать внесение изменений в ИС на всех стадиях жизненного цикла ИС.
   - обеспечение качества (quality assurance); Этот процесс определяет работы, направленные на то, чтобы ИС соответствовала установленным требованиям и создавалась в рамках утверждённых планов. В качестве методов обеспечения качества могут использоваться совместные оценки, аудиторские проверки, верификация и аттестация.
   - верификация (verification);

     > ***Верификация*** *— подтверждение реализации конкретных требований к ИС.*

   - аттестация (validation);

     > ***Аттестация*** *— подтверждение полноты реализации всех требований к ИС.*

   - совместный анализ (joint review); Процесс может использоваться двумя и более сторонами на совместном совещании.
   - аудит (audit);

Аудит *— процедура независимой оценки соответствия требованиям, планам, договору для выявления реального положения дел.*

- решение проблем (problem resolution). Этот процесс определяет работы, связанные с анализом и устранением проблем, которые были обнаружены во время разработки, эксплуатации, сопровождения и др. процессов.

3. **Организационные процессы** (определяют действия и задачи, выполняемые как заказчиком, так и разработчиком проекта для управления своими процессами):

   - управление (management);
   - создание инфраструктуры (infrastructure);
   - усовершенствование (improvement);
   - обучение (training).

Ниже приведены ориентировочные описания основных процессов ЖЦ.

2. Процессы жизненного цикла ИС

Таблица 4.1.

Содержание основных процессов ЖЦ ПО систем (ISO/IEC 12207)

Действия	**Вход**	**Результат**
1. **Приобретение** (заказчик)		
Инициирование	Решение о начале работ по внедрению системы	Технико-экономическое обоснование внедрения системы
Подготовка заявки на подряд	Результаты обследования деятельности заказчика	Техническое задание на систему (ТЗ)
Подготовка договора	Результаты анализа рынка систем / тендера	Договор на поставку / разработку
Надзор за поставщиком	План поставки / разработки	Акты приёмки этапов работы
Приёмка системы	Комплексный тест системы	Акт приёмно-сдаточных испытаний

Действия	**Вход**	**Результат**
2. **Поставка** (поставщик)		
Инициирование	Техническое задание на систему (ТЗ)	Решение об участии в разработке
Ответ на заявку	Решение руководства об участии в разработке	Коммерческие предложения / конкурсная заявка
Подготовка договора	Результаты тендера	Договор на поставку / разработку
Планирование исполнения	ТЗ	План управления проектом
Выполнение и контроль	ТЗ, план управления проектом	Разработанная система и документация
Проверка и оценка;	План управления проектом, разработанная система и документация	Разработанная система и документация, отчёты о проверках
Поставка системы	Разработанная система и документация	Акт приёмно-сдаточных испытаний

Действия	**Вход**	**Результат**
3. **Разработка** (разработчик)		
Подготовка	ТЗ; договор; стандарты разработки	Модель ЖЦ; план работ
Анализ требований к системе	ТЗ; модель ЖЦ	План работ; состав подсистем; требования пользователя; квалификационные требования; спецификации требования к компонентам системы
Проектирование системной архитектуры	ТЗ; состав подсистем	Архитектура системы; спецификации требования к объектам архитектуры; компоненты оборудования

Действия	**Вход**	**Результат**
3. **Разработка** (разработчик)		
Анализ требований к программным средствам	Архитектура системы	Требования к внешним интерфейсам ПО ИС; функциональные, технические, квалификационные и др. требования
Проектирование программной архитектуры	Спецификации требования к компонентам ИС	Архитектура ПО системы, состав компонентов ИС, эскизные проекты интерфейсов компонентов системы, предварительные версии документации пользователя и требований к испытаниям системы

Действия	**Вход**	**Результат**
3. **Разработка** (разработчик)		
Техническое проектирование ИС	Материалы проектирования программной архитектуры	Технические проекты компонентов ИС, интерфейсов, БД; требования к испытанию; документация пользователя; план интеграции ИС
Программирование и тестирование ПО ИС	Материалы технического проектирования ПО ИС	БД; модули ПО ИС; тестовые данные и процедуры испытаний, акты тестирования
Сборка ПО ИС	План интеграции ПО ИС; тесты	ПО ИС; оценка соответствия ПО ИС, БД, технического комплекса и документации требованиям

Действия	**Вход**	**Результат**
3. **Разработка** (разработчик)		
Квалификационные испытания ПО ИС	ПО системы; тесты	Квалификационные испытания; аудиторская проверка; оценка соответствия системы требованиям заказчика
Сборка системы	Архитектура системы; квалификационные требования	ИС; состав испытаний и контрольных примеров
Квалификационные испытания системы	ИС; тесты	Квалификационные испытания; аудиторская проверка; оценка соответствия системы ожидаемым результатам

Действия	**Вход**	**Результат**
3. **Разработка** (разработчик)		
Ввод в действие ПО ИС	Договор; ресурсы для системы	План по вводу в действие ПО системы в среде эксплуатации; работоспособное ПО системы
Обеспечение приёмки ПО системы	ПО системы; результаты тестирований	Обученный персонал

Действия	**Вход**	**Результат**
4. **Процесс эксплуатации** (оператор)		
Подготовка процесса	Набор стандартов по эксплуатации; документация на ИС	План эксплуатации; процедуры для получения и документирования сведений о возникающих проблемах, решения и контроля проблем и обеспечения обратной связи с пользователем; процедуры для тестирования ПО системы в эксплуатационной среде, ввода сообщений о проблемах и предложений об изменениях в процесс сопровождения

Действия	**Вход**	**Результат**
4. **Процесс эксплуатации** (оператор)		
Эксплуатационные испытания	План эксплуатации ПО системы; тесты	Эксплуатационные испытания ПО системы
Эксплуатация системы	Документация пользователя	Эксплуатация
Поддержка пользователя	ИС; тесты	Помощь и консультации пользователям; отправка запросов пользователей в процесс сопровождения

Действия	**Вход**	**Результат**
5. **Процесс сопровождения** (персонал сопровождения)		
Подготовка процесса	Документация на систему	Планы и процедуры для проведения работ и задач процесса сопровождения; процедуры для получения, документирования и контроля сообщений о возникающих проблемах и заявок на внесение изменений от пользователей, а также обеспечения обратной связи с пользователями; процесс управления конфигурацией

Действия	**Вход**	**Результат**
5. **Процесс сопровождения** (персонал сопровождения)		
Анализ проблем и изменений	Материалы подготовки процесса сопровождения	Анализ сообщений о проблеме, заявок на внесение изменений в систему; верификация проблем; варианты реализации изменения (на основе анализа)
Внесение изменений	Материалы анализа проблем и изменений	Анализ и выявление документов и программных модулей для изменения; тесты, результаты испытаний
Проверка и приёмка при сопровождении	Результаты внесения изменений	Проверка внесённых изменений

Действия	**Вход**	**Результат**
5. **Процесс сопровождения** (персонал сопровождения)		
Перенос	Система (переносится в новую эксплуатационную среду); договор	План переноса объекта, уведомление пользователей о планах и работах по переносу; обучение персонала; архив прежнего ПО ИС и документации; результаты анализа переноса
Снятие с эксплуатации	Результаты внесения изменений	План снятия с эксплуатации; уведомление пользователей о планах и работах по снятию с эксплуатации

Для поддержки практического применения стандарта ISO/IEC 12207 разработан ряд технологических документов:

- Руководство для ISO/IEC 12207 (ISO/IEC TR 15271:1998 Information technology — Guide for ISO/IEC 12207);
- Руководство по применению ISO/IEC 12207 к управлению проектами (ISO/IEC TR 16326:1999 Software engineering — Guide for the application of ISO/IEC 12207 to project management).
- IEEE/EIA 12207.0-1996 — адаптация IEEE стандарта ISO/IEC 12207, она содержит:
  - ISO/IEC 12207 с добавлениями (улучшенный подход к совместимости, назначение процессов жизненного цикла, назначение данных жизненного цикла, список исправлений).
  - IEEE/EIA 12207.1-1997, IEEE/EIA Руководство по информационной технологии. Процессы жизненного цикла программного обеспечения. Данные жизненного цикла.
  - IEEE/EIA 12207.2-1997, IEEE/EIA Руководство по информационной технологии. Процессы жизненного цикла программного обеспечения. Замечания по реализации.

  На их основе создан ГОСТ Р ИСО/МЭК ТО 15271-2002 Руководство по применению ISO/IEC 12207 (Процессы жизненного цикла программных систем).

Позднее был разработан и в 2002 г. опубликован стандарт на процессы жизненного цикла систем — ISO/IEC 15288 (System engineering — System life cycle processes), национальный стандарт — ГОСТ Р ИСО/МЭК 15288-2005 (Системная инженерия — процессы жизненного цикла систем). Последняя версия ISO/IEC 15288 утверждена в 2008 г.

К разработке стандарта были привлечены специалисты различных областей: системной инженерии, программирования, управления качеством, человеческими ресурсами, безопасностью и пр. Был учтён практический опыт создания систем в правительственных, коммерческих, военных и академических организациях. Стандарт применим для широкого класса систем, но его основное предназначение — поддержка создания компьютеризированных систем.

2.1. Структура жизненного цикла

Согласно стандарту ГОСТ Р ИСО/МЭК 15288-2005 в структуру ЖЦ следует включать следующие группы процессов:

1. **Процессы соглашения:**
   - приобретение (продукции или получение услуг);
   - поставка (продукции или оказания услуг).

2. Процессы предприятия:

- управление средой предприятия;
- управления инвестициями;
- управление ЖЦ системы;
- управление ресурсами;
- управление качеством.

3. Процессы проекта:

- планирование проекта;
- оценка проекта;
- контроль проекта;
- принятие решений;
- управление рисками;
- управление конфигурацией;
- управление информацией.

4. Технические процессы:

- определение требований;
- анализ требований;
- проектирование архитектуры;
- реализация элементов системы;
- комплексирование (сборка системы согласно архитектурному проекту);
- верификация;
- передача;
- валидация (аттестация);
- функционирование (эксплуатация);
- сопровождение (техническое обслуживание);
- утилизация (изъятие и списание).

Стадии создания системы, предусмотренные в стандарте ISO/IEC 15288, несколько отличаются от рассмотренных выше. Перечень стадий и основные результаты, которые должны быть достигнуты к моменту их завершения, приведены ниже.

Стадии создания систем (ISO/IEC 15288):

1. *Стадия замысла* — осознание потребности в системе (или её модификации), *подготовка* и *планирование* проекта.
2. *Разработка* — *моделирование* системы (включает *анализ* потребностей, *проектирование* системы).
3. *Реализация* — *конструирование* системы (включает *кодирование* и *тестирование*).
4. *Эксплуатация* — *развёртывание* (*поставка*, *ввод в эксплуатацию*) и *использование* системы.
5. *Поддержка* — *сопровождение* (обеспечение функционирования системы).
6. *Снятие с эксплуатации* — прекращение использования, *демонтаж*, *архивирование* системы.

3. Модели жизненного цикла ИС

> ***Модель жизненного цикла ИС*** *— структура, содержащая процессы, действия и задачи, которые осуществляются в ходе разработки, функционирования и сопровождения ИС в течение всей жизни системы, от определения требований до завершения её использования.*

Простыми словами: **модель жизненного цикла ИС** отражает различные состояния системы, начиная с момента возникновения необходимости в данной системе и заканчивая моментом её полного выхода из употребления.

В настоящее время широко известны:

- *каскадная (водопадная) модель*,
- *инкрементная модель*,
- *спиральная (итеративная) модель*.

Существуют 3 стратегии разработки ИС:

Однократный проход (водопадная стратегия) — линейная последовательность этапов конструирования.

Инкрементная стратегия — в начале процесса определяются все пользовательские и системные требования, оставшаяся часть конструирования выполняется в виде последовательности версий. Первая версия реализует часть запланированных возможностей, следующая версия реализует дополнительные возможности и т. д., пока не будет получена полная система.

Эволюционная стратегия — система также строится в виде последовательности версий, но в начале процесса определены не все требования. Требования уточняются в результате разработки версий.

Таблица 4.2.

Характеристики стратегий конструирования ИС в соответствии с требованиями стандарта IEEE/EIA 12207.2

Стратегия разработки	**В начале процесса определены все требования?**	**Множество циклов конструирования?**	**Промежуточные версии ИС распространяются?**
Однократный проход	Да	Нет	Нет
Инкрементная (запланированное улучшение продукта)	Да	Да	Может быть
Эволюционная	Нет	Да	Да

3.1. Каскадная модель

Каскадная модель (или водопадная, классическая) характеризуется последовательным выполнением всех этапов проекта в строго фиксированном порядке. Переход на следующий этап означает полное завершение работ на предыдущем этапе. Характерна для периода 1970–1985 гг.

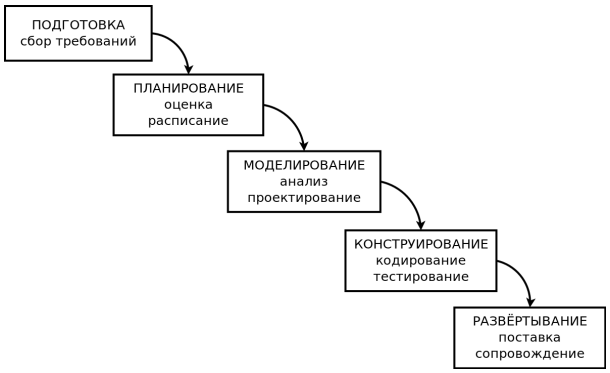


Рис. 4.1. Каскадная модель ЖЦ системы

В ранних проектах, когда каждое приложение представляло собой единый, функционально и информационно независимый блок, данный способ оказался вполне эффективным.

Положительные стороны применения каскадного подхода:

- на каждом этапе формируется законченный набор проектной документации, отвечающий критериям полноты и согласованности;
- выполняемые в логической последовательности этапы работ позволяют планировать сроки завершения всех работ и соответствующие затраты.

3.2. Каскадная модель с промежуточным контролем

Каскадный подход хорошо зарекомендовал себя при построении относительно простых систем, когда в самом начале разработки можно достаточно точно и полно сформулировать все требования к системе. Основным недостатком каскадного подхода является то, что реальный процесс создания системы никогда полностью не укладывается в его жёсткую схему, постоянно возникает потребность в возврате к предыдущим этапам и уточнении или пересмотре ранее принятых решений.

Реальный процесс создания систем оказывается соответствующим *каскадной модели с промежуточным контролем*. Эта модель (рис. 4.2) характеризуется *итеративной* разработкой систем с циклами обратной связи между этапами. Межэтапные корректировки позволяют учитывать реально существующее взаимовлияние результатов разработки на различных этапах. Время жизни каждого из этапов растягивается на весь период разработки.

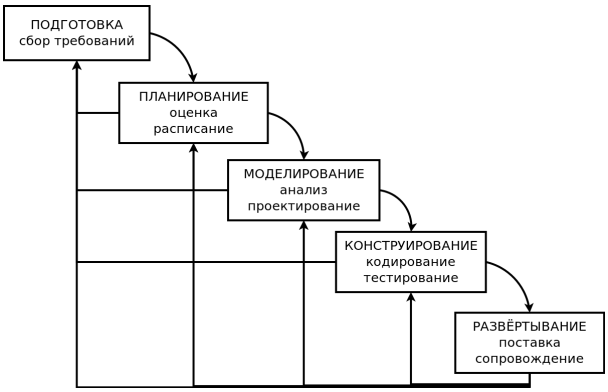


Рис. 4.2. Каскадная модель с промежуточным контролем

На практике эта схема не получила большого распространения, т. к. она не позволяет оперативно учитывать возникающие изменения и уточнения требований к системе. Согласование результатов разработки с пользователями производится только в точках, планируемых после завершения каждого этапа работ, а общие требования к системам зафиксированы в виде технического задания на всё время её создания. Таким образом, пользователи зачастую получают систему, не удовлетворяющую их реальным потребностям.

3.3. Спиральная (итеративная) модель

Для преодоления перечисленных проблем была предложена *спиральная модель ЖЦ*, характерная для периода после 1986 г.

> Спиральная модель *(рис. 4.3) характеризуется тем, что на каждом витке спирали выполняется создание очередной версии продукта, уточняются требования проекта, определяется его качество и планируются работы следующего витка.*

Рис. 4.3. Спиральная модель ЖЦ

Особое внимание уделяется начальным этапам разработки — анализу и проектированию. На этих этапах реализуемость технических решений и степень удовлетворения потребностей заказчика проверяется и обосновывается посредством создания моделей и **прототипов**.

Каждый виток спирали соответствует созданию работоспособного фрагмента или версии системы, что позволяет уточнить требования, цели и характеристики проекта, определить качество разработки, спланировать работы следующего витка спирали. Таким образом углубляются и последовательно конкретизируются детали проекта и в результате получается вариант, который действительно удовлетворяет требованиям заказчика.

Итеративная разработка отражает объективно существующий спиральный цикл создания сложных систем. Она позволяет переходить на следующий этап, не дожидаясь полного завершения работы на текущем и решить главную задачу — как можно быстрее показать пользователям системы работоспособный продукт, тем самым активизируя процесс уточнения и дополнения требований.

Основная *проблема* спиральной модели — определение момента перехода на следующий этап. Для её решения вводятся временные ограничения на каждый из этапов жизненного цикла, и переход осуществляется в соответствии с планом, даже если не вся запланированная работа закончена. Планирование производится на основе статистических данных, полученных в предыдущих проектах, и личного опыта разработчиков.

Очевидно, что если требования к информационной системе меняются в ходе реализации проекта, а качество документов оказывается невысоким (требования неполны и/или противоречивы), то в действительности использование каскадной модели создаёт лишь иллюзию определённости и на деле увеличивает риски, уменьшая лишь ответственность участников проекта. При формальном подходе менеджер проекта реализует только те требования, которые содержатся в спецификации, опирается на документ, а не на реальные потребности бизнеса.

Есть два основных *типа контрактов* на разработку ИС. Первый тип предполагает выполнение определённого объёма работ за определённую сумму в определённые сроки (*fixed price*). Второй тип предполагает повремённую оплату работы (*time work*). Выбор того или иного типа контракта зависит от степени определённости

задачи.

Каскадная модель с определёнными этапами и их результатами лучше приспособлена для заключения контракта с оплатой по результатам работы, именно этот тип контрактов позволяет получить полную оценку стоимости проекта до его завершения. Разработка и внедрение интегрированной информационной системы требует существенных финансовых затрат, поэтому используются контракты с фиксированной ценой, а значит и каскадная модель разработки и внедрения.

Спиральная модель чаще применяется при разработке информационной системы силами собственного ИТ отдела предприятия. Заключение контракта на небольшую систему, с относительно небольшим весом в структуре затрат предприятия, более вероятно с повремённой оплатой.

Проблемы внедрения при использовании спиральной модели. В некоторых областях спиральная модель не может применяться, поскольку невозможно использование/тестирование продукта, обладающего *неполной функциональностью* (например, военные разработки, атомная энергетика и т. д.). Поэтапное итерационное внедрение информационной системы для бизнеса возможно, но сопряжено с организационными сложностями (перенос данных, интеграция систем, изменение бизнес-процессов, учётной политики, обучение пользователей). Трудозатраты при поэтапном итерационном внедрении оказываются значительно выше, а управление проектом требует настоящего искусства. Предвидя указанные сложности, заказчики выбирают каскадную модель, чтобы «внедрять систему один раз».

3.4. Компонентно-ориентированная модель

Компонентно-ориентированная модель является развитием спиральной модели и тоже основывается на эволюционной стратегии конструирования. В этой модели конкретизируется содержание квадранта конструирования — оно отражает тот факт, что в современных условиях новая разработка должна основываться на повторном использовании существующих программных компонентов (рис. 4.4).

Программные компоненты, созданные в реализованных программных проектах, хранятся в библиотеке. В новом программном проекте, исходя из требований заказчика, выявляются кандидаты в компоненты. Далее проверяется наличие этих кандидатов в библиотеке. Если они найдены, то компоненты извлекаются из библиотеки и используются повторно. В противном случае создаются новые компоненты, они применяются в проекте и включаются в библиотеку.

Достоинства компонентно-ориентированной модели:

1. уменьшает на 30% время разработки программного продукта;
2. уменьшает стоимость программной разработки до 70%;
3. увеличивает в полтора раза производительность разработки.

3.5. Инкрементная модель

Инкрементная модель основана на инкрементной стратегии конструирования (рис. 4.5).

Рис. 4.4. Компонентно-ориентированная модель ЖЦ

Она объединяет элементы последовательной водопадной модели с итерационной философией **макетирования**.

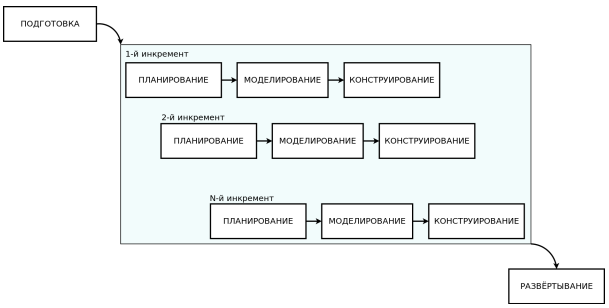


Рис. 4.5. Инкрементная модель ЖЦ

Каждая линейная последовательность здесь вырабатывает поставляемый инкремент ИС. Например, ИС для обработки слов в 1-м инкременте реализует функции базовой обработки файлов, функции редактирования и документирования; во 2-м инкременте — более сложные возможности редактирования и документирования; в 3-м инкременте — проверку орфографии и грамматики; в 4-м инкременте — возможности компоновки страницы. Первый инкремент приводит к получению базового продукта, реализующего базовые требования (правда, многие вспомогательные требования остаются нереализованными). План следующего инкремента предусматривает модификацию базового продукта, обеспечивающую дополнительные характеристики и функциональность.

По своей природе инкрементный процесс интегративен, но, в отличие от **макетирования**, инкрементная модель может обеспечить на каждом инкременте работающий продукт.

Глава 5

Методологии и стандарты разработки информационных систем

1. Стандарты на процессы и организацию жизненного цикла

Каждая из стадий создания системы предусматривает выполнение определённого объёма работ, которые представляются в виде *процессов ЖЦ*. *Процесс* определяется как совокупность взаимосвязанных действий, преобразующих входные данные в выходные. Описание каждого процесса включает в себя перечень решаемых задач, исходных данных и результатов.

Существует ряд *стандартов*, регламентирующих ЖЦ ИС, а в некоторых случаях и процессы разработки. Среди наиболее известных стандартов можно выделить следующие:

ГОСТ 34.601-90 — распространяется на автоматизированные системы и устанавливает стадии и этапы их создания, описывает содержание работ на каждом этапе, которые в большей степени соответствуют каскадной модели ЖЦ.

ГОСТ Р ИСО/МЭК 12207-99 — стандарт на процессы и организацию ЖЦ. Распространяется на все виды заказного ПО. Стандарт не содержит описания фаз, стадий и этапов.

Custom Development Method (CDM) — методика Oracle по разработке прикладных информационных систем — технологический материал, детализированный до уровня заготовок проектных документов, рассчитанных на использование в проектах с применением Oracle. **Применяется для каскадной модели ЖЦ** (предусмотрены все работы/задачи и этапы), а также **для технологий «быстрой разработки»** (Fast Track) или **«облегчённого подхода»**, рекомендуемых в случае малых проектов.

1.1. Microsoft Solution Framework (MSF)

MSF является *итерационной* (сочетает в себе свойства каскадной и спиральной моделей ЖЦ), предполагает использование *объектно-ориентированного моделирования*. MSF опирается на практический опыт Microsoft и описывает управление людьми и рабочими процессами в ходе разработки ПО.

Первая версия MSF появилась в 1994 году.

Модель процессов MSF (MSF process model) представляет общую методологию разработки и внедрения ИТ решений. Особенность этой модели состоит в том, что благодаря своей гибкости и отсутствию жёстко навязываемых процедур она может быть применена при разработке весьма широкого круга ИТ проектов.

MSF включает фазы:

1. *выработка концепции* (*envisioning*),
2. *планирование* (*planning*),
3. *разработка* (*developing*),
4. *стабилизация* (*stabilizing*),
5. *внедрение* (*deploying*).

Процесс MSF ориентирован на «вехи» (milestones) — ключевые точки проекта, характеризующие достижение в его рамках какого-либо существенного (промежуточного либо конечного) результата. Этот результат может быть оценён и проанализирован, что подразумевает ответы на вопросы: «Пришла ли проектная группа к однозначному пониманию целей и рамок проекта?», «В достаточной ли степени готов план действий?», «Соответствует ли продукт утверждённой спецификации?», «Удовлетворяет ли решение нужды заказчика?» и т. д.

Модель процессов MSF учитывает постоянные изменения проектных требований. Она исходит из того, что разработка решения должна состоять из коротких циклов, создающих поступательное движение от простейших версий решения к его окончательному виду.

Особенностями модели процессов MSF являются:

- подход, основанный на фазах и вехах;
- итеративный подход;
- интегрированный подход к созданию и внедрению решений.

В рамках MSF программный код, документация, дизайн, планы и другие рабочие материалы создаются, как правило, итеративными методами. MSF рекомендует начинать разработку решения с построения, тестирования и внедрения его базовой функциональности. Затем к решению добавляются всё новые и новые возможности. Такая стратегия именуется **стратегией версионирования**. Несмотря на то, что для малых проектов может быть достаточным выпуск одной версии, рекомендуется не упускать возможности создания для одного решения ряда версий. С созданием новых версий эволюционирует функциональность решения.

Итеративный подход к процессу разработки требует использования гибкого способа ведения документации. «Живые» документы (living documents) должны изменяться по мере эволюции проекта вместе с изменениями требований к конечному продукту. В рамках MSF предлагается ряд шаблонов стандартных документов, которые являются артефактами каждой фазы разработки продукта и могут быть использованы для планирования и контроля процесса разработки.

Дисциплина управления рисками в MSF (MSF risk management discipline) отстаивает превентивный подход к работе с рисками в условиях такой неопределённости, непрерывное оценивание рисков и использование информации о рисках в рамках процесса принятия решений на протяжении всего жизненного цикла проекта. Данная дисциплина предлагает принципы, идеи и рекомендации, подкреплённые описанием пошагового процесса для успешного активного управления рисками. Этот процесс включает в себя выявление и анализ рисков; планирование и реализацию стратегий по их профилактике и смягчению возможных последствий; отслеживание состояния рисков и извлечение уроков из обретённого опыта. Девиз MSF — *Мы не боремся с рисками — мы ими управляем.*

В MSF нет роли «менеджер проекта». Деятельность по управлению проектом распределяется между лидерами групп и ролевым кластером «Управление программой».

Управление подготовкой — это также одна из ключевых дисциплин MSF. Она посвящена управлению знаниями, профессиональными умениями и способностями, необходимыми для планирования, создания и сопровождения успешных решений. Дисциплина управления подготовкой MSF описывает фундаментальные принципы MSF и даёт рекомендации по применению превентивного подхода к управлению знаниями на протяжении всего жизненного цикла информационных технологий. Эта дисциплина также рассматривает планирование процесса управления подготовкой. Будучи подкреплённой испытанными практическими методиками, дисциплина управления подготовкой предоставляет проектным группам и отдельным специалистам базу для осуществления этого процесса.

Текущая версия — MSF 4.0 принята в 2005 году. В последней версии произошло разделение методологии на два направления:

- **MSF for Agile Software Development**
- **MSF for CMMI Process Improvement**.

Кроме этого, появилась роль архитектора и поддержка методологии в инструменте — Microsoft Visual Studio Team System.

Следует отметить, что MSF не навязывает использование других продуктов Microsoft. Например, для организации процесса производства ИС можно использовать MSF 3.0 и при этом применять инструменты Borland, хотя MSF 4.0 жёстко привязана к Team System — новому инструментальному средству Microsoft для поддержки командной работы над проектом.

1.2. Rational Unified Process (RUP)

Rational Unified Process (RUP) предлагает итеративную модель разработки, включающую четыре стадии: начало, проектирование, построение и внедрение. Каждая

стадия может быть разбита на этапы (итерации), на выходе которых получается версия для внутреннего или внешнего использования. Прохождение через четыре основные стадии называется циклом разработки, каждый цикл завершается генерацией версии системы. Если после этого работа над проектом не прекращается, то полученный продукт продолжает развиваться и снова минует те же стадии. *Суть работы в рамках RUP — это создание и сопровождение моделей на базе* ***UML***.

В основе RUP лежат следующие принципы:

- Ранняя идентификация и непрерывное (до окончания проекта) устранение основных рисков.
- Акцент на выполнении требований заказчиков к функционалу (построение и анализ модели прецедентов (вариантов использования)).
- Ожидание изменений в требованиях, проектных решениях и реализации в процессе разработки.
- Компонентная архитектура, реализуемая и тестируемая на ранних стадиях проекта.
- Постоянное обеспечение качества на всех этапах разработки проекта (продукта).
- Работа над проектом в сплочённой команде, ключевая роль в которой принадлежит архитекторам.

RUP использует итеративную модель разработки. В конце каждой итерации (в идеале продолжающейся от 2 до 6 недель) проектная команда должна достичь запланированных на данную итерацию целей, создать или доработать проектные артефакты и получить промежуточную, но функциональную версию конечного продукта. Итеративная разработка позволяет быстро реагировать на меняющиеся требования, обнаруживать и устранять риски на ранних стадиях проекта, а также эффективно контролировать качество создаваемого продукта.

Полный жизненный цикл разработки продукта состоит из четырёх стадий, каждая из которых включает в себя одну или несколько итераций:

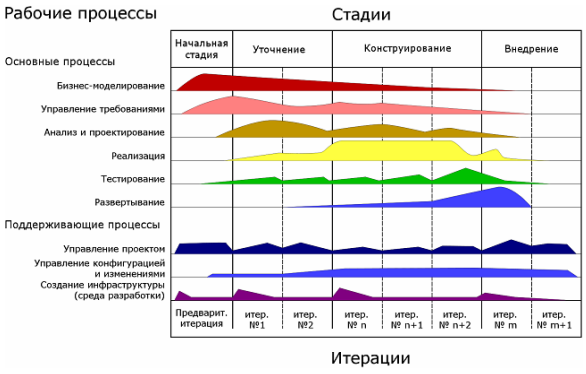


Рис. 5.1. Графическое представление процесса разработки по RUP

Начало (Inception) На этом этапе:

- Формируются *видение* и *границы проекта*.
- Создаётся *экономическое обоснование* (business case).
- Определяются *основные требования*, *ограничения* и *ключевая функциональность* продукта.
- Создаётся базовая версия *модели прецедентов*.
- *Оцениваются риски*.

Если по завершению запланированного времени цели стадии не достигнуты, то выполняется следующая итерация этой стадии. При завершении этой стадии оценивается достижение *вехи целей жизненного цикла* (*Lifecycle Objective Milestone*), которое предполагает соглашение заинтересованных сторон о продолжении проекта.

Проектирование (Elaboration) На этой стадии производится *анализ предметной области* и *построение исполняемой архитектуры*. Она включает в себя следующие работы:

- *Документирование требований* (включая детальное описание для большинства прецедентов).
- Проектирование, реализацию и тестирование *исполняемой архитектуры*.
- Обновление *экономического обоснования*, уточнение *оценок сроков* и *стоимости*.
- Снижение *основных рисков*.

Если по завершению запланированного времени цели стадии не достигнуты, то выполняется следующая итерация этой стадии. Успешное выполнение этой стадии означает достижение *вехи архитектуры жизненного цикла* (*Lifecycle Architecture Milestone*).

Построение (Construction) Во время этой стадии происходит *реализация* большей части *функциональности продукта*. Если по завершению запланированного времени цели стадии не достигнуты, то выполняется следующая итерация этой стадии. Стадия завершается первым внешним релизом системы и *вехой начальной функциональной готовности* (*Initial Operational Capability*).

Внедрение (Transition) Во время этой стадии создаётся *финальная версия продукта* и она передаётся от разработчика к заказчику. Продукт включает в себя *программу бета-тестирования*, *обучение пользователей*, а также *определение качества продукта*. В случае, если качество не соответствует ожиданиям пользователей или критериям, установленным в стадии «Начало», стадия «Внедрение» повторяется снова. Выполнение всех целей означает достижение *вехи готового продукта* (*Product Release*) и завершение полного цикла разработки.

2. Гибкая методология разработки

***Гибкая методология разработки** (**Agile software development**, **agile-методы**) — серия подходов к разработке программного обеспечения, ориентированных на использование итеративной разработки и динамическое формирование требований и обеспечение их реализации в результате постоянного взаимодействия внутри самоорганизующихся рабочих групп, состоящих из специалистов различного профиля.*

Большинство гибких методологий нацелены на минимизацию рисков путём сведения разработки к серии коротких циклов, называемых итерациями, которые обычно длятся две-три недели.

Каждая итерация сама по себе выглядит как программный проект в миниатюре и включает все задачи, необходимые для выдачи мини-прироста по функциональности: планирование, анализ требований, проектирование, программирование, тестирование и документирование. Хотя отдельная итерация, как правило, недостаточна для выпуска новой версии продукта, всё же подразумевается, что гибкий программный проект готов к выпуску в конце каждой итерации. По окончании каждой итерации команда выполняет переоценку приоритетов разработки.

Agile-методы делают упор на непосредственное общение лицом к лицу. Все члены agile-команды работают в одном офисе. В команду включается и «заказчик» (product owner — заказчик или его полномочный представитель, определяющий требования к продукту; эту роль может выполнять менеджер проекта, бизнес-аналитик или клиент). В команду могут также входить тестировщики, дизайнеры интерфейса, технические писатели, программисты и менеджеры.

Основной метрикой **agile-методов** является рабочий продукт. Отдавая предпочтение непосредственному общению, **agile-методы** уменьшают объём письменной документации по сравнению с другими методами. Это привело к критике этих методов как *недисциплинированных*.

Многие руководители проектов, работающие в традиционных методологиях вроде «водопада», критикуют **agile-методы**.

Один из повторяющихся пунктов критики: при agile-подходе часто пренебрегают созданием **плана** развития продукта и **управлением требованиями**, в процессе которого и формируется такой **план**. Гибкий подход к управлению требованиями не подразумевает далеко идущих планов (по сути, управления требованиями просто не существует в данной методологии), а подразумевает возможность заказчика в конце каждой итерации выставлять новые требования, часто противоречащие архитектуре уже созданного продукта. Такое иногда приводит к катастрофическим «авралам» с массовыми переделками практически на каждой очередной итерации.

Кроме того, считается, что работа в **agile** мотивирует разработчиков решать все поступившие задачи простейшим и быстрейшим способом, при этом зачастую не обращая внимания на правильность кода с точки зрения требований нижележащей платформы (подход — «работает, и ладно», при этом не учитывается, что может перестать работать при малейшем изменении или же дать тяжёлые к воспроизводству дефекты после реального развёртывания у клиента). Это приводит к снижению качества продукта и накоплению дефектов.

2.1. Agile-манифест разработки программного обеспечения

В феврале 2001 в штате Юта США был выпущен «Манифест гибкой методологии разработки программного обеспечения». **Agile** является альтернативой управляемым документацией, «тяжеловесным» практикам разработки программного обеспечения, таким как «метод водопада» («каскадный метод»), являвшимся золотым стандартом разработки в то время. Данный манифест был одобрен и подписан представителями многих методологий: экстремального программирования, Crystal Clear, DSDM, Feature driven development, Scrum, Adaptive software development, Pragmatic Programming. Гибкая методология разработки использовалась многими компаниями и до принятия манифеста, однако именно после этого события произошло вхождение Agile-разработки в массы.

> ***Agile*** *— семейство процессов разработки, а не единственный подход в разработке программного обеспечения, и определяется* ***Agile Manifesto****.*

Agile не включает практик (**Agile Manifesto** не содержит практических советов), а определяет ценности и принципы, которыми руководствуются успешные команды.

Agile Manifesto содержит 4 основные идеи и 12 принципов.

Основные идеи изложены на главной странице оргкомитета Agile-манифест разработки программного обеспечения.

> «Мы постоянно открываем для себя более совершенные методы разработки программного обеспечения, занимаясь разработкой непосредственно и помогая в этом другим. Благодаря проделанной работе мы смогли осознать, что:
>
> *Люди и взаимодействие* **важнее** *процессов и инструментов*
> *Работающий продукт* **важнее** *исчерпывающей документации*
> *Сотрудничество с заказчиком* **важнее** *согласования условий контракта*
> *Готовность к изменениям* **важнее** *следования первоначальному плану*
>
> То есть, не отрицая важности того, что справа, мы всё таки больше ценим то, что слева.»
>
> Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas.

Основополагающие принципы Agile-манифеста

1. *Наивысшим приоритетом является удовлетворение потребностей заказчика*, благодаря регулярной и ранней поставке ценного программного обеспечения.

2. *Изменение требований приветствуется, даже на поздних стадиях разработки.* Agile-процессы позволяют использовать изменения для обеспечения заказчику конкурентного преимущества.

3. *Работающий продукт следует выпускать как можно чаще*, с периодичностью от пары недель до пары месяцев.

4. На протяжении всего проекта *разработчики и представители бизнеса должны ежедневно работать вместе.*

5. *Над проектом должны работать мотивированные профессионалы.* Надо создать условия, обеспечить поддержку и полностью довериться им.

6. *Непосредственное общение является наиболее практичным и эффективным способом обмена информацией* как с самой командой, так и внутри команды.

7. *Работающий продукт — основной показатель прогресса.*

8. *Инвесторы, разработчики и пользователи должны иметь возможность поддерживать постоянный ритм бесконечно.* Agile помогает наладить такой устойчивый процесс разработки.

9. *Постоянное внимание к техническому совершенству и качеству проектирования* повышает гибкость проекта.

10. *Простота — искусство минимизации лишней работы* — крайне необходима.

11. *Самые лучшие требования, архитектурные и технические решения рождаются у самоорганизующихся команд.*

12. *Команда должна систематически анализировать возможные способы улучшения эффективности* и соответственно корректировать стиль своей работы.

2.2. Методологии Agile

Методологии, которые придерживаются ценностей и принципов заявленных в **Agile Manifesto**:

Agile Modeling — набор понятий, принципов и приёмов (практик), позволяющих быстро и просто выполнять моделирование и документирование в проектах разработки ПО. Не включает в себя детальную инструкцию по проектированию, не содержит описаний, как строить диаграммы на UML. Основная цель: эффективное моделирование и документирование. Не охватывает программирование и тестирование, не включает вопросы управления проектом, развёртывания и сопровождения системы, однако включает в себя проверку модели кодом.

Agile Unified Process (AUP) — упрощённая версия IBM **Rational Unified Process** (**RUP**), разработанная Скоттом Амблером, которая описывает простое и понятное приближение (модель) для создания программного обеспечения для бизнес-приложений.

Agile Data Method — группа итеративных методов разработки ПО, в которых требования и решения достигаются в рамках сотрудничества разных кросс-функциональных команд.

DSDM — основан на концепции быстрой разработки приложений (Rapid Application Development, RAD). Представляет собой итеративный и инкрементный подход, который придаёт особое значение продолжительному участию в процессе пользователя/потребителя.

Essential Unified Process (EssUP) — объединение различных практик в рамках единой целостной методологии, охватывающей все аспекты разработки. По мнению автора методологии, одного из отцов **RUP**, *Ивара Якобсона* (*Ivar Jacobson*), основными характеристиками являются его открытость/свободная доступность, agile-практики и расширяемость.

Feature driven development (FDD) — функционально-ориентированная разработка. Используемое в FDD понятие функции или свойства системы достаточно близко к понятию прецедента, используемому в **RUP**, существенное отличие — это дополнительное ограничение: «каждая функция должна допускать реализацию не более, чем за две недели». То есть если сценарий прецедента достаточно мал, его можно считать функцией. Если же велик, то прецедент надо разбить на несколько относительно независимых функций.

Getting Real — итеративный подход без функциональных спецификаций, использующийся для веб-приложений. В данном методе сперва разрабатывается интерфейс программы, а потом её функциональная часть.

Scrum — устанавливает правила управления процессом разработки и позволяет использовать уже существующие практики кодирования, корректируя требования или внося тактические изменения. Использование этой методологии даёт возможность выявлять и устранять отклонения от желаемого результата на более ранних этапах разработки программного продукта.

OpenUP — итеративно-инкрементальный метод разработки программного обеспечения. Позиционируется как лёгкий и гибкий вариант RUP. OpenUP делит жизненный цикл проекта на четыре фазы: начальная фаза, фазы уточнения, конструирования и передачи. Жизненный цикл проекта обеспечивает предоставление заинтересованным лицам и членам коллектива точек ознакомления и принятия решений на протяжении всего проекта. Это позволяет эффективно контролировать ситуацию и вовремя принимать решения о приемлемости результатов. План проекта определяет жизненный цикл, а конечным результатом является окончательное приложение.

Lean Software Development — бережливая разработка программного обеспечения, использует методы концепции бережливого производства.

Принципы:

- *Исключение затрат.* Затратами считается всё, что не добавляет ценности для потребителя. В частности: излишняя функциональность; ожидание (паузы) в процессе разработки; нечёткие требования; бюрократизация; медленное внутреннее сообщение.

- *Акцент на обучении.* Короткие циклы разработки, раннее тестирование, частая обратная связь с заказчиком.
- *Предельно отсроченное принятие решений.* Решение следует принимать не на основе предположений и прогнозов, а после открытия существенных фактов.
- *Предельно быстрая доставка заказчику.* Короткие итерации.
- *Мотивация команды.* Нельзя рассматривать людей исключительно как ресурс. Людям нужно нечто большее, чем просто список заданий.
- *Интегрирование.* Передать целостную информацию заказчику. Стремиться к целостной архитектуре. Рефакторинг.
- *Целостное видение.* Стандартизация, установление отношений между разработчиками. Разделение разработчиками принципов бережливости: «мыслить широко, действовать мало, промахиваться быстро, учиться стремительно».

Экстремальное программирование (Extreme programming, XP)

2.3. Экстремальное программирование

Extreme Programming (**XP**, **экстремальное программирование**) — методология, сформированная в 1996 году. В основе методологии **XP** лежит командная работа, эффективная коммуникация между заказчиком и исполнителем в течение всего проекта. Разработка ведётся с использованием последовательно дорабатываемых **прототипов**.

Основные приёмы XP:

- Короткий цикл обратной связи (Fine scale feedback):
  - *Разработка через тестирование* (Test driven development).

    В XP особое внимание уделяется двум разновидностям тестирования: тестированию модулей (unit testing) и приёмочному тестированию (acceptance testing). Разработчик не может быть уверен в правильности написанного им кода до тех пор, пока не сработают абсолютно все тесты модулей разрабатываемой им системы. Тесты модулей позволяют разработчикам убедиться в том, что их код работает корректно. Они также помогают другим разработчикам понять, зачем нужен тот или иной фрагмент кода и как он функционирует. Тесты модулей также позволяют разработчику без каких-либо опасений выполнять **рефакторинг** (**refactoring**).

    Приёмочные тесты позволяют убедиться в том, что система действительно обладает заявленными возможностями. Кроме того, приёмочные тесты позволяют проверить корректность функционирования разрабатываемого продукта. Для XP приоритетным является подход TDD (Test Driven Development): сначала пишется тест, который не проходит, затем пишется код, чтобы тест прошел, а уже после делается **рефакторинг** кода.

— *Игра в планирование* (Planning game).

Основная цель игры в планирование — быстро сформировать приблизительный план работы и постоянно обновлять его по мере того, как условия задачи становятся всё более чёткими. Артефактами игры в планирование является набор бумажных карточек, на которых записаны пожелания заказчика (customer stories), и приблизительный план работы по выпуску следующих одной или нескольких небольших версий продукта. Такой стиль планирования оказывается эффективным, если заказчик отвечает за принятие бизнес-решений, а команда разработчиков отвечает за принятие технических решений. Если не выполняется это правило, весь процесс распадается на части.

— *Заказчик всегда рядом* (Whole team, Onsite customer).

«Заказчик» в XP — это не тот, кто оплачивает счета, а тот, кто на самом деле использует систему. XP утверждает, что заказчик должен быть всё время на связи и доступен для вопросов.

— *Парное программирование* (Pair programming).

Весь код создаётся парами программистов, работающих за одним компьютером. Один из них работает непосредственно с текстом программы, другой просматривает его работу и следит за общей картиной происходящего. При необходимости клавиатура свободно передаётся от одного к другому. В течение работы над проектом пары не фиксированы: рекомендуется их перемешивать, чтобы каждый программист в команде имел хорошее представление о всей системе. Таким образом, парное программирование усиливает взаимодействие внутри команды.

- Непрерывный, а не пакетный процесс:

— *Рефакторинг* (Design improvement, Refactoring).

> ***Рефакторинг*** *(refactoring) — это методика улучшения кода без изменения его функциональности.*

XP подразумевает, что однажды написанный код в процессе работы над проектом почти наверняка будет неоднократно переделан. Разработчики XP безжалостно переделывают написанный ранее код для того, чтобы улучшить его. Этот процесс называется рефакторингом. Отсутствие тестового покрытия провоцирует отказ от рефакторинга, в связи с боязнью поломать систему, что приводит к постепенной деградации кода.

— *Непрерывная интеграция* (Continuous integration).

В традиционных методиках интеграция, как правило, выполняется в самом конце работы над продуктом, когда считается, что все составные части разрабатываемой системы полностью готовы. Если выполнять интеграцию разрабатываемой системы достаточно часто, то можно избежать большей части связанных с этим проблем. В XP интеграция кода всей системы выполняется несколько раз в день, после того, как разработчики убедились в том, что все тесты модулей корректно срабатывают.

— *Частые небольшие релизы* (Small releases).

Версии (releases) продукта должны поступать в эксплуатацию как можно чаще. Работа над каждой версией должна занимать как можно меньше времени. При этом каждая версия должна быть достаточно осмысленной с точки зрения полезности для бизнеса.

Чем раньше выпустить первую рабочую версию продукта, тем раньше заказчик начнёт получать за счёт неё дополнительную прибыль. Следует помнить, что деньги, заработанные сегодня, стоят дороже, чем деньги, заработанные завтра. Чем раньше заказчик приступит к эксплуатации продукта, тем раньше разработчики получат от него информацию о том, что соответствует требованиям заказчика, а что — нет. Эта информация может оказаться чрезвычайно полезной при планировании следующего выпуска.

- Понимание, разделяемое всеми:

  — *Простота дизайна* (Simple design).

  XP исходит из того, что в процессе работы условия задачи могут неоднократно измениться, а значит, разрабатываемый продукт не следует проектировать заблаговременно целиком и полностью. Если в самом начале работы пытаться от начала и до конца детально спроектировать систему, то это будет напрасной тратой времени. XP предполагает, что проектирование — это настолько важный процесс, что его необходимо выполнять постоянно в течение всего времени работы над проектом. Проектирование должно выполняться небольшими этапами, с учётом постоянно изменяющихся требований. В каждый момент времени следует использовать наиболее простой дизайн, который подходит для решения текущей задачи, его изменяют по мере того, как условия задачи меняются.

  — *Метафора системы* (System metaphor).

  Архитектура — это некоторое представление о компонентах системы и о том, как они взаимосвязаны между собой. Разработчики используют архитектуру для того, чтобы понять, в каком месте системы добавляется некоторая новая функциональность и с чем будет взаимодействовать некоторый новый компонент.

  Метафора системы — это аналог того, что в большинстве методик называется архитектурой, она даёт команде представление о том, каким образом система работает в настоящее время, в каких местах добавляются новые компоненты и какую форму они должны принять. Подобрав хорошую **метафору**, можно облегчить команде понимание того, каким образом устроена разрабатываемая система. Иногда сделать это не просто.

  — *Стандарт кодирования* (Coding standard or Coding conventions).

  Все члены команды в ходе работы должны соблюдать требования общих стандартов кодирования. Благодаря этому: члены команды не тратят время на глупые споры о вещах, которые фактически никак не влияют на скорость работы над проектом; обеспечивается эффективное выполнение остальных практик.

Если в команде не используются единые стандарты кодирования, разработчикам становится сложнее выполнять рефакторинг; при смене партнёров в парах возникает больше затруднений; в общем и целом, продвижение проекта затрудняется. В рамках XP необходимо добиться того, чтобы было сложно понять, кто является автором того или иного участка кода, — вся команда работает унифицированно, как один человек. Команда должна сформировать набор правил и каждый член команды должен следовать этим правилам в процессе кодирования.

Перечень правил не должен быть исчерпывающим но не слишком объёмным: формулируются общие указания, благодаря которым код станет понятным для каждого из членов команды. Стандарт кодирования поначалу должен быть простым, затем он будет эволюционировать по мере того, как команда обретает опыт. Не следует тратить на предварительную разработку стандарта слишком много времени.

— *Коллективное владение кодом* (Collective code ownership) или *выбранными шаблонами проектирования* (Collective patterns ownership).

Каждый член команды несёт ответственность за весь исходный код. Таким образом, каждый вправе вносить изменения в любой участок программы. Парное программирование поддерживает эту практику: работая в разных парах, все программисты знакомятся со всеми частями кода системы. Важное преимущество коллективного владения кодом — в том, что оно ускоряет процесс разработки, поскольку при появлении ошибки её может устранить любой программист.

Давая каждому программисту право изменять код, увеличивается риск появления ошибок, вносимых программистами, которые считают что знают что делают, но не рассматривают некоторые зависимости. Хорошо определённые юнит-тесты решают эту проблему: если нерассмотренные зависимости порождают ошибки, то следующий запуск юнит-тестов будет неудачным.

- *Социальная защищённость программиста* (Programmer welfare): *40-часовая рабочая неделя.*

Глава 6

Планирование и контроль

1. Планы и планирование

Общий план проекта является основным документом, с которого должен начинаться любой проект. Рассмотрение плана как метода проектной деятельности указывает на его другой аспект: *регламенты и соглашения о том, как предполагается достигать цели.*

В Плане проекта должно быть отражено:

- содержание проекта и его границы;
- ключевые вехи проекта;
- плановый бюджет проекта;
- предположения и ограничения;
- требования и стандарты.

Планирование как род деятельности — это то, без чего любой сложный процесс, скорее всего, превратится в хаос. Точки зрения на планирование, представленные разными методологиями, отличаются очень сильно.

Для последовательного проекта план исходит из того, что каждый из этапов ЖЦ ПО в принципе может быть выполнен полностью, поставляя результаты для следующего этапа. Поэтому задачи, которые требуется решать на каждом этапе, диктуются сразу всеми задачами проекта. Эту позицию называют *монументальностью последовательных проектов*. Такие проекты могут претендовать на успех только в том случае, если можно предсказать, какое развитие будет иметь данная разработка. Эта позиция годится только для *предсказуемых проектов*.

Для итеративных проектов план рассматривается по-другому: установка на полное выполнение каждого из этапов ЖЦ действует только в пределах фиксированной для итерации задачи. Задачи последующих итераций учитываются лишь как

возможные перспективы. Поэтому планированию работ итерации предшествует отбор требований и сценариев, которые должны быть решены на этой итерации. Точки зрения на критерии отбора и на задачи, оставляемые для реализации на других итерациях разделяют подходы, которые относятся к разным *методологиям итеративного развития проектов*. Практически все сходятся лишь в том, что в качестве текущих задач должны быть выбраны те, решение которых реализует наиболее актуальные для пользователей сценарии. В остальном же взгляды расходятся, что вполне объяснимо спецификой позиций разработчиков по отношению к предсказуемости развития проектов.

Консервативная позиция итеративного направления требует создания на текущей итерации *базы для последующего развития*. Необходимо планировать к реализации ещё и те средства, которые потребуются в будущем, а реализацию отобранных сценариев строить так, чтобы возможно было развитие без переписывания кода. Эта позиция поддерживается средствами ООАП. Кроме того, разрабатываются специальные приёмы проектирования, использование которых способствует независимости частей системы и, как следствие, простоте наращивания без переделок.

Консервативная позиция по существу, как и в случае последовательных проектов, ориентируется на предсказуемое развитие и, как все последовательные методологии, является монументальной.

Радикальная позиция итеративного направления представлена в рамках методологического направления быстрого развития программных проектов (**RAD**) и Agile-методологий, в частности, в случае **экстремального программирования**. Здесь вовсе не пытаются заглядывать вперёд, далее задачи текущей итерации, которая определяется исключительно актуальностью реализуемых сценариев для пользователей. Вместо планирования как процесса, диктуемого внешней постановкой проектного задания, используется, так называемая, *игра в планирование*, в ходе которой разработчики и заказчики в диалоге определяют, что можно реализовать в ближайшей версии системы, всё остальное отбрасывается. Работы ведутся так, как будто они в любой момент могут быть направлены по совершенно непредсказуемому пути.

Основная цель **игры в планирование** — быстро сформировать приблизительный план работы и постоянно обновлять его по мере того, как условия задачи становятся всё более чёткими. Артефактами игры в планирование является набор бумажных карточек, на которых записаны пожелания заказчика (customer stories), и приблизительный план работы по выпуску следующих одной или нескольких небольших версий продукта.

Критическим фактором, благодаря которому такой стиль планирования оказывается эффективным, является то, что в данном случае заказчик отвечает за принятие бизнес-решений, а команда разработчиков отвечает за принятие технических решений. Если не выполняется это правило, весь процесс распадается на части.

Последовательный и экстремальный подходы — это две крайние позиции, по отношению к планам. Реальность такова, что обе эти позиции являются идеализацией. Первая обыкновенно нарушается, причём не только в связи с ошибками этапов, предшествующих текущим работам. Вторая не может не рассчитывать на гипотезы

о продолжении, о том, как проект будет развиваться, и разработчики всегда готовятся ко всем возможным поворотам. В экстремальном программировании изменение плана считается нормой и на деле оно постоянно происходит. Это не значит, что с появлением новой актуальной для реализации пользовательской истории или со сменой приоритетов кто-то переписывает некий документ, утверждает его, производит другие бюрократические ритуалы. Корректируется представление работников о перспективах, а как и когда оно материализуется — другой вопрос.

Главная причина различий отношений к планам проекта — различия гипотез о степени изменчивости и непредсказуемости требований.

Выигрышная стратегия опирается на разбиение проекта на две части: *ближайшую* и *перспективные задачи*. Для ближайшей задачи неопределённость развития должна быть сведена к минимуму, пусть даже за счёт ошибочных предположений о перспективах. Следовательно, планирование решения ближайшей задачи не только возможно, но и целесообразно. Для перспективных задач должна быть определена стратегия, которая не должна быть жёстко фиксированной. При такой установке последовательное и экстремальное развитие проектов смыкаются: различия только в масштабе ближайшей задачи проекта и в отношении к одинаково неопределённой гипотезе о продолжении. В первом случае любое продолжение есть новый проект, а во втором имеется в виду настолько широкий спектр продолжений, что ориентация на какое-либо из них считается бессмысленной.

Все итеративные методологии укладываются в эту схему, если принять, что при определении ближайшей задачи часть требований сознательно откладывается до последующих итераций. Скорее всего, но совсем необязательно, эта часть будет реализовываться в дальнейшем среди других требований, которые, возможно, появятся в течение развития проекта. И опять крайние позиции смыкаются: для последовательных проектов откладываемая часть требований считается пустой (ничто не откладывается), а для быстрых — это требования, которые не удовлетворили критерию пользовательской актуальности, понимаемой строго на момент выбора реализуемых сценариев.

Для большинства методологий считается, что основой планирования программной разработки является принятая для проекта модель ЖЦ. Как правило, выполняемые работы укладываются в основные (общие для выбранной методологии) этапы.

Конкретные проекты чаще всего требуют введения в планы дополнительных этапов и контрольных точек, которые отражают начало и завершение важных или ответственных работ, события, существенные для проекта. Конкретизация схемы ЖЦ — *это первоочередная работа по составлению планов проекта.* И уже на этом уровне видно, что выполнение её для большинства основных этапов зависит от результатов предыдущих этапов. Наглядный пример тому — невозможность конкретного планирования этапа программирования без проведения архитектурных работ, определяющих декомпозицию системы, которая служит базой для выставления заданий разработчикам подсистем. Зависимость работ, выполняемых в ходе развития проекта, требует своих методов преодоления трудностей конкретизации планов.

На уровне первичного планирования используется *принцип подмены конкретных проектных заданий стратегическими установками*, которые уточняются, когда появляется дополнительная информация. Положения о связи планирования с ЖЦ сформировались в традиционных методологиях, они (как и всё традиционное

планирование) отвергаются *быстрыми подходами*. В то же время, с учётом различий типов ЖЦ быстрых и традиционных методологий, эти связи сохраняются, хотя и приобретают некоторую специфику.

Так, конкретизация схемы ЖЦ для RAD в расчёте на долгосрочную перспективу является *планом релизов*. С этого, как и при традиционном подходе, начинается планирование, а точнее — ориентировочное прогнозирование проекта, без которого просто немыслимо получить заказ. Последовательная зависимость итераций друг от друга в этом подходе, безусловно, предполагается, хотя не ей, а другим критериям (актуальности для пользователя) отдаётся предпочтение при распределении работ по итерациям.

Оценки плана даются *априорно* (до его выполнения, когда разработчики договариваются с заказчиком о том, какие функции и за какие сроки они будут реализовывать на предстоящей итерации) и *апостериорно* (одна их целей которых — корректировка предположений о процессе на основании полученного опыта). *Контрольные точки* играют важную роль в традиционном планировании, для итераций дольше трёх недель. Локализованный в этих точках контроль эффективен только тогда, когда он выявляет отклонения от плана. Только в этом случае необходимы воздействия, которые и оправдывают смысл контрольных мероприятий. Но отклонение могло произойти ещё до контроля, а значит, для своевременного воздействия момент упущен. В то же время контроль сроков выполнения заданий необходим для того, чтобы постоянно вносить коррективы в развивающийся план.

Кент Бек и *Мартин Фаулер*[1] считают, что *перенос сроков это чрезвычайная ситуация*. Если корректировка необходима, то лучше производить её по объемам работ, запланированных в контрольной точке. Но выявлять нарождающуюся необходимость корректировки нужно всегда как можно раньше. Отсюда следует, что при любой методологии требуется постоянное текущее наблюдение за ходом развития этапа проекта. Для задач, решаемых в контрольных точках, — использовать лишь оценку соответствия планируемых и полученных результатов, на основе которой корректируется план (это можно считать корректировкой целевой области и траектории планировочной деятельности).

Эти рекомендации не зависят от методологии и стратегии и указывают на целесообразность принципа выяснения отклонений и быстрой корректировки. Дальнейшая конкретизация того, как осуществлять текущее наблюдение, безусловно, необходима, но это уже зависит от выбранной для проекта методики.

Для методологий, предполагающих организацию производства как процесс, допускающий внешнее отслеживание, общий план развития проекта (в виде суммы планов процессов), которые должны осуществляться в ходе выполнения проектной деятельности в целом, является обязательным, документально оформленным руководством разработчиков. Его корректировка допускается, но весьма регламентированная: в контрольных точках (заранее спланированных!) и в ходе откликов на риски. Конечно, если в результате наблюдения выясняется, что текущий этап проекта не может быть выполнен до конца или запланированное выполнение не имеет смысла и что целесообразно изменить содержание проектных или этапных задач, то материалы для такой корректировки готовятся в момент выявления отклонения.

[1]Martin Fowler — автор ряда книг и статей об архитектуре ПО, объектно-ориентированному анализу и разработке, языку UML, рефакторингу, экстремальному программированию.

Они согласуются с заказчиком, чтобы в контрольной точке ограничиться лишь юридическими аспектами и утверждением новых условий продолжения работ.

Роль планирования ставится очень высоко. Общий план и его составляющие рассматриваются как главные документы проекта. Так, стандарт **PMBOK** последней версии из 47 процессов, которые он фиксирует в проектной деятельности, 24 процесса относит к группе планирования. Последняя версия стандарта не требует обязательного составления всех этих планов, но должна сохраняться возможность доказать, что при выполнении проекта все обозначенные в них процессы должны быть предусмотрены.

2. Наблюдения и контроль

Если планы рассматривать как основу организации работ, то контроль — это текущая деятельность, которая осуществляется для того, чтобы компенсировать неизбежные отклонения от планов. Способы контроля могут быть совершенно разными. Причины различий — и специфика проектов и коллективов, и выбранная методология процесса разработки. Правильный, разумно организованный контроль со стороны менеджера должен сбалансированно сочетать в себе два противоречивых аспекта:

- Необходимо добиваться знания абсолютно точной картины того, как выполняются порученные исполнителям работы. И если наблюдаются отклонения от принятых планов, обязательств и соглашений, то менеджер должен принять соответствующие меры для исправления ситуации.

- Необходимо добиваться того, чтобы наблюдение и контроль не мешали деятельности исполнителей. Если эти действия требуют от исполнителей времени, других ресурсов, то нужно добиваться их минимизации.

Наблюдения, контроль и оценка — постоянная обязанность менеджера. С каждой из них связаны мероприятия, осуществляемые для получения исходной информации и выполнения воздействий.

В первом случае это наименее затратные мероприятия. Менеджер, как наблюдатель заинтересован в том, чтобы извлечение информации не требовало или почти не требовало ни усилий, ни времени. Важно постараться построить взаимоотношения так, чтобы в случаях, не нуждающихся в специальном внимании, ни наблюдателю, ни работнику совсем не приходилось ничего делать специально.

Для наблюдений применяются:

- *анализ кода программ и данных автоматического мониторинга* с помощью специальных инструментов;

- *специальное, т. е. со стороны наблюдателя, тестирование;*

- *опрос работников* (что означает не устный отчёт, а свободный разговор, в ходе которого выясняется, как идут дела);

- *простые и короткие вопросы по сути*: как было принято то или иное решение, от чего пришлось отказаться, что не удалось предусмотреть и т. д.;

- *анализ коротких отчётов, периодически (возможно, ежедневно) предоставляемых разработчиками и автоматически интегрируемых в сводные таблицы.*

Эти действия должны давать представление о том, что ещё осталось сделать сотруднику до установленного контрольного срока. При подозрениях, что сроки нарушаются, требуется более пристальное наблюдение. Если опасения подтверждаются, то необходимо выяснить причины отклонений и попытаться их как-то нейтрализовать или компенсировать. Здесь годятся «провокационные» вопросы, цель которых — поставить разработчика в ситуацию, в которой он сам идентифицирует проблему и, быть может, найдёт решение. Только в крайних случаях, когда все эти локальные воздействия не приводят к нужным результатам, можно обратиться за помощью к другим разработчикам команды и внешним специалистам.

Мероприятия в контрольных точках проводятся, во-первых, для того, чтобы доказать, что определённые заранее локальные цели проекта достигнуты (*контрольные функции*), а во-вторых, чтобы узнать качество рабочих продуктов процесса (*оценочные функции*). Мероприятия в контрольных точках должны обеспечивать уверенность доступных инициаторов работ в правильности стратегической линии и принятых решений, а также в том, что результаты удовлетворяют заранее фиксированным критериям. Когда наблюдение организовано правильно, специальных действий для контрольных функций, скорее всего, не потребуется. В контрольных точках, которые заранее определены для независимой инспекции, а также, если на материале наблюдений цель контроля не достигнута, прибегают к **верификации** и **аттестации** как к явным процедурам.

Для конца того или иного этапа выполнение соответствующего контроля обязательно. В качестве обязательных рассматриваются также моменты начала и окончания наиболее важных запланированных работ, моменты согласования работ и другие события. Помимо планируемых обязательных контрольных точек целесообразно предусматривать **факультативные точки**, в которых контроль осуществляется при определённых условиях (например, отставание от графика работ).

Для отдельной работы мероприятия, связанные с контролем, должны проводиться *не чаще чем раз в неделю, но не реже чем ежемесячно*. Для долгосрочных работ (больше месяца) необходимы промежуточные контрольные точки.

3. Оценка выполнения проектных заданий

Задача оценивания результатов проектной деятельности имеет очень много аспектов. Следует различать *внутреннюю оценку*, направленность которой связывается с улучшением качества процесса производства, роста квалификации сотрудников и т. п., и *внешнюю оценку*, которая отражает отношение к проектной деятельности с точки зрения потребителей продукции и других инициаторов работ, не связанных с производством рабочих продуктов непосредственно. Эти оценки могут (а иногда и должны!) существенно различаться. Но как та, так и другая должны быть представлены в процедуре оценивания, баланс между ними достаточно очевиден. Внутренняя оценка по сравнению с внешней имеет больший удельный вес в контрольных точках, которые выставлены для коррекции возможных отклонений проектной деятельности

от целей. Внешняя оценка — в тех точках, которые связаны с выпуском продукции. Тем не менее более низкий удельный вес внутренней или внешней оценок не означает, что в соответствующих контрольных точках им не следует уделять внимания вовсе: внешняя оценка даёт главный критерий качества процесса производства, обусловленный потребительской значимостью, а внутренняя оценка определяет вектор развития коллектива.

В любом подходе, который предполагает итеративное развитие проекта, в конце основных работ итерации выполняется этап оценки. Это, пожалуй, самый критичный этап с точки зрения развития, поскольку он определяет дальнейшую стратегию. При выполнении работ этапа анализируются как полученные результаты, так и качество проведённого планирования и контроля работ.

Типичные параметры для *оценивания результатов*:

Оценка продукта безотносительно его производства. Определяется его эффективность с точки зрения автоматизации пользовательской деятельности. Целесообразно при выработке этой оценки брать за основу мнения пользователей и других заинтересованных лиц.

Оценка побочных продуктов производства. Побочные продукты могут быть использованы в данном проекте, в других проектах, которые разрабатываются (будут разрабатываться) данным коллективом или независимыми специалистами. При выставлении этой оценки в рамках данного проекта нужно указать, для каких сценариев предполагается переиспользование. Для переиспользования в других внутренних проектах следует указать качества, которые позволяют говорить об этой возможности, а для внешнего переиспользования, кроме того, требуется обозначить уровень готовности подсистемы, документа и т. п. как продукта (наличие руководств, пособий и др.).

Оценка соответствия требованиям, которые известны на текущий момент (первичным, поступающим в ходе выполнения проекта и после начала использования).

Оценка удовлетворения пользовательской потребности. Эта оценка выставляется на основе анализа обратной связи с пользователями (изучаются их рекламации, предложения и пожелания).

Оценка соответствия спросу. Выставление этой оценки — задача маркетологов. В команде, выполняющей проект, такие специалисты не обязательно представлены, но это означает не отказ от работ, необходимых для оценивания спроса, а лишь расход средств для их проведения внешними силами.

Оценка соответствия рыночным потребностям. Здесь, в отличие от предыдущей оценки говорится о сравнении результатов с тем, что предлагается на рынке и может оказаться конкурирующим товаром, дополняющим, сопутствующим продуктом и др. Комментарий в точности повторяет предыдущий.

Оценка качества. Эта задача решается при таком рассмотрении продукта, которое обосабливается от процесса его производства. Необходимо провести сравнительный анализ оцениваемых результатов с конкурирующими разработками

и выяснить его достоинства и недостатки, а также определить границы применимости продукта.

Полученные оценки дополняются оценками *процесса разработки* и *планирования*.

Оценка соответствия графику запланированных работ. С точки зрения качества планирования это главная оценка процесса производства. Расхождение предварительной и фактической оценок чаще всего свидетельствуют только о недостаточности априорной информации о возможностях развиваемого проекта, а не о недобросовестности сотрудников. Смысл оценки соблюдения графика в том, что в результате каждый последующий план становится всё более точным.

Оценка проводимых мероприятий. Это оценка организационных инструментов, которые применялись в ходе выполнения работы. Исходным материалом для неё служат накапливаемые после каждого мероприятия сведения о его эффективности. По сути дела, выставляя оценку мероприятий, менеджер выясняет методы своей работы и их соответствие специфике проекта и команде исполнителей. По мере того как проект проходит всё большее число итераций, данная оценка становится всё более точной.

Оценка коллектива осуществляется по следующим параметрам:

- квалификация сотрудников;
- рост квалификации сотрудников;
- стабильность кадров;
- слаженность;
- распределение обязанностей и разделение труда.

Эта оценка должна быть непрерывной, т. е. менеджер оценивает коллектив в течение всего времени работы, в ходе наблюдений за проектом. На этапе оценки подводится итог. В результате проведения этих работ менеджер может более точно устанавливать назначение *проектных ролей* исполнителям. Это и является целью оценки коллектива, а вовсе не публичные поощрения и порицания.

Оценка реалистичности плана. Эта оценка, с одной стороны, соприкасается с оценкой соответствия графику запланированных работ. С другой стороны, она рассматривается как апостериорное подтверждение/опровержение выполнимости запланированных работ.

Оценка выполнения каждого из видов плана. В рамках планирования работ в зависимости от принятой методической установки и в соответствии с ней определяются различные виды планов (график этапов и работ проекта, план отслеживания рисков и мероприятий по их преодолению, схемы расходования ресурсов и др.). Качество выполнения каждого из них требует проверки. На основании анализа сведений, касающихся того или иного плана, в сопоставлении их с общим контекстом проектных результатов определяется эффективность этого плана как инструмента управления.

Представленный перечень никак не связан ни со спецификой проекта, ни с особенностями принятой методологии. В конкретных случаях он может быть расширен — это будет означать, что в оценке нуждаются другие аспекты процесса разработки (пример — взаимодействие с подрядчиками), или сужен, если какие-то из аспектов признаются несущественными, т. е. их оценка либо тривиальна, либо просто остаётся на интуитивном уровне.

Оценивание, как организационная процедура должно осуществляться непрерывно с подведением итогов и принятием решений во время этапа оценки. Оценивание (в терминологии модели «фазы — функции») есть *производственная функция* с распределением интенсивности выполнения по жизненному циклу. Производственная функция — это количественная зависимость между количеством выпускаемой продукции и факторами производства (затраты ресурсов, уровень технологий, и др.). Естественные локальные пики её интенсивности связываются с контрольными точками (см. рис. 6.1). Следует подчеркнуть, что рассмотрение оценивания как производственной функции не зависит от методик выполнения оценок, равно как и от выбранной методологии проектирования. Разумеется, распределение интенсивности зависит от этих факторов.

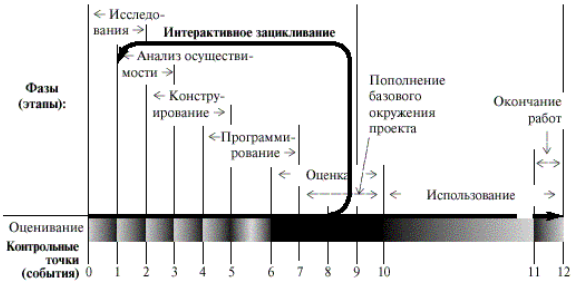


Рис. 6.1. Оценивание как производственная функция

4. Цикл управления проектом

Наблюдение, **контроль** и **оценивание** — это элементы управления любым проектом, использование которых направляется планированием. Можно сказать, что это инструменты управления. Если управление рассматривать в качестве способа организации работы исполнителей над проектом, то связанную с ним деятельность менеджера можно представить в виде цикла управления, включающего в себя следующие процессы (деятельности):

постановка задач — определение заданий для разработчиков, проверка понимания заданий и др. действия, необходимые для решения задачи текущего периода;

выделение ресурсов и указание контрольных сроков — обеспечение осуществимости решения задачи текущего периода;

текущий мониторинг активизированных процессов решения задачи — набор действий, которые позволяют менеджеру быть в курсе того, как исполнители справляются со своими сферами ответственности, определёнными для них постановкой задачи периода (обычно эти действия подчиняются специальным методикам организации управления в части наблюдений);

мероприятия в контрольных точках — контрольные и оценочные действия, позволяющие сформулировать итоги решения поставленной задачи с точки зрения запланированных результатов и с позиций качества процесса выполнения работ;

оптимизация процесса разработки — коррекция расстановки кадров, априорных решений и планов.

Составляющие цикла управления естественно связывать с контрольными точками линии жизни проекта, используя их как точки отсчёта управленческой деятельности. Это именно те моменты, когда выдаются задания разработчикам на очередные работы и подводятся итоги завершаемой ими деятельности, вложенной в деятельность проекта в целом. Схематически этот цикл изображён на рис. 6.2.

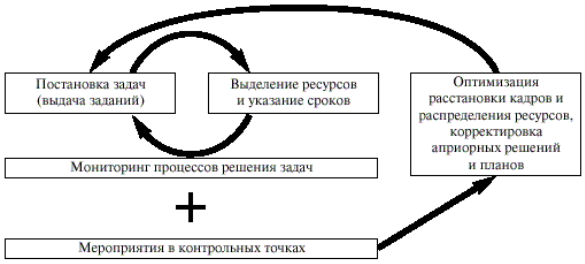


Рис. 6.2. Цикл управления в проекте

Представленная схема не связана ни с выбором методической стратегии для выполнения проекта, ни вообще со спецификой программистских работ. Она фиксирует метод работы менеджера как элемент деятельности. Рассмотренный цикл управления даёт представление о локализованной (привязанной к контрольным точкам) деятельности менеджеров.

Таким образом, на самом абстрактном уровне представлена организация решения управленческих задач. В каждом конкретном случае управляемые процессы и цикл управления наполняются своим содержанием.

Глава 7

Инструментальные средства управления проектами

1. Продукты, ориентированные на автоматизацию услуг (PSA)

Рассмотрим сначала PSA-системы. **PSA-системы** (Professional Services Automation — автоматизация деятельности компаний), — комплексные информационные системы, предназначенные для автоматизации профессиональной деятельности компаний, предоставляющих продукты/услуги на проектной основе. Такое ПО позволяет значительно увеличить эффективность и прозрачность бизнеса компании.

- Agresso.
- Augeo Software.
- CA Clarity.
- Epicor Software.
- IRIS Software Group SharpOWL.
- Lawson;
- Maconomy.
- Microsoft Project Professional.
- OpenAir — разработана NetSuite совместно с Российской компанией АйГейт (iGate ltd.). http://www.igate.ru/products/psa Функциональность *облегчённой версии OpenAir* включает: автоматизированный учёт времени и расходов и выставления счетов; мощную отчётность в рамках управления проектами; базовое управление проектами и возможность создавать комплексные структуры работ; интерфейс с системами QuickBooks и salesforce.com; бесплатный интерфейс с системой NetSuite.

Профессиональная версия OpenAir предоставляет: все возможности облегчённой версии OpenAir; мощный биллинговый механизм; функциональность управление ресурсами; функциональность признания доходов; мультивалютность и множественные графики; возможность автономной работы через iPhone, Blackberry, КПК и обычный компьютер; интерфейса с системами (кроме salesforce.com и QuickBooks), Microsoft Dynamics, Microsoft CRM, MAS90, Sage, Solomon, Peachtree.

Корпоративная версия предоставляет: все возможности профессиональной версии OpenAir; выделенного менеджера по работе с клиентами; поддержку в режиме 24/7; неограниченное количество точек интеграции; бесплатный доступ к API.

- Oracle E-Business Suite.
- Oracle PeopleSoft.
- Primavera Systems Evolve.
- ProjectMate — Российская PSA-система автоматизации профессиональной деятельности. Помимо модуля управления проектами имеет массу функций, востребованных в компаниях сферы консультационных услуг (юристы, адвокаты, аудиторы и пр.) — начиная от учёта времени и заканчивая выставлением счетов (биллингом).
- QuickArrow.
- SAP Professional Services Automation.
- Tenrox.

2. Системы управления проектами и задачами

- Basecamp — онлайн-инструмент для управления проектами, совместной работы и постановки задач по проектам, созданный компанией 37signals. В процессе разработки и практического использования Basecamp был создан фреймворк Ruby on Rails.
- Bitrix24 — Российская облачная система управления проектами. Для одного небольшого проекта тариф бесплатный. Имеется коробочный вариант, для установки на сервер компании (в том числе для очень больших холдингов).
- Confluence — облачная вики-система для создания единой базы знаний проектной команды от австралийской компанией Atlassian. Написана на Java, является одним из основных продуктов (наряду с системой отслеживания ошибок Jira). Распространяется под проприетарной лицензией. Бесплатна для некоммерческих организаций и открытых проектов в облаке (до 10 пользователей).
- Devprom — система управления проектами с поддержкой полного цикла разработки проектов.

- DotProject — OpenSource система управления проектами. Признана разработкой месяца (апрель 2009) на SourceForge.net.
- Easy Projects .NET — система для управления проектами, написанная на .NET.
- eGroupWare — бесплатное ПО для управления проектами.
- Entexo iProject — система автоматизации задач для управления проектами.
- Feng Office — корпоративный веб-офис для совместной работы с проектами, задачами и документами. Этот онлайн-офис с открытым исходным кодом, разработанный сообществом OpenGoo. Приложение может быть скачано и установлено на сервер. Список его основных функций включает систему управления документами, управление контактами, e-mail, управление проектами и управление временем. OpenGoo также может быть отнесено к категории Groupware и к персональному информационному менеджеру. Основная функция — управление (документооборотом (WORKFLOW), контактами (CRM), проектами (PM), временем (TM), коммуникациями (e-mail, уведомления)). Работа с проектами — использует концепцию «рабочих пространств», которую можно сравнить с виртуальными отделами и командами. Рабочее пространство даёт возможность: распределять права доступа, делегировать полномочия, контролировать выполнение задач. В каждом рабочем пространстве пользователь может посылать электронную почту, сохранять комментарии, управлять задачами, назначать этапы, загружать/выгружать файлы, добавлять контакты, ссылки, события, назначать задачи другим участникам. Проект может объединять клиентов, документы или любые другие объекты, классифицированные в системе, содержать подпроекты (можно построить иерархию проектов до 10 уровней), использовать для разграничения доступа (пользователь имеет права на работу с определёнными проектами, тогда как другие проекты будут полностью скрыты от него).

  Имеется пробная версия, бесплатный вариант поставки и платные тарифы. Версия Feng Office Community Edition (ранее известная по названием OpenGoo) может быть скачана и установлена на сервер бесплатно.
- GanttProject — маленькая бесплатная программка с диаграммой Ганта и ресурсами.
- Invest Sign — ПО для семейства MS Windows.
- IBM Project and Product Portfolio Management — управление проектами от IBM.
- IBM Rational Method Composer — гибкая платформа для управления процессами, предлагающая инструменты и богатейшую библиотеку процессов, чтобы помочь компаниям повышать эффективность для успешной разработки ПО и реализации других ИТ-проектов; эффективная среда управления конфигурациями.

- Jira Software — самый популярный в мире инструмент для управления проектами разработки ПО для agile-команд. Написана на Java, эффективна для отслеживания ошибок, является одним из основных продуктов (наряду с Confluence, могут использоваться совместно). Распространяется под проприетарной лицензией. Бесплатна для команд до 10 пользователей (облачная версия).

- MeisterTask — облачный сервис управления проектами, ориентированный на поддержку гибких методологий. Базовый тариф (бесплатный) включает базовые функции для одиночного управления задачами.

- Microsoft Project Standard — однопользовательская версия MSP для небольших проектов.

- OnePoint Project — первый проект с открытым кодом.

- PHProjekt — набор утилит группового взаимодействия для поддержки связи и управление группами и компаниями через интранет и Интернет. PHProjekt — это пакет программного обеспечения для совместной работы, который поддерживает взаимодействие и управление командами и компаниями. Он состоит из нескольких компонентов, включая групповой календарь с резервированием ресурсов, систему временных карт, управление проектами, трекер запросов, общую файловую систему, менеджер контактов, почтовый клиент, форум, чат, заметки, общие закладки, списки задач, система голосования и напоминания. API позволяет интегрировать несколько доступных дополнений (например, Интернет-магазин, пакет WAP, инструменты синхронизации) или собственные приложения. Он доступен для MySQL/MariaDB, PostgreSQL, Interbase, Oracle, Informix, MS-SQL и 25 языков.

- Project Kaiser — веб-ориентированная система управления проектами и задачами с поддержкой wiki и развитыми средствами взаимодействия пользователей.

- ProjectLibre (ранее — OpenProj) — кроссплатформенное ПО для управления проектами. Распространяется на условиях лицензии Common Public Attribution License Version 1.0 (CPAL). Позиционируется создателями как открытая замена коммерческому продукту Microsoft Project Standard, имеет схожий с MS Project интерфейс и аналогичный подход к построению плана проекта. ПО доступно для ОС Microsoft Windows, Linux, Unix, Mac OS X, локализовано на 27 языков. ProjectLibre вышло в августе 2012, сразу же в сообществе SourceForge признан проектом месяца. Последняя версия 1.9.3 (8 января 2021).

- Spider Project — российская система управления проектами.

- Borland StarTeam — автоматизированная система управления конфигурацией и изменениями, которая обеспечивает эффективный контроль процесса разработки. StarTeam улучшает взаимодействие между всеми сотрудниками проектных групп, предоставляя пользователям доступ к любой важной информации проекта через центральный репозиторий, который поддерживается системой управления технологическими потоками и процессами. StarTeam превосходит традиционные SCM-средства, которые предлагают только функции контроля

файловых версий, поскольку предоставляет пользователям гибкое, удобное в использовании комплексное решение на основе систем управления требованиями, управления изменениями, отслеживания дефектов, контроля файловых версий, тематических обсуждений, управления задачами и управления проектом.

- TrackStudio Enterprise — система управления задачами. Есть экспорт в MS Project.

- Trac — инструмент управления проектами и отслеживания ошибок в программном обеспечении. Trac является открытым программным обеспечением, разработанным и поддерживаемым компанией Edgewall Software.

  Trac использует минималистский веб-интерфейс, основанный на технологии Wiki, и позволяет организовать перекрёстные гиперссылки между базой данных зарегистрированных ошибок, системой управления версиями и вики-страницами. Это даёт возможность использовать Trac в том числе и как веб-интерфейс для доступа к системе контроля версий Subversion, а также, через плагины, к Mercurial, Git, Bazaar и другим. Trac написан на языке программирования Python и в настоящее время распространяется по модифицированной лицензии BSD. Поддерживаются базы данных SQLite, PostgreSQL, MySQL и MariaDB. В качестве системы HTML-шаблонов веб-интерфейса Trac до версии 0.11 использовал ClearSilver. Новые версии, начиная с 0.11, используют разработанную в Edgewall систему шаблонов Genshi, при этом совместимость с плагинами, использующими ClearSilver, будет оставлена еще в течение нескольких версий.

- WebCollab — легковесная (< 1 МБ) свободная система. Зарегистрированные в системе пользователи могут объединятся в команды. Можно создавать несколько проектов, создавать задачи, отслеживать их выполнение. Коммуникации через форум и email.

- «Офис Управления Проектами — PMO» — Российская система управления проектами и портфелями проектов на веб-платформе.

- JIRA — система отслеживания ошибок от компании Atlassian, предназначена для организации взаимодействия с пользователями, часто используется и для управления ИТ-проектами по гибкой методологии. JIRA это продукт, предназначенный для организации процесса контроля запросов и задач, имеющий часть функциональности обычно присущей большим и дорогим системам управления проектами.

  Ключевыми понятиями в JIRA являются проекты и задачи. Задачи создаются в проектах, для выполнение задач назначаются исполнители. Задачи могут быть разного типа и иметь подзадачи, задачи могут быть связанными с другими задачами. Статус задач меняется в процессе их выполнения.

  Есть пробная версия.

- RedMine.

2.1. RedMine

RedMine — гибкая веб-система управления проектами с открытым исходным кодом, написанная на языке Ruby. Этот open-source проект имеется в репозиториях многих GNU/Linux.

Существует коммерческий вариант — Easy Redmine. Redmine оптимизирует и упрощает:

- создание проектов;
- реализацию проектов;
- поддержку проектов после реализации;
- внутриофисные процессы;
- личное планирование и управление временем.

Может работать в облаке или на сервере компании, локализация на 14 языков.

Также имеются клиентские приложения для многих популярных мобильных платформ.

Возможности Redmine:

- ведение нескольких проектов;
- гибкая система доступа, основанная на ролях;
- система отслеживания ошибок;
- диаграммы Ганта и календарь;
- ведение новостей проекта, документов и управление файлами;
- оповещение об изменениях с помощью RSS-потоков и электронной почты;
- вики для каждого проекта;
- форумы для каждого проекта;
- учёт временных затрат;
- настраиваемые произвольные поля для инцидентов, временных затрат, проектов и пользователей;
- лёгкая интеграция с системами управления версиями (SVN, CVS, Git, Mercurial, Bazaar и Darcs);
- создание записей об ошибках на основе полученных писем;
- поддержка множественной аутентификации LDAP;
- возможность самостоятельной регистрации новых пользователей;

- многоязычный интерфейс (в том числе русский);
- поддержка СУБД MySQL/MariaDB, PostgreSQL, SQLite, Oracle.

RedMine используется во многих, в т. ч. открытых проектах (см., например, forge.ispras.ru).

Официальный сайт проекта: http://www.redmine.org/. Скачать инсталлятор (в более удобной форме) можно с ресурса https://bitnami.com/stack/redmine.

3. Системы управления версиями

> ***Система управления версиями*** *(Version Control System,* ***VCS*** *или Revision Control System) — программное обеспечение для хранения нескольких версий одного и того же документа.*

При необходимости **VCS** может возвращаться к более ранним версиям, определять, кто и когда сделал то или иное изменение, и многое другое.

Распределённые системы управления версиями (Distributed Version Control System, DVCS) используют распределённую модель вместо традиционной клиент-серверной. Они, в общем случае, не нуждаются в централизованном хранилище.

Вся история изменения документов хранится на каждом компьютере, в локальном хранилище, и при необходимости отдельные фрагменты истории локального хранилища синхронизируются с аналогичным хранилищем на другом компьютере. В некоторых таких системах локальное хранилище располагается непосредственно в каталогах рабочей копии.

Утилиты

- Make — классическая утилита для создания ПО с отслеживанием изменений, входит в состав всех платформ.
- Automake — утилита GNU для автоматического создания файлов Makefile.
- A-A-P — это инструмент с открытым кодом для разработки ПО, созданный с целью заменить классическую утилиту Make.

Инструменты

- GNU Arch — кроссплатформенная распределённая система управления версиями, которая является частью проекта GNU и лицензируется по GNU GPL. Он используется для отслеживания изменений, внесённых в исходное дерево, и помогает программистам комбинировать и иным образом манипулировать изменениями, сделанными несколькими людьми или в разное время. GNU Arch — это система контроля версий с функциями, которые идеально подходят для проектов, характеризующихся широко распределённой разработкой, одновременной поддержкой нескольких выпусков и значительным объёмом разработки в ветвях. Последняя версия 1.3.5 (20 июля 2006 г.). С 2009 года официальным статусом GNU arch является амортизация, и применяются только исправления безопасности.

- CVS (Concurrent Version System, «Система Одновременных Версий») — система управления версиями. CVS хранит историю изменений определённого набора файлов, как правило, исходного кода программного обеспечения, и облегчает совместную работу группы людей (в первую очередь — программистов) над одним проектом. CVS была популярна в мире открытого ПО, распространяется на условиях GNU GPL.

  В настоящее время активная разработка системы прекращена (последняя версия выпущена в мае 2008 года), в исходный код вносятся только небольшие исправления.

  На данный момент CVS является устаревшей системой, потому что она имеет ряд недостатков, и имеются более молодые альтернативные системы управления версиями (Subversion, Git, Mercurial), свободные от большинства недостатков CVS.

- bazaar — распределённая система контроля версий, замена CVS. bazaar — это реализация GNU Arch на C, но более интуитивно понятная и удобная для пользователя. Он может заменить CVS и исправить многие недостатки этой системы. Bazaar — это распределённая система контроля версий. Она позволяет членам команды очень легко разветвлять и объединять исходный код. Распределённые системы контроля версий позволяют нескольким людям иметь свою собственную ветвь проекта и эффективно объединять код между ними. Это позволяет новым участникам получить доступ ко всем инструментам, которые ранее были ограничены только комментариями проекта.

- bzr — распределённая система контроля версий следующего поколения.

- olive — графический интерфейс к системе Bazaar. Olive стремится стать полнофункциональным графическим интерфейсом для Bazaar. Он представит все основные функции Bazaar в удобном графическом интерфейсе.

- Subversion, SVN — алтернатива CVS. Subversion — это параллельная система контроля версий, которая позволяет одному или нескольким пользователям совместно разрабатывать и поддерживать иерархию файлов и каталогов, сохраняя при этом историю всех изменений. Subversion хранит только различия между версиями, а не все файлы целиком. Subversion также ведет журнал того, кто, когда и почему произошли изменения. По сути, он делает то же самое, что и CVS (система одновременного управления версиями), но имеет значительные улучшения по сравнению с CVS и устраняет множество неудобств, с которыми сталкиваются пользователи CVS. Этот пакет содержит книгу по подрывной работе и файлы с информацией о дизайне.

- qsvn — графический клиент к системе Subversion, работает на всех платформах. QSvn is a graphical Subversion Client for Linux, UNIX, Windows and Mac OS X.

- crossvc — графическая подсистема контроля версий (интерфейс для CVS и SVN).

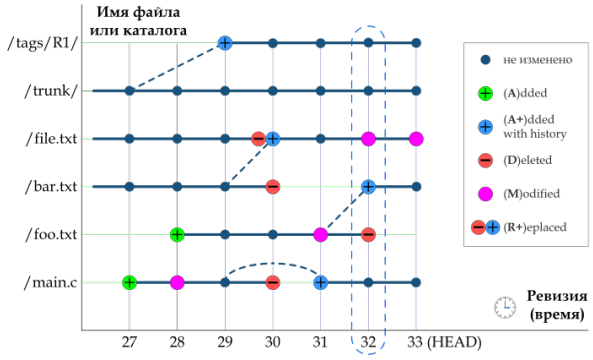


Рис. 7.1. Двумерное представление файловой системы в Subversion

- Darcs — David's Advanced Revision Control System, — система контроля версий, аналогичная CVS или arch. Darcs — это система контроля версий, аналогичная CVS или Arch. Это означает, что он отслеживает различные версии и ветки проекта, позволяет изменениям распространяться от одной ветки к другой. Darcs задуман как «продвинутая» система контроля версий. Darcs имеет две особенно отличительные особенности, которые отличаются от других систем контроля версий: 1) каждая копия исходного кода является полностью функциональной ветвью и 2) основа Darcs представляет собой последовательную и мощную теорию исправлений.

- Mercurial — кроссплатформенная распределённая система управления версиями, разработанная для эффективной работы с очень большими репозиториями кода. В первую очередь она является консольной программой. Репозитории Mercurial управляются при помощи утилиты командной строки `hg`, но есть и GUI—интерфейсы. Проекты, использующие Mercurial: Mozilla, OpenOffice.org, OpenJDK, NetBeans, OpenSolaris, ALSA, Xen, XINE, XEmacs.

- IBM Rational ClearCase — система управления версиями от IBM, широко используется компаниями, производящими программное обеспечение.

- Git.

3.1. Git

> ***Git*** *— распределённая система управления версиями файлов, ориентированная на работу с изменениями (а не с файлами), спроектированная как набор программ, специально разработанных с учётом их использования в скриптах.*

Это позволяет удобно создавать специализированные системы контроля версий на базе Git или пользовательские интерфейсы.

Проект был создан Линусом Торвальдсом для управления разработкой ядра Linux, первая версия выпущена 7 апреля 2005 года. На сегодняшний день его поддерживает Джунио Хамано. Программа является свободной и выпущена под лицензией GNU GPLv2.

Git предоставляет каждому разработчику локальную копию всей истории разработки, изменения копируются из одного репозитория в другой.

Самые значительные проекты, использующие Git: ядро Linux, Android, Drupal, Joomla!, Cairo, GNU Core Utilities, Mesa, Wine, Chromium, Compiz Fusion, FlightGear, jQuery, PHP, NASM, MediaWiki, Mono и некоторые дистрибутивы GNU/Linux (Debian, Alt-Linux, Archlinux, Fedora).

Git поддерживает быстрое разделение и слияние версий, включает инструменты для визуализации и навигации по нелинейной истории разработки.

Удалённый доступ к репозиториям Git обеспечивается git-daemon, SSH- или HTTP-сервером. TCP-сервис git-daemon входит в дистрибутив Git и является наряду с SSH наиболее распространённым и надёжным методом доступа. Метод доступа по HTTP, несмотря на ряд ограничений, очень популярен в контролируемых сетях, потому что позволяет использовать существующие конфигурации сетевых фильтров.

Архитектура

Ядро Git представляет собой набор утилит командной строки с параметрами. Все настройки хранятся в текстовых файлах конфигурации. Такая реализация делает Git легко портируемым на любую платформу и даёт возможность легко интегрировать Git в другие системы (в частности, создавать графические git-клиенты с любым желаемым интерфейсом).

Репозиторий Git представляет собой каталог файловой системы, в котором находятся файлы конфигурации репозитория; файлы журналов, хранящие операции, выполняемые над репозиторием; индекс, описывающий расположение файлов и хранилище, содержащее собственно файлы. Структура хранилища файлов не отражает реальную структуру хранящегося в репозитории файлового дерева, она ориентирована на повышение скорости выполнения операций с репозиторием. Когда ядро обрабатывает команду изменения (неважно, при локальных изменениях или при получении патча от другого узла), оно создаёт в хранилище новые файлы, соответствующие новым состояниям изменённых файлов. Существенно, что никакие операции не изменяют содержимого уже существующих в хранилище файлов.

По умолчанию репозиторий хранится в подкаталоге с названием `.git` в корневом каталоге рабочей копии дерева файлов, хранящегося в репозитории. Любое файловое дерево в системе можно превратить в репозиторий git, отдав команду создания репозитория из корневого каталога этого дерева (или указав корневой каталог в параметрах программы).

Репозиторий может быть импортирован с другого узла, доступного по сети. При импорте нового репозитория автоматически создаётся рабочая копия, соответствующая последнему зафиксированному состоянию импортируемого репозитория.

Нижний уровень git является т. н. файловой системой с адресацией по содержимому (content-addressed file system). Git содержит ряд команд по непосредственной манипуляции этим репозиторием на низком уровне. Эти команды не нужны при нормальной работе с git как с системой контроля версий, но нужны для реализа-

ции сложных операций (ремонт повреждённого репозитория и т. д.), а также дают возможность создать на базе репозитория git какое-то своё приложение.

Для каждого объекта в репозитории считается SHA-1 хэш, именно он становится именем файла, содержащего данный объект в директории `.git/objects`. Для оптимизации работы с файловыми системами, не использующими деревья для директорий, первый байт хэша становится именем поддиректории, а остальные — именем файла в ней, что снижает количество файлов в одной директории (ограничивающий фактор производительности на устаревших файловых системах).

Все ссылки на объекты репозитория, включая ссылки на один объект, находящийся внутри другого объекта, являются SHA-1 хэшами.

Кроме того, в репозитории существует директория `refs`, которая позволяет задать читаемые человеком имена для каких-то объектов git. В командах git оба вида ссылок — читаемые человеком из `refs`, и нижележащие SHA-1 — полностью взаимозаменяемы.

В классическом обычном сценарии в репозитории git есть три типа объектов — *файл*, *дерево* и **коммит** (commit). Файл есть какая-то версия какого-то пользовательского файла, дерево — совокупность файлов из разных поддиректорий.

> ***Коммит*** *— это дерево + некая дополнительная информация (например, родительский(е) коммит(ы), а также комментарий).*

В репозитории иногда производится сборка мусора, во время которой устаревшие файлы заменяются на «дельты» (Δ) между ними и актуальными файлами (актуальная версия файла хранится не-инкрементально, инкременты используются только для шагания назад), после чего данные «дельты» складываются в один большой файл, к которому строится индекс. Это снижает требования по месту на диске.

Репозиторий git бывает локальный и удалённый. Локальный репозиторий — это поддиректория `.git`, создаётся (в пустом виде) командой `git init` или (в непустом виде с немедленным копированием содержимого родительского удалённого репозитория и простановкой ссылки на родителя) командой `git clone`.

Практически все обычные операции с системой контроля версий, такие, как коммит и слияние, производятся только с локальным репозиторием. Удалённый репозиторий можно только синхронизировать с локальным как «вверх» (`push`), так и «вниз» (`pull`).

Наличие полностью всего репозитория проекта локально у каждого разработчика даёт git ряд преимуществ перед SVN. Так, например, все операции, кроме `push` и `pull`, можно осуществлять без наличия интернет-соединения.

Очень мощной возможностью git являются ветви, реализованные куда более полно, чем в SVN. Создать новую ветвь так же просто, как и совершить коммит. По сути, *ветвь git* есть не более чем имя, «навешенное» на некий коммит в репозитории (используется поддиректория `refs`). Коммит без создания новой ветви всего лишь передвигает эту ссылку на себя, а коммит с созданием ветви — оставляет старую ссылку на месте, но создаёт новую на новый коммит, и объявляет её текущей. Заменить локальные файлы на набор файлов из иной ветви, тем самым перейдя к работе с ней — также просто.

Также поддерживаются суб-репозитории с синхронизацией текущих ветвей в них.

Команда `push` передаёт все новые данные из локального репозитория в репозиторий удалённый. Для исполнения этой команды необходимо, чтобы удалённый репозиторий не имел новых коммитов в себя от других клиентов, иначе `push` завершается ошибкой, и придётся делать `pull` и слияние.

Команда `pull` — обратна команде `push`. В случае, если одна и та же ветвь имеет независимую историю развития от какого-то момента в локальной и в удалённой копии, `pull` немедленно приводит к слиянию.

Слияние в пределах разных файлов осуществляется автоматически (все это поведение настраивается), а в пределах одного файла — стандартным трёхпанельным сравнением файлов. После слияния нужно объявить конфликты как разрешённые.

Результатом всего этого является новое состояние в локальных файлах у того разработчика, который осуществил слияние. Ему нужно немедленно сделать коммит, при этом в данном объекте коммита в репозитории окажется информация о том, что коммит есть результат слияния двух ветвей и имеет два родительских коммита.

Git поддерживает и `rebase` — получение набора всех изменений в ветви A, с последующим их «накатом» на ветвь B. В результате ветвь B продвигается до состояния AB. В отличие от слияния, в истории ветви AB не останется никаких промежуточных коммитов ветви A (только история ветви B и запись о самом `rebase`, это упрощает интеграцию крупных и очень крупных проектов).

Git имеет временный локальный индекс файлов — промежуточное хранилище между собственно файлами и очередным коммитом (коммит делается только из этого индекса). С помощью этого индекса осуществляется добавление новых файлов (`git add` добавляет их в индекс, они попадут в следующий коммит), а также коммит НЕ ВСЕХ изменённых файлов (коммит делается только тем файлам, которым был сделан `git add`). После `git add` можно редактировать файл далее, получатся три копии одного и того же файла — последняя, в индексе (та, что была на момент `git add`), и в последнем коммите.

Имя ветви по умолчанию: *master*.

Имя удалённого репозитория по умолчанию, создаваемое `git clone` (типичной операции «взять имеющийся проект с какого-то сервера себе на машину»): *origin*.

Таким образом, в локальном репозитории всегда есть ветвь `master`, которая есть последний локальный коммит, и ветвь `origin/master`, которая есть последнее состояние удалённого репозитория на момент завершения исполнения последней команды `pull` или `push`.

Команда `fetch` (частичный `pull`) — берёт с удалённого сервера все изменения в `origin/master`, и переписывает их в локальный репозиторий, продвигая метку `origin/master`.

Если после этого `master` и `origin/master` разошлись в стороны, то необходимо сделать слияние, установив `master` на результат слияния (команда `pull` есть `fetch`+`merge`). Далее возможно сделать `push`, отправив результат слияния на сервер и установив на него `origin/master`.

В Windows версии (официальная Windows-версия называется mSysGit) используется пакет mSys, который является эмулятором POSIX-совместимой командной строки над Windows (производный от MinGW, примерный аналог Cygwin, но не

имеет присущей последнему проблемы с крайне примитивной и субоптимальной реализацией `fork()` без использования соответствующей возможности ядра Windows).

Под mSys перенесены все необходимые для git библиотеки и инструменты, а также сам git.

При работе с удалёнными репозиториями по протоколу SSL будет использоваться хранилище сертификатов из mSys, а НЕ из Windows.

Существует немало графических оболочек для Git, например, TortoiseGit. Все они основаны на mSysGit, требуют его установки на машину и работают через исполнение его командных строк. Реализация компании Atlassian под названием SourceTree содержит mSysGit внутри себя, и устанавливает его в очень глубокую поддиректорию на Windows-машине, что затрудняет добавление нужных SSL-сертификатов.

Возможности и ограничения

Git использует сеть только для операций обмена с удалёнными репозиториями.

Возможно использование следующих протоколов:

- `git://` — открытый протокол, требующий наличие на сервере запущенного демона (поставляется вместе с `git`). Протокол не имеет средств аутентификации пользователей.

- `ssh://` — использует аутентификацию пользователей с помощью пар ключей, а также встроенный в UNIX-систему «основной» SSH-сервер (`sshd`). Со стороны сервера требуется создание аккаунтов у которых будет некая команда git.

- `http(s)://` — использует внутри себя инструмент `curl` (для Windows — поставляется вместе с `git`), и его возможности HTTP-аутентификации, как и его поддержку SSL и сертификатов.

В последнем случае необходимо наличие на сервере некоего ПО веб-приложения, которое будет исполнять роль прослойки между командами `git` на сервере и HTTP-сервером. Такие прослойки существуют как под Linux, так и под Windows (например, WebGitNet, разработанный на ASP.NET MVC 4).

Кроме поддержки серверной стороны команд `push` и `pull`, такие веб-приложения могут также давать доступ только на чтение к репозиторию через веб-браузер.

Преимущества и недостатки `git` по сравнению с централизованными системами управления версиями (такими как, например, Subversion) типичны для любой распределённой системы

Если же сравнивать git с «родственными» ей распределёнными системами, можно отметить, что Git изначально идеологически ориентирован на работу с изменениями, а не с файлами, «единицей обработки» для него является набор изменений, или патч. Эта особенность прослеживается как в структуре самой системы (в частности — в структуре репозитория), так и в принципах построения команд; она отражается на производительности системы в различных вариантах её использования и на достоинствах и недостатках git по сравнению с другими динамическими VCS.

Недостатки Git:

- Отсутствие сквозной нумерации коммитов монотонно непрерывно возрастающими целыми числами. Во многих проектах используется автоматическое получение номера этой версии (например, командой `svnversion`), построение `.H` файла на основе этого числа, и далее его использование при создании штампа версии исполняемого файла, некоторых вшитых в него строк и так далее.

- Отсутствие переносимой на другие операционные системы поддержки путей в кодировке Unicode в Microsoft Windows (для версий msysgit до 1.8.1). Если путь содержит символы, отличные от ANSI, то их поддержка из командной строки требует специфических настроек, которые не гарантируют правильного отображения файловых имён при пользовании тем же репозиторием из других ОС. Одним из способов решения проблемы для git 1.7 является использование специально пропатченного консольного клиента. Другой вариант — использование графических утилит, работающих напрямую через API, таких как TortoiseGit.

- Некоторое неудобство для пользователей, переходящих с других VCS. Команды git, ориентированные на наборы изменений, а не на файлы, могут вызвать недоумение у пользователей, привыкших к файл-ориентированным VCS, таким как SVN. Например, команда «add», которая в большинстве систем управления версиями производит добавление файла к проекту, в git подготавливает к фиксации сделанные в файлах изменения. При этом сохраняется не патч, описывающий изменения, а новая версия целевого файла.

- Использование для идентификации ревизий хэшей SHA-1, что приводит к необходимости оперировать длинными строками вместо коротких номеров версий, как во многих других системах (хотя в командах допускается использование неполных хэш-строк).

- Бóльшие накладные расходы при работе с проектами, в которых делаются многочисленные несвязанные между собой изменения файлов. При работе в таком режиме размеры наборов изменений становятся достаточно велики и происходит быстрый рост объёма репозиториев.

- Бóльшие затраты времени, по сравнению с файл-ориентированными системами, на формирование истории конкретного файла, истории правок конкретного пользователя, поиска изменений, относящихся к заданному месту определённого файла.

- Отсутствие отдельной команды переименования/перемещения файла, которая отображалась бы в истории как соответствующее единое действие. Существующий скрипт `git mv` фактически выполняет переименование, копирование файла и удаление его на старом месте, что требует специального анализа для определения, что в действительности файл был просто перенесён (этот анализ выполняется автоматически командами просмотра истории). Однако, учитывая тот факт, что наличие специальной команды для переименования/перемещения файлов технически не вынуждает пользователя использовать именно её (и, как

следствие, в этом случае возможны разрывы в истории), поведение `git` может считаться преимуществом.

- Система работает только с файлами и их содержимым, и не отслеживает пустые каталоги.

В ряде публикаций, относящихся преимущественно к 2005—2008 годам можно встретить также нарекания в отношении документации Git, отсутствия удобной windows-версии и удобных графических клиентов. В настоящее время эта критика неактуальна: существует версия git на основе MinGW («родная» сборка под Windows), и несколько высококачественных графических клиентов для различных операционных систем, в частности, под Windows имеется клиент TortoiseGit, идеологически очень близкий к широко распространённому TortoiseSVN — клиенту SVN, встраиваемому в оболочку Windows.

Преимущества Git перед другими DVCS:

- Высокая производительность.
- Развитые средства интеграции с другими VCS, в частности, с CVS, SVN и Mercurial. Помимо разнонаправленных конвертеров репозиториев, имеющиеся в комплекте программные средства позволяют разработчикам использовать git при размещении центрального репозитория в SVN или CVS, кроме того, git может имитировать CVS-сервер, обеспечивая работу через клиентские приложения и поддержку в средах разработки, специально не поддерживающих git.
- Продуманная система команд, позволяющая удобно встраивать git в скрипты.
- Качественный веб-интерфейс «из коробки».
- Репозитории git могут распространяться и обновляться общесистемными файловыми утилитами архивации и обновления, такими как `rsync`, благодаря тому, что фиксации изменений и синхронизации не меняют существующие файлы с данными, а только добавляют новые (за исключением некоторых служебных файлов, которые могут быть автоматически обновлены с помощью имеющихся в составе системы утилит). Для раздачи репозитория по сети достаточно любого веб-сервера.

Графические интерфейсы

- SmartGit — кроссплатформенный интерфейс для Git на *Java*.
- gitk — простая и быстрая программа, написана на Tcl/Tk, распространяется с самим Git.
- QGit — графический клиент, написан с использованием Qt, во многом схож с gitk, но несколько отличается набором возможностей. В настоящее время существуют реализации на Qt3 и Qt4.

- Giggle — вариант на Gtk+.
- gitg — ещё один интерфейс для gtk+/GNOME
- Git Extensions — кроссплатформенный вариант на `.NET`.
- TortoiseGit — интерфейс, реализованный как расширение для проводника Windows.

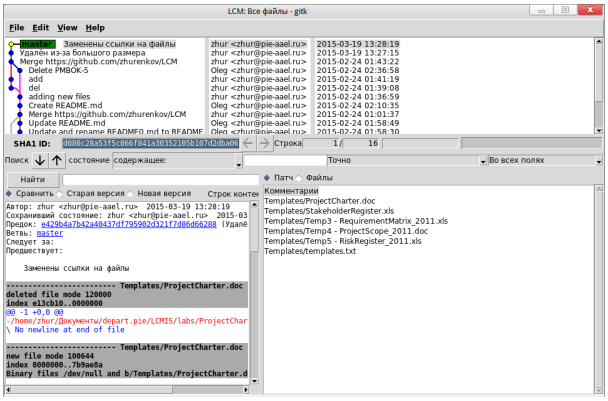


Рис. 7.2. Интерфейс gitk

- SourceTree — бесплатный git клиент для Windows и Mac.
- Git-cola — кроссплатформенный интерфейс на *Python*.
- GitX — оболочка для Mac OS X с интерфейсом *Cocoa*, интерфейс схож с gitk.
- Gitti — оболочка для Mac OS X с интерфейсом *Cocoa*.
- Gitbox — оболочка для Mac OS X с интерфейсом *Cocoa*.
- Github-клиент.
- StGit — написанная на *Python* система управления коллекцией патчей (Catalin Marinas).

Фронтенды для Web

- Gitlab
- Gitblit
- Gerrit
- Gitweb
- cgit
- ViewGit

- GitList
- git-php
- wit
- gitarella

Сервисы, предоставляющие хостинг для git-репозиториев

- GitHub — широко известная веб-система коллективной работы на основе Git, построенная как облачный сервис.
- SourceForge — система совместной разработки компьютерных программ, один из самых больших в мире веб-сайтов для разработчиков открытого программного обеспечения. Программное обеспечение SourceForge разрабатывается и продаётся компанией CollabNet.
- Codebase — *базовый код*, коллекция исходных кодов ПО для совместного использования с применением Subversion, Git, Mercurial.
- Gitorious — Git-хостинг и программное обеспечение для совместной работы, которое Вы можете установить самостоятельно.
- Google Code — сервис для разработчиков ПО, в первую очередь, связанных с продуктами компании Google. Однако сервис предоставляется и для сторонних разработок с открытым кодом (open source). Есть возможность выбрать из 9 возможных лицензий: Apache, Artistic, BSD, GPLv2, GPLv3, LGPL, MIT, MPL и EPL.
- Bitbucket — веб-сервис для хостинга проектов и их совместной разработки, основанный на системе контроля версий Git. Ранее была поддержка ещё и Mercurial, но с 1 июня 2020 года из BitBucket Cloud и API проекта поддержка Mercurial полностью удалена. По назначению и предлагаемым функциям Bitbucket аналогичен GitHub (однако Bitbucket предоставляет бесплатные «закрытые» репозитории, в отличие от GitHub). Слоган сервиса — «We're here to serve» (Мы здесь, чтобы служить).

Основы работы с Git.
Git Book.

Глава 8

Управление документацией

1. Основные понятия

Качество программного обеспечения, наряду с другими факторами, определяется качеством пакета документов, сопровождающих ПО.

Когда проектная команда получает в той или иной форме задание на разработку, перед ними встаёт ряд вопросов:

- Что должно быть сделано, кроме собственно программы?
- Что и как должно быть оформлено в виде документации?
- Что передавать пользователям, а что — службе сопровождения?
- Как управлять всем этим процессом?
- Что должно входить в само задание на программирование?
- ...

Прошло много лет с появления профессии программист, сейчас программирование происходит в среде совершенно новых технологий, многие программисты, работая в стиле drag-and-drop в low-code среде могут годами не видеть текст своих программ. Это не значит, что исчезла необходимость в их документировании.

Для профессиональной разработки ПО существуют обязательные и рекомендательные для применения стандарты (особенно остро стоит этот вопрос, когда разработка выполняется по заказу государственной организации).

> ***Документация на программное обеспечение*** *— печатные руководства пользователя, диалоговая (оперативная) документация и справочный текст, описывающие, как пользоваться программным продуктом.*

> ***Документ*** *— элемент документации: целевая информация, предназначенная для конкретной аудитории, размещённая на конкретном носителе (например, в книге, на диске, в краткой справочной карте) в заданном формате.*

Программный документ *— документ, содержащий в зависимости от назначения, данные, необходимые для разработки, производства, эксплуатации, сопровождения программы или программного средства.*

Управление документацией программного проекта требует значительных организационных навыков, поскольку документацию можно уподобить сложному, живому организму, который подвержен многочисленным изменениям. Эти изменения очень часто вносятся одновременно и самыми разными людьми. Создание хорошей документации во многом похоже на написание хорошего программного кода.

Управление документацией ориентировано на поддержание её **полноты** и **согласованности**.

2. Полнота документации

Полнота *означает, что комплект документации должен охватывать весь жизненный цикл ПО.*

Разработка программного обеспечения поддерживается документацией. Когда мы смотрим на документированный программный код, его значение становится намного понятнее. Чтобы документация была понятна всем, она должна соответствовать принятым стандартам.

Рассмотрим пример описания полного комплекта документов из набора IEEE, как хорошо проработанный комплект, оказавший решающее влияние на все остальные стандарты.

Описание полного комплекта документов из набора IEEE:

1. *План верификации и валидации ПО* (Software verification and validation plan). Этот план определяет содержание и последовательность проверки шагов проекта и самого продукта на его соответствие поставленным требованиям. Часто валидацию и верификацию осуществляют сторонние организации, тогда экспертиза называется независимой (Independent V&V, IV&V).

2. *План обеспечения качества ПО* (Software quality assurance plan). Этот план определяет действия для достижения проектом установленного уровня качества.

3. *План управления конфигурацией ПО* (Software configuration management plan). Он указывает порядок хранения документов, программного кода и их версий, а также устанавливает между ними взаимное соответствие. Нельзя начинать работу без такого плана, так как любой документ обречён на изменения, а мы должны знать, как управлять этими изменениями для поддержания равновесия.

4. *План управления программным проектом* (Software project management plan). Этот план определяет, каким образом руководить проектом.

5. *Спецификация требований к ПО* (Software requirements specification). Этот документ задаёт требования к продукту и является частью контракта, заключаемого между заказчиком и фирмой-разработчиком.

6. *Описание проектирования ПО* (Software design description). Описывает архитектуру и детали проектирования продукта, обычно с использованием графических диаграмм.

7. *Документация тестирования ПО* (Software test documentation). Этот документ описывает содержание процесса тестирования программных модулей и всего ПО.

Иногда в проектах привлекается дополнительная документация, разработанная внутри организации разработчика.

3. Согласованность документации

> ***Согласованность*** *означает, что комплект документов не содержит внутренних противоречий.*

Проблема в том, что когда этот комплект достаточно велик, трудно избежать появления в нём взаимоисключающих утверждений.

Для обеспечения непротиворечивости следует разместить описание некоторого факта только в одном документе.

Положим, для банковской системы, работающей в веб-среде, задано требование «количество попыток аутентификации в системе равно трём». Это требование документируется в *Спецификации требований к ПО*. Хотелось бы отразить это требование во всех документах, где оно используется. К чему это приведет? Необходимо отслеживать изменение требования во всех этих местах. Сделать это в условиях дефицита времени крайне сложно. Для этого следует применять *механизм гиперссылок* (ссылки из всех зависимых документов на спецификации требований).

Большинство документов зависят друг от друга. Так формируется многоуровневая структура документации. Если появилась новая версия какого-то документа, нужно оперативно отразить это в зависимых документах, автоматически отразив эту информацию через гиперссылки. Иначе говоря, большинство документов являются живыми объектами, о которых надо заботиться.

Для небольших организаций довольно трудно синхронизировать документооборот. Здесь важно обучить сотрудников применению техники гиперссылок и *убедить* их в необходимости и важности записи одного факта только в одном месте, а также регулярного обновления документации.

Механизм гиперссылок имеется в языках разметки (X)HTML, XML (и др., основанных на нём), LaTeX, wiki, bbCode, и др.

4. Автоматизация процесса документирования

Наиболее полной и согласованной получается документация при автоматизации процесса документирования. *Автоматизированное документирование* — одна из самых важных сторон разработки программного кода. Идея автоматизированного документирования заключается в том, что в исходном коде пишутся комментарии в специальном формате. Далее запускается специальная утилита, которая разбирает исходный код, вытаскивая комментарии и объединяя их в единый структурированный документ. Таким образом, с плеч программиста снимается вся черновая работа по созданию программной документации. Кроме того, такой способ документирования позволяет более строго контролировать версии программного кода и документации, поскольку при изменении программного кода программист должен изменять расположенные рядом комментарии.

Для документирования программного кода используют *программы-генераторы*, сканирующие файлы проекта в поисках комментариев. Без комментариев даже небольшую программу трудно понять не только постороннему программисту, но и самому автору.

Многие среды разработки (Zend, Visual Studio, Netbeans) уже включают в себя средства документирования, но никакого общего стандарта до сих пор нет. В своём большинстве, цель генераторов — составить нечто типа энциклопедии всех функций, классов, объектов, иерархий классов, связей между файлами, используемые компоненты и третьи программные продукты.

Генераторы можно условно разделить на:

- Универсальные
  - — Doxygen.
  - — Natural Docs. На выходе — HTML.
  - — AsciiDoc (GNU GPLv2). На выходе — HTML(4,5), XHTML, DocBook, LaTeX, PDF, EPUB, DVI, PS, man, HTML Help, txt.
  - — Doc-O-Matic — для C/C++, C#, Delphi, VB.NET, IDL, Java, PHP, JavaScript, ASPX, JSP, MatLab. Doc-O-Matic поставляется с базами данных ссылок для Microsoft Visual Studio 2010, 2008, 2005, MSDN, Embarcadero RAD Studio XE, 2010, а также предыдущих версий Delphi и C ++ Builder для CodeGear и Borland Delphi. V7 — $49.

  Специфичные
  - — javadoc (Free) — для Java.
  - — phpDocumentor (Open Source) — для PHP.
  - — Docutils (Open Source) — для Python.
  - — NDoc, Sandcastle, GhostDoc — для .NET.
  - — PasDoc (GNU GPLv2) — для Pascal (Object Pascal). На выходе — HTML, HtmlHelp, LaTeX, ...

Генераторы, как правило, по-своему хранят данные и комментарии, завися от языка, однако последние тенденции, например в Visual Studio, показывают популяризацию XML. *DocBook* — одна из предлагаемых универсальных схем.

4.1. Doxygen

Doxygen — мощная и распространённая утилита для документирования программного кода, написанного на C, C++, Objective-C, Java, Python, IDL (Corba и Microsoft) и некоторых версий PHP, C#, и D. Распространяется по лицензии GNU GPL.

Doxygen позволяет из комментариев кода, записанных в специальном формате, генерировать документы HTML, LaTeX, RTF, PDF с ссылками, диаграммами классов и прочими удобствами.

Выходная документация представляет собой файлы в формате HTML, XML, LaTeX (в частности, для дальнейшего создания PDF-файлов), RTF, справочного руководства в стиле кроссплатформенной библиотеки Qt, справочного руководства в стиле Unix — man и т. д.

Даже, если не оформлять специальным образом исходные тексты, Doxygen сгенерирует из них замечательные справочники программного интерфейса приложения. В выходной документ могут быть включены разделы с описанием используемых пространств имён, классов, файлов и их указатели на соответствующие разделы. В случае использования пакета визуализации графов Graphviz, описание классов может сопровождаться диаграммами UML.

Комментарии для документации

Существует два вида комментариев:

Комментарии для автоматизированного документирования исходного кода. Эти комментарии оформляются с помощью специальных форматов (doxygen, javadoc, MS XML Documentation Comments).

Комментарии исходного кода — сопровождающие исходный код, объясняющие его работу, раскрывающие непонятные и сомнительные моменты.

В отличие от комментариев для автоматизированного документирования, эти комментарии используются в любом программном коде. *Стоимость владения* исходным кодом уменьшается прямо пропорционально количеству и качеству таких комментариев. Это связано с тем, что хорошо структурированный и комментированный код легче поддерживается и сопровождается. То есть программистам просто требуется меньше времени для того, чтобы разобраться с ним. Для анализа качества исходного кода можно использовать специализированные инструменты (как например, Borland Together ControlCenter), которые подсчитывают различные метрики кода, описывающие его качество.

В Doxygen существуют несколько вариантов комментариев для автоматизированного документирования:

1. блочный стиль JavaDoc

```
/**
комментарий
*/
```

2. блочный стиль Qt

```
/*!
комментарий
*/
```

3. строчный стиль C++

```
/// комментарий
```

4. строчный стиль C++

```
//! комментарий
```

Всё, что написано внутри комментариев, оформленных таким образом, выносится в документацию. Для единообразия лучше использовать `///` для коротких комментариев и `/** */` для длинных.

Служебные слова

В Doxygen есть служебные слова, которые задают правила отображения частей документации (выделение в отдельный раздел, специальное форматирование и шрифт для выделения примера кода, вынесение текста на заглавную страницу и т. п.).

Автор — `@author` *имя*

Текст на главной странице — `@mainpage`

Создание новой страницы — `@page` *идентификатор название*

Пример .6. `@page pagereq Требования к системе`

Создание раздела — `@section` *идентификатор название*

Пример .7. `@section common Общие требования`

Вставка примера кода — `@code` текст программы `@endcode`

Заметка — `@note`

Предупреждение — `@warning`

Описание к файлу — `@file`

Краткое описание — `@brief` Можно настроить файл опций таким образом, чтобы в краткое описание выносилось первое предложение до точки подробного описания (по умолчанию описание считается подробным). Для этого нужно включить опцию JAVADOC_AUTOBRIEF — в утилите `doxywizard` она находится на первой странице.

Подробное описание — `@full`

Именованная группа — `@name` *имя*. После этого группа в скобках @{ ... @}

Внутри парного тега `\f` можно задавать формулу в нотации LaTeX. Этот тег будет проигнорирован, если в качестве выходного формата используется не LaTeX.

Пример .8 (формулы в Doxygen). `Расстояние между \f$(x_1,y_1)\f$ и \f$(x_2,y_2)\f$ вычисляется, как \f$\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}\f$.`

Расстояние между (x_1, y_1) и (x_2, y_2) вычисляется, как $\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}$.

Документировать следует классы, члены классов и функции со всеми параметрами. Кроме того, для отдельных модулей в целом следует документировать, как ими пользоваться.

Документирование функций `@function` *имя-функции*

Пример .9 (функции в Doxygen).
```
  /**
    @function MyFunction
    Тестовая функция, имеет два входных параметра.
@param in_Parameter1 --- параметр, отвечающий за что-то
@param out_Parameter2 --- параметр, отвечающий за что-то ещё
@return true если всё хорошо, false если всё плохо
    */

bool MyFunction(int in_Parameter1, std::string out_Parameter2);
```

Документирование классов `@class` *имя-класса*

Пример .10 (документирование класса). Тестовый класс для примера документирования кода содержит конструктор, один член данных и один метод.

```
  ///@class MyClass
class MyClass : public OurClass {
public:
  /// Конструктор по умолчанию
  MyClass();
/// Метод, делающий что-то. Он делает это таким-то образом.
/// @param in_iParameter1 --- параметр, отвечающий за что-то
/// @param out_sParameter2 --- параметр, отвечающий за что-то ещё
/// @return true если всё хорошо, false если всё плохо
  bool DoSmthng(int in_iParameter1, std::string out_sParameter2);
private:
  /// этот член хранит в себе какую-то информацию
  int m_iParameter1;
  /// этот член тоже хранит в себе какую-то информацию
  std::string m_sParameter2; }
```

Пример .11 (классы и группировка). Таким образом можно группировать методы класса:

```
///@class MyClass
class MyClass : public OurClass {
public:
  /// @name Конструктор/Деструктор
  /// @{
  MyClass();
  virtual ~MyClass();
  /// @}
  /// @name Установка параметров
  /// @{
  /// Устанавливает параметр1
  SetParameter1(int);
  /// Возвращает параметр1
  int GetParameter1();
  /// @}
  /// Делает ещё что-то
  void DoSmthng();
private:
/// этот член хранит в себе какую-то информацию
int m_iParameter1; }
```

Синтаксис имеет следующий вид:

1. Для генерации шаблона конфигурационного файла

   ```
   doxygen [-s] -g [configName]
   ```

   Если указано '`-s`', комментарии в файле конфигурации будут опущены. Если configName не указан `Doxyfile` будет использоваться по умолчанию.

   Если вместо `configName` указать '`-`', то doxygen будет писать в стандартный вывод.

2. Для обновления шаблона конфигурационного файла

   ```
   doxygen [-s] -u [configName]
   ```

3. Генерация документации на основе конфигурационного файла

   ```
   doxygen [configName]
   ```

   Если вместо `configName` указать '`-`', то doxygen будет читать из стандартного ввода.

6. Для создания файла шаблона управления макетом при генерации документации

   ```
   doxygen -l layoutFileName.xml
   ```

7. Для генерации файла документации с заданным расширением (на примере `.rtf`)

```
RTF:   doxygen -e rtf extensionsFile
```

8. Для создания шаблона стиля для RTF, HTML или LaTeX

```
RTF:   doxygen -w rtf styleSheetFile
HTML:  doxygen -w html headerF footerF styleSheetF [configFile]
LaTeX: doxygen -w latex headerF footerF styleSheetF [configFile]
```

Заголовок (`headerF`) для LaTeX'а должен содержать всё до первой главы, а `footerF` — всё после последней главы.

Файл настроек можно редактировать в любом текстовом редакторе или с помощью специальных утилит с графическим интерфейсом. Одна из них, Doxywizard, поставляется вместе с Doxygen.

Схема работы Doxygen (совместно с Doxywizard) представлена на рис. 8.1.

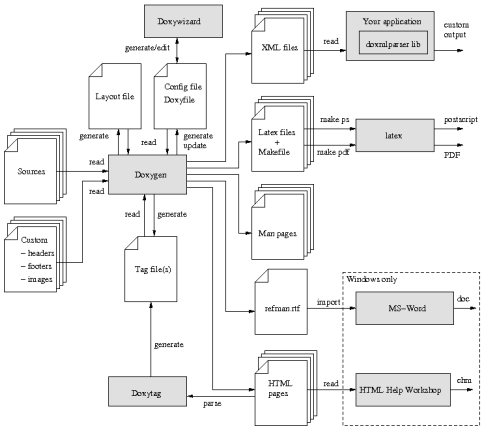


Рис. 8.1. Информационные потоки Doxygen

Глава 9

Управление рисками

1. Основные понятия

В ИТ риск понимается, в обычном смысле, как *возможность опасности, неудачи.*

> ***Риск*** *— следствие влияния неопределённости на достижение поставленных целей. (ГОСТ Р 51897-2011 «Менеджмент риска. Термины и определения»)*

К этому определению необходимо сделать несколько комментариев.

> ***Неопределённость*** *— это состояние, полного или частичного отсутствия информации относительно понимания или знания события, его последствий или вероятности.*

> ***Влияние неопределённости*** *— это отклонение от ожидаемого результата (целей) с позитивными или негативными последствиями.*

Поэтому риск часто характеризуют путём описания возможного события и его последствий или их сочетания.

> ***Событие*** *— возникновение или изменение специфического набора условий. (ГОСТ Р 51897-2011)*

> ***Последствие*** *— результат события. Событие может привести к ряду последствий. (ГОСТ Р 51897-2011)*

Суммируя вышеприведённые определения можно сказать следующее. **Риск** — это следствие вероятности возникновения события, которое окажет влияние на достижение поставленных целей.

Здесь важно учесть три обстоятельства:

1. Имеется в виду будущее событие.

2. Событие может заключаться в том, что какое-то явление не имело места.

3. Событие может быть случайным, т. е. заранее неизвестно, произойдёт оно или нет, но тем не менее, есть основания полагать, что оно может произойти.

Кроме того, что результатом события может быть одно или более последствий, первоначальные последствия могут усиливаться за счёт эффекта «домино».

Последствие может иметь не только отрицательные, но и положительные влияния на цели.

Последствия могут быть ранжированы от позитивных до негативных. Однако применительно к аспектам безопасности всегда рассматриваются именно негативные последствия.

С учётом рассмотренных определений риски характеризуются двумя величинами:

1. вероятность P, которая характеризует наступление события (рискового события);

2. степень влияния возможных последствий L.

Влияние риска на проект вычисляют по выражению:

$$E = P \times L, \tag{9.1}$$

где E — подверженность риску (Risk Exposure). P — вероятность негативного события, неудовлетворительного результата (Unsatisfactory Outcome); L — потеря при негативном событии.

При разработке программного продукта неудовлетворительным результатом может быть: превышение бюджета, низкая надёжность, неправильное функционирование и т. д.

> ***Источник риска (risk source)*** *— объект или деятельность, которые самостоятельно или в комбинации с другими обладают возможностью вызывать повышение риска. (ГОСТ Р 51897-2011)*

Источник риска может быть материальным или нематериальным.

В области информационной безопасности часто употребляется термин «угроза», например, ГОСТ Р ИСО/МЭК 27000-2012 определяет угрозу (threat) как «возможная причина нежелательного инцидента, который может нанести ущерб системе или организации».

> ***Уязвимость*** *— внутренние свойства или слабые места объекта, вызывающие его чувствительность к источнику риска, что может привести к реализации события и его последствий. (ГОСТ Р 51897-2011)*

Уязвимость объекта может позволить источнику риска спровоцировать некоторое событие, которое может иметь одно или несколько последствий. Событие может произойти или не произойти, и даже если оно произойдёт, последствия от данного события могут возникнуть или не возникнуть, следовательно появляется неопределённость, которая может повлиять на достижение поставленных целей, т. е. возникает риск.

Оценка рисков позволяет организации учитывать, в какой степени потенциальные события могут оказать влияние на достижение её целей.

Иногда эти две величины объединяют в один показатель — **значимость риска**.

> *Совокупность факторов, по сопоставлению с которыми оценивают значимость риска, называют* ***критерием риска*** *(ГОСТ Р 51897-2011).*

> ***Идентификация риска*** *— процесс обнаружения, распознавания и описания рисков. (ГОСТ Р 51897-2011).*

Идентификация включает распознавание источников риска, событий, их причин и возможных последствий. Введём ещё несколько важных определений.

> ***Управление рисками, риск-менеджмент (risk management)*** *— процесс принятия и выполнения управленческих решений, направленных на снижение вероятности возникновения неблагоприятного результата и минимизацию возможных потерь, вызванных им.*

> *Процесс* ***менеджмента риска (управления риском)*** *— взаимосвязанные действия по обмену информацией, консультациям, установлению целей, области применения, идентификации, исследованию, оценке, обработке, мониторингу и анализу риска, выполняемые в соответствии с политикой, процедурами и методами менеджмента организации.*

Управление рисками включает:

1. *Идентификация рисков* — выявление рисков в проекте.
2. *Анализ рисков* — оценка вероятности и величины потери по каждому риску.
3. *Ранжирование рисков* — упорядочивание рисков по степени их влияния.
4. *Планирование управления рисками* — подготовка к работе с каждым риском.
5. *Разрешение рисков* — устранение или разрешение рисков.
6. *Наблюдение рисков* — отслеживание динамики рисков, выполнение корректирующих действий.

Первые три действия относят к этапу *оценивания рисков*, последние три — к этапу *контроля рисков*.

2. Идентификация рисков

Классификация ИТ-рисков по источникам рисков:

Технологии — программное обеспечение, средства вычислительной техники и оборудование.

Процессы — отсутствие процедур и политик, приводящее к неправильному использованию или проблемам в использовании ИТ-систем.

Люди — человеческие ошибки, увольнение ключевых сотрудников.

Внешние — некачественное предоставления внешних услуг (электроэнергии, интернета и т.п.), законодательные инициативы, геополитика.

Для эффективного управления этими рисками методология управления ИТ предполагает встраивание управления рисками в систему управления ИТ-услугами. При этом для каждой группы рисков предлагаются свои инструменты управления рисками.

Классификация рисков по их природе (области, в которой они проявляются):

Риски конфиденциальности и целостности — потеря или изменение важных данных, несанкционированный доступ, раскрытие конфиденциальной информации, компрометация данных в результате мошенничества или воровства.

Риски доступности — простой систем, большое время восстановления ИС после отказа или из-за ошибки работника.

Риски производительности — недостаточная производительность ИТ-сервиса, которая может привести к потерям в бизнесе.

Риски неэффективности — превышение согласованного бюджета на создание/сопровождение ИС.

Риски несоответствия нормативным требованиям — штрафные санкции, судебные издержки или ущерб репутации в связи с несоблюдением законодательных норм.

В проектной деятельности выделяют три основные категории риска: **проектный риск**, **технический риск**, **коммерческий риск**.

Источниками **проектного риска** являются:

- выбор бюджета, плана, человеческих ресурсов для программного проекта;
- формирование требований к программному продукту;
- сложность, размер и структура программного проекта;
- методика взаимодействия с заказчиком.

К источникам **технического риска** относят:

- трудности проектирования, конструирования, формирования интерфейса, тестирования и сопровождения;
- неточность спецификаций;
- техническая неопределённость или отсталость принятого решения.

Главная причина **технического риска** — реальная сложность проблем выше предполагаемой сложности.

Источниками **коммерческого риска** являются:

- создание продукта, не требующегося на рынке;
- создание продукта, опережающего требования рынка (или отстающего от них);
- потеря финансирования.

Лучший способ идентификации рисков — использование *проверочных списков рисков*. Проверочный список десяти главных рисков программного проекта:

1. Дефицит персонала.
2. Нереальные расписание и бюджет.
3. Разработка неправильных функций и характеристик.
4. Разработка неправильного пользовательского интерфейса.
5. Слишком дорогое обрамление.
6. Интенсивный поток изменения требований.
7. Дефицит поставляемых компонентов.
8. Недостатки в задачах, разрабатываемых смежниками.
9. Дефицит производительности.
10. Деформирование научных возможностей.

Каждый элемент списка снабжается комментарием — набором методик для предотвращения источника риска. После идентификации элементов риска следует количественно оценить их влияние на программный проект, решить вопросы о возможных потерях. Эти вопросы решаются на этапе анализа рисков.

В практической работе можно использовать ту классификацию рисков, которая удобна для конкретного проекта, при этом важно включить в него все существенные риски, которые могут повлиять на достижение поставленных целей.

Рисками также можно управлять для максимизации выгоды. Современная методология управления рисками предполагает не только управление с целью минимизации негативных последствий, но и рассматривает риски как возможности.

Примером «позитивного» риска может служить включение в состав «коробочного» продукта компанией-производителем новых возможностей (конечно, данный

Таблица 9.1.

Оценка влияния риска

R_i, i-й риск	P_i, %	L_i	E_i
1. Отслеживание опасного условия как безопасного	5	9	0,45
2. Отказоустойчивость недопустимо снижает производительность	4–8	7	0,28–0,56
3. Критическая программная ошибка	3-5	10	0,3–0,5
4. Потеря ключевых данных	3–5	8	0,24–0,4
5. Плохой интерфейс пользователя снижает эффективность работы	6	5	0,3
6. Аппаратные задержки срывают планирование	6	4	0,24
7. Отслеживание безопасного условия как опасного	5	3	0,15
8. Ошибки преобразования данных приводят к избыточным вычислениям	8	1	0,08
9. Дефицит процессорной памяти	1	7	0,07
10. СУБД теряет данные	2	2	0,04

продукт вы уже используете). Новшества зачастую анонсируют или сами производители, или социальные сообщества, поэтому возможно заранее подготовить свои бизнес-процессы к изменениям, использовать передовое ИТ-решение с большей выгодой.

В качестве примера негативного риска разберём управление операционным риском, связанным с неспособностью восстановления ИТ-системы после сбоев в установленное время (нарушение SLA) в организации, в которой внедрён процесс управления рисками, в том числе определены цели, область применения, ответственные и другие составляющие контекста риск-менеджмента.

При идентификации риска проводится определение перечня и описания элементов риска. Наиболее эффективным способом в данном случае является привлечение для оценки экспертов в предметной области. Эксперты должны зафиксировать, что «мы, возможно, не сможем восстановить информацию после сбоя за оговоренное в SLA время из хранилища данных из-за большого объёма».

3. Анализ рисков

В результате идентификации формируется список рисков, специфичных для данного проекта.

В ходе анализа оценивается вероятность возникновения P_i и величина потери L_i для каждого выявленного R_i (i-го риска). В результате вычисляется влияние i-го показателя риска на проект E_i.

Вероятности определяются с помощью экспертных оценок или на основе статистики, накопленной за предыдущие разработки. Итоги анализа сводятся в таблицу (см. табл. 9.1).

4. Ранжирование рисков

Ранжирование заключается в назначении каждому риску приоритета, который пропорционален влиянию риска на проект. Это позволяет выделить категории рисков и определить наиболее важные из них. Например, представленные в табл. 9.1 риски упорядочены по их приоритету.

Для больших проектов количество элементов списка рисков может быть очень велико (30–40 элементов). В этом случае управление рисками затруднено. К списку рисков применяют *принцип Парето 80/20*: 80% всего проектного риска приходится на долю 20% от общего количества элементов списка. В ходе ранжирования определяют эти 20% элементов списка, — *существенные риски*. В дальнейшем учитывается влияние только *существенных рисков*.

5. Планирование управления рисками

Цель планирования — сформировать набор функций управления каждым риском.

В планировании используют понятие **эталонного уровня риска**. Обычно выбирают три **эталонных уровня риска**: *превышение стоимости*, *срыв планирования*, *упадок производительности*. Они могут быть причиной прекращения проекта. Если комбинация проблем, создающих риск, станет причиной превышения любого из этих уровней, работа будет остановлена. В фазовом пространстве риска эталонному уровню риска соответствует эталонная точка. В эталонной точке решения «продолжать проект» и «прекратить проект» имеют одинаковую силу. На рис. 9.1 показана кривая останова, составленная из эталонных точек.

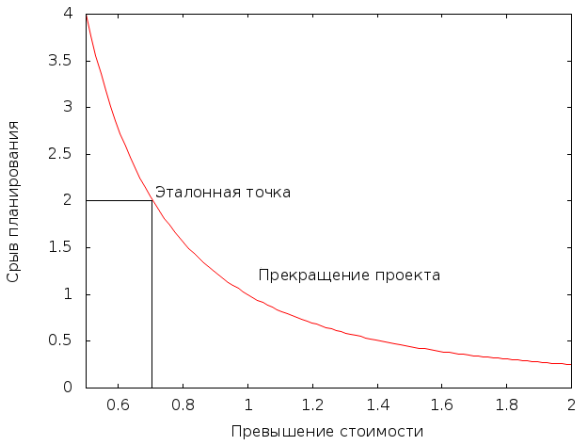


Рис. 9.1. Граница остановки проекта (срез для фиксированного упадка производительности)

Ниже кривой располагается рабочая область проекта, выше кривой — запретная область (при попадании в эту область проект должен быть прекращён).

Реально эталонный уровень редко представляется как кривая, чаще это сфероид, в котором есть области неопределённости (в них принять решение невозможно).

Теперь рассмотрим последовательность шагов планирования.

1. Исходными данными для планирования является набор четвёрок $[R_i, P_i, L_i, E_i]$, где R_i — i-й риск, P_i — вероятность появления i-го риска, L_i — потеря по i-му риску, E_i — влияние i-го риска.

2. Определяются эталонные уровни риска в проекте.

3. Разрабатываются зависимости между каждой четвёркой $[R_i, P_i, L_i, E_i]$ и каждым эталонным уровнем.

4. Формируется набор эталонных точек, образующих сфероид останова. В сфероиде останова предсказываются области неопределённости.

5. Для каждого элемента списка рисков разрабатывается план управления. Предложения плана составляются в виде ответов на вопросы «зачем?, что?, когда?, кто?, где?, как?, сколько?».

6. План управления каждым риском интегрируется в общий план программного проекта.

6. Разрешение и наблюдение рисков

Основанием для разрешения и наблюдения является план управления риском. Работы по разрешению и наблюдению производятся с начала и до конца процесса разработки.

Разрешение риска состоит в плановом применении действий по уменьшению риска.

Наблюдение риска гарантирует:

1. цикличность процесса слежения за риском;

2. вызов необходимых корректирующих воздействий.

Для управления рисками используется эффективная методика — **«Отслеживание 10 верхних рисков»**. Эта методика концентрирует внимание на факторах высоких рисков, экономит много времени, минимизирует «сюрпризы» разработки.

Рассмотрим шаги методики **«Отслеживания 10 верхних рисков».**

1. Выполняется выделение и ранжирование наиболее существенных элементов списка рисков в проекте.

2. Производится планирование регулярных просмотров (проверок) процесса разработки. В больших проектах (в группе больше 20 человек) просмотр должен проводиться ежемесячно, в остальных проектах — чаще.

3. Каждый просмотр начинается с обсуждения изменений в 10 верхних рисков (их количество может изменяться от 7 до 12). В обсуждении фиксируется текущий приоритет каждого из 10 верхних рисков, его приоритет в предыдущем просмотре, частота попадания риска в список верхних рисков. Если риск в списке

опустился, он по-прежнему нуждается в наблюдении, но не требует управляющего воздействия. Если риск поднялся в списке или только появился в нём, то он требует повышенного внимания. Кроме того, в обзоре обсуждается прогресс в разрешении риска (по сравнению с предыдущим просмотром).

4. Внимание участников просмотра концентрируется на любых проблемах в разрешении рисков.

Глава 10

Формирование и анализ требований

1. Виды требований

> ***Требованиями** (**requirements**) называют описание функциональных возможностей и ограничений, накладываемых на создаваемую систему.*

Обычно требования выражают, что система должна делать, а не как добиться реализации этих функций.

Пример .12 (требование к банковской системе). Система должна предоставить клиенту возможности выполнения следующих операций над его счётом: просмотр, снятие денег, добавление денег.

Пример .13 (не требование к банковской системе). А вот такая запись требованием не является: Информация банковского счёта должна храниться в виде трёх таблиц СУБД MySQL.

Здесь указывается как должна быть построена система, а не то, что она должна делать.

Впрочем, могут быть и исключения из этого правила. Например, у заказчика могут быть особые причины для такой реализации.

Различают две категории представления требований: требования заказчика (**первичные требования**) и требования разработчика (**детальные требования**).

Отличаются они друг от друга степенью проработки описаний. **Первичные требования** документируют желания и потребности заказчика и пишутся на языке, понятном заказчику.

Детальные требования документируют требования в специальной, структурированной форме, они детализированы по отношению к первичным требованиям.

> *Работу по созданию первичных требований называют **сбором**, или **формированием требований**.*

Проводится она на этапе «подготовка» жизненного цикла разработки.

> *Работу по созданию детальных требований называют **анализом требований**.*

Проводится она на этапе «моделирование» жизненного цикла разработки.

Некоторые проблемы порождены отсутствием чёткого понимания различия между этими категориями требований.

Пример .14 (первичные и детальные требования). Требования заказчика:

1. ПО должно обеспечить средства для ввода и сохранения разнообразных данных абонента пользователя.

Соответствующее ему требование разработчика:

1.1. Пользователь должен иметь возможность выбирать/определять тип вводимых данных.

1.2. Для каждого типа данных должно иметься соответствующее средство, обеспечивающее ввод и сохранение элемента данных этого типа.

1.3. Каждый тип данных должен представляться соответствующей пиктограммой на экране пользователя.

1.4. Пользователю должна предлагаться пиктограмма для каждого типа данных. Кроме того, должна предлагаться возможность самостоятельного выбора пиктограммы для каждого типа данных.

1.5. При выборе пользователем пиктограммы типа данных к элементу данных должно быть применено средство, ассоциированное с указанным типом.

Таким образом, требования заказчика являются первичным описанием функций, выполняемых системой, и ограничений, накладываемых на неё, на естественном языке. Дополнительно к списку требований могут прикладываться поясняющие диаграммы (UML диаграммы прецедентов). Требования заказчика помещаются в **системную спецификацию**.

Требования разработчика содержат детализированное описание функций и ограничений системы. Они оформляются в виде **спецификации анализа**. Эта спецификация служит основой для заключения контракта между заказчиком и разработчиком.

Часто обе спецификации интегрируют в единый документ.

Различают два вида требований:

Функциональные требования — описывают поведение системы и сервисы (функции), которые она должна выполнять. При этом исходят из всестороннего анализа проблемной (предметной) области. Рассматриваются разнообразные варианты поведения, определяемые различными данными и состояниями внешней среды.

Нефункциональные требования — относятся к характеристикам системы и её внешнего окружения. Дополнительно могут перечисляться ограничения, накладываемые на действия и функции системы, а также на условия разработки (ограничения по времени, ограничения на организацию проекта, стандарты и т. д.).

Пример .15 (функциональное требование). ПО должно вычислять стоимость товаров, находящихся в корзине пользователя.

Пример .16 (нефункциональное требование). Вычисление стоимости товаров должно производиться менее, чем за одну секунду.

Оно не задаёт конкретную работу, а задаёт количественную оценку работы, то есть характеризует работу.

Формы записи требований не регламентируются, однако требования должны быть:

- *ясными* (не допускать двоякого толкования, приводящего к искажению смысла пожеланий заказчика);
- *согласованными* (не содержать противоречивых и взаимоисключающих утверждений);
- *полными* (определять всю требуемую функциональность системы).

1.1. Нефункциональные требования

Нефункциональные требования не связаны непосредственно с функциями системы. Многие нефункциональные требования относятся к системе в целом, а не к отдельным её элементам. Это означает, что они могут быть более критичными, чем единичные функциональные требования. Ошибка в функциональном требовании может понизить функциональные возможности системы, а ошибка в нефункциональном требовании может привести к отказу всей системы.

Иан Соммервилл (Ian F. Sommerville, известный британский программист, преподаватель и автор популярного учебника по программной инженерии) предложил выделять три группы нефункциональных требований:

Требования к программной системе — описывают свойства и характеристики системы. Сюда относятся требования к скорости работы, производительности, объёму необходимой памяти, надёжности, переносимости системы на разные компьютерные платформы и удобству эксплуатации.

Организационные требования — отображают вопросы работы и организации взаимодействия заказчика и разработчика. Они включают стандарты разработки программной системы, требования к методам и средствам разработки, указывают сроки создания и набор документации.

Внешние требования — учитывают факторы внешней среды. Они определяют требования по взаимодействию данной системы с внешним окружением, юридические обязательства, а также этические требования, гарантирующие приемлемость системы для пользователей.

Примеры нефункциональных требований

Требования к производительности определяют временные ограничения, которые должны быть выполнены в системе. Это могут быть ограничения по времени вычислений, периодичности вычислений, использованию оперативной памяти, использованию внешних устройств и т. д.

Пример .17. Цикл регулирования скорости летательного аппарата должен укладываться в 64 мс.

Требования к производительности являются важной частью систем, работающих в реальном времени, в которых действия должны уложиться в определённые временные рамки. Примерами систем реального времени могут быть системы предотвращения столкновений, управления полётом.

Требования к надёжности задают уровень надёжности, определяя границы (или величину) отклонений в работе системы.

Пример .18. Система управления микроклиматом оранжереи должна давать не более двух ошибок в месяц.

Требования к доступности задают степень доступности системы для пользователей.

Пример .19. Система продажи авиабилетов должна быть доступна пользователям 24 часа в сутки. Она может быть недоступна (находиться в состоянии профилактики) в течение 10 минут за 30-дневный период.

Требования ограничений описывают границы характеристик или условий работы системы. В некоторых случаях могут ограничиваться и условия разработки.

Пример .20. Система управления крылатой ракеты (СУ КР) должна рассчитывать координаты цели с точностью до трёх метров.

Часто накладываются ограничения по инструментам и языкам. Они обусловлены сложившимися традициями организации, опытом своих программистов.

Пример .21. Система управления крылатой ракетой должна быть разработана на языке *Ada* 2005.

Управление конфигурацией разработки должно проводиться в среде IBM Rational ClearCase.

Ограничение, требующее следовать определённому стандарту, часто определяется политикой фирмы или внутренним стандартом заказчика.

Пример .22. Документация на СУ КР должна удовлетворять требованиям стандарта ГОСТ Р 51189-98.

Проекты часто ограничены платформами, на которых они будут использоваться.

Пример .23. Система управления крылатой ракеты должна работать на компьютерах Агат 1415 с расширением ёмкости оперативной памяти до 4 Гбайт.

Интерфейсные требования описывают формат, в котором система общается с внешней средой.

Пример .24. Для передачи сообщений в систему телеметрии используется строковый формат `out_tm<code>`, где `<code>` — двухбайтовый код из таблицы посылок `Tajna_321b`.

2. Формирование требований

Цель работы «формирование требований» — сформировать требования заказчика, она закреплена за ролью «разработчик требований». Требования заказчика представляются в такой форме, что они понятны любому пользователю, не владеющему специальными техническими знаниями. **Первичные требования** должны определять только внешнее поведение ПО, без детализации структурной организации системы. Они записываются на естественном языке с использованием простых таблиц, списков, а также наглядных диаграмм, рисунков.

К сожалению, естественный язык является *неоднозначным*, поэтому текстовое описание требований подвержено следующим недостаткам:

- *витиеватость стиля изложения*. Иногда нелегко выразить какую-то мысль на человеческом языке ясно и недвусмысленно, не сделав при этом текст многословным и трудночитаемым;
- *смешение и объединение требований*. В требованиях могут быть размыты границы между функциональными и нефункциональными требованиями. Несколько различных требований могут описываться как единое требование заказчика, и разработчик может сосредоточиться только на одном из них, потеряв другое.

2.1. Процесс формирования требований

Опишем шаги процесса формирования требований.

1. *Выявление представителей заказчика.*

   Важно выявить такой круг лиц, который позволит составить комплексное представление о портрете будущей системы (её функционале и полном перечне характеристик). Иногда это сделать довольно сложно. В любом случае надо разобраться в предметной области системы, выявить круг пользователей и заинтересованных лиц, в состав которых могут войти работники и руководство заказчика (и даже руководство других разработчиков), обслуживающий персонал и т. д.

2. *Проведение опроса представителей заказчика.*

   - Если запланирован опрос большого количества людей, то лучше провести онлайн опрос, используя, например, сервис SurveyMonkey или Google Формы.
   - Поскольку обычно имеется несколько заинтересованных лиц, первый вопрос — решить, в каком порядке их опрашивать (при проведении очного опроса). Очевидно, нужно расставить приоритеты в списке опрашиваемых (согласно вашим предположениям о важности получаемых сведений).
   - Далее планируется время и длительность опросов, на которых должны присутствовать как минимум два члена команды разработчиков.

- Во время интервью следует вести диалог (надо задавать вопросы и мотивировать собеседника, уточнять пожелания и потребности, предлагать варианты поведения и использования системы), делать подробные заметки, вести записи разговоров (на диктофоны, камеры и т. д.). В конце опроса следует запланировать следующую встречу.

  Пара разработчиков, участвующих в опросе и поддерживающих друг друга, обеспечивают возможность оперативной формулировки «трудных» требований со слов заказчика. Часто заказчик сам формулирует требования по ходу разговора, но иногда нуждается в помощи. В этих случаях разработчик и заказчик совместно «шлифуют» концепцию требования. Иными словами, заказчику бывают необходимы подсказки для формирования концепции.

  Эффективным способом получения и формулировки требований являются примеры. Эти примеры предлагаются разработчиками, как правило, в графической форме или в виде работающих прототипов (в т. ч., выполненных другими разработчиками).

3. *Документирование результатов опроса.*

После каждого опроса создаётся черновик набора требований. Он отсылается заказчикам для комментариев и коррекции. Как правило, затем проводится серия повторных опросов. Завершается серия опросов собранием, затем готовится документ, содержащий все требования. Этот документ утверждается заказчиком.

4. *Проверка требований.* Цель проверки спецификации требований состоит в оценке правильности определений, которые в ней содержатся. Проверка гарантирует, что все положения требований корректны, отражают желаемые характеристики и удовлетворяют потребностям заказчика. Может оказаться, что требования, которые в спецификации выглядели превосходно, при реализации чреваты проблемами. Разработчики испытывают серьёзный дискомфорт при реализации неясных или неполных требований. В случае отсутствия необходимой информации, они вынуждены ориентироваться на собственные предположения, которые не всегда верны. Доказано, что исправление ошибок в требованиях, работа над которыми уже завершена, требует очень много усилий. Исследования показали: исправлять ошибки требований в конце разработки системы в 100 раз дороже, чем в ходе формирования этих требований. Многократно подтверждён следующий постулат: «любые усилия, затраченные на выявление ошибок в спецификации требований, сэкономят реальные время и деньги».

Проверка требований выполняется заказчиком и разработчиком совместно, она удостоверяет:

4.1. предметная область проекта описана корректно;

4.2. разработчик и заказчик имеют одинаковые представления о целях системы;

4.3. анализ внешней среды и риска разработки подтверждает возможность создания системы;

4.4. спецификация требований верно описывает желаемую функциональность и характеристики системы, которые соответствуют потребностям заказчика и других заинтересованных лиц;

4.5. требования полные и качественные;

4.6. все требования согласованы друг с другом, не содержат противоречий;

4.7. требования обеспечивают реальную возможность разработки системы.

Проверка должна подтвердить, что в спецификации присутствуют только качественные требования как по содержанию (корректные, полные, согласованные, осуществимые и поддающиеся проверке), так и по форме записи (ясные, легко модифицируемые и поддающиеся отслеживанию).

Разумеется, проверить можно только задокументированные, а не воображаемые требования. Дело в том, что многие организации не справляются с формированием требований. Это не значит, что они не пользуются требованиями — просто требования существуют лишь в головах конкретных разработчиков. В итоге крах программного проекта становится почти неизбежным. Ещё одна проблема создаётся организациями, которые формируют лишь часть требований — требования для начальной итерации разработки. С этой порцией требований начинается проект, а вот изменения в спецификации требований для последующих итераций уже не поддерживаются. Конечно, сопровождение спецификации требований организовать нелегко, но всё же это необходимо сделать!

3. Анализ требований

Формирование требований является лишь начальной фазой работы с требованиями.

Анализ требований рассматривает требования заказчика как исходные данные, на выходе анализа — требования разработчика, которые называют **детальными требованиями**. Анализ требований служит мостом между подготовкой и проектированием, который служит для перехода из мира заказчика в мир разработчика. На этом этапе меняется язык записи требований, естественный язык уступает место языку формализованных моделей. Он непонятен заказчику, но близок разработчику. С помощью этих моделей можно добиться более точного отображения множества деталей, присущих программной системе. Причём в компактной форме, минимальным набором выразительных средств.

Конечно, эти модели тоже отвечают принципам построения требований: показывать «что» надо делать, а не «как» это делается. Однако для полного описания системы требуется такая детализация, которая должна включать информацию об организации системы на уровне архитектуры. Разработчикам программных систем нужен базис для проектирования и конструирования. Этот базис и образуется набором детальных требований, которые подробно, полно и согласованно описывают свойства и функциональность системы.

Каждое из этих требований происходит из первичных требований. При документировании каждое требование нумеруется и отслеживается по ходу разработки. Реализация каждого требования тестируется.

Рассмотрим типичные шаги анализа требований.

Организация первичных требований

Необходимость этой работы обусловлена большим количеством требований. По мере их разрастания неорганизованный список быстро превращается в неуправляемый. Стандарты рекомендуют несколько способов организации:

- По *режиму* — в случае, если система меняет поведение в зависимости от режима работы.

  Пример .25. Система управления может иметь различные наборы функций в зависимости от режима: обучение, нормальный режим или аварийный режим.

По *категориям пользователей* — если система предоставляет различные функции для разных категорий пользователей.

Пример .26. Система электронного документооборота предоставляет различные возможности для персонала компании, администратора системы и обычных интернет-посетителей.

- По *объектам*. Объекты — это программные сущности системы, которые могут иметь физические аналоги во внешней среде.

  Пример .27. В системе контроля за пациентом объекты включают пациентов, датчики, медсестёр, помещения, врачей, лекарства. Каждый объект несёт в себе набор данных и функций. Функции объектов также называют услугами, методами, операциями.

По *свойствам*. Свойство — сервис, предоставляемый внешней среде, определяется с помощью пар «входное воздействие — реакция». При этом «реакция» может быть рассредоточена по различным частям программной системы.

Пример .28. В телефонной системе свойства включают локальный вызов, переадресацию вызовов и циркулярный вызов.

- По *стимулам*. Некоторые системы легко организуются при описании их функций на языке стимулов.

  Пример .29. Функции автоматической системы посадки самолета могут быть организованы в разделы по энергетическим потерям, сдвигу ветра, изменению направления качения, избыточной вертикальной скорости и т. д.

По *откликам*. Некоторые системы организуются посредством описания всех функций, поддерживающих генерацию различных откликов.

Пример .30. Функции системы учёта персонала могут быть организованы в разделы, соответствующие всем функциям для составления чека по оплате, всем функциям для составления списка служащих и т. д.

- По *иерархии функций*, т. е. путём разделения ПО на множество высокоуровневых функций и последующего разбиения их на подфункции.

  Пример .31. Требования для ПО домашнего бюджета можно разбить на функции проверки, функции сбережений и функции инвестирования. Функции проверки могут затем быть разложены на функции чековой книжки, баланса счёта, функцию составления отчётов и т. д. Это классический способ упорядочения требований.

Иерархия функций может быть образована по общим вводам, по общим выводам, или по доступу к общим данным. Связи между функциями и данными можно показывать с помощью диаграмм BPMN с потоками данных и артефактами.

При рассмотрении системы могут оказаться применимыми несколько классификационных признаков организации. В таких случаях можно организовать конкретные требования в виде нескольких иерархий, построенных по 2–3 признакам. Например, возможна организация одновременно по категориям пользователей и свойствам.

Выбор организации требований автоматически влияет на выбор языка записи требований — формализованного метода и моделей анализа.

Пример .32 (формы записи требований).

- при организации по иерархии функций — диаграммы прецедентов, диаграммы BPMN с потоками данных и артефактами;
- при организации требований по режимам применяются конечные автоматы (диаграммы состояний);
- при организации по объектам — диаграммы классов;
- при организации по свойствам — диаграммы последовательности и состояний.

Преобразование первичных требований в детальные требования

Обычно одно требование заказчика преобразуется в несколько детальных требований, хотя возможно и отображение «один в один». Рекомендации по работе с детальным требованием:

1. *Первичная оценка.* Возможны варианты:

1.1. функциональное требование соответствует реализующему методу;

1.2. слишком большое — трудно управлять, следует разделить на части;

1.3. слишком маленькое — нет смысла рассматривать отдельно, надо присоединить к другому требованию.

2. *Обеспечение прослеживаемости требования* — анализ возможности прослеживания при проектировании и конструировании.

3. *Обеспечение тестируемости требования* — написание требований и рекомендаций к тестам. Продумываются варианты как положительного, так и отрицательного исхода тестов.

4. *Анализ однозначности толкования требования.*

5. *Назначение приоритета требования* — выбираются варианты: *существенное*, *желательное* или *необязательное.*

6. *Проверка полноты требования* — следует убедиться в наличии всех *обеспечивающих* требований.

7. *Проверка согласованности требования с другими требованиями* — анализируются и устраняются возможные противоречия.

8. Требование заносится в спецификацию анализа (спецификацию требований).

Аттестация детальных требований

Аттестация (**валидация**) должна подтвердить, что требования действительно определяют ту систему, которая нужна заказчику. Аттестация очень важна, так как ошибки требований могут привести к существенному изменению системы и большим затратам, если будут обнаружены на поздних стадиях разработки системы.

В состав аттестации входят:

1. *Проверка правильности требований.* Заказчик считает, что система необходима для выполнения заданных им функций. Однако обсуждение с заинтересованными лицами и последующий анализ могут выявить дополнительные функции, и их тоже надо учесть.

2. *Проверка на непротиворечивость.* Требования не должны определять противоречивые факты. В требованиях не должно быть противоречащих друг другу ограничений или различных определений одной и той же функции.

3. *Проверка на полноту.* Требования должны описывать все необходимые функции и ограничения системы.

4. *Проверка на выполнимость.* Здесь анализируется реализуемость требований в рамках *вменённых* и *бюджетных ограничений* проекта.

В ходе аттестации применяют следующие методы.

1. *Совместные проверки требований.* Требования анализируются рецензентами, избираемыми из числа заинтересованных лиц и внешних экспертов с целью найти неточности и ошибки. Обнаруженные противоречия, ошибки и упущения в требованиях документируются и передаются заказчику и разработчикам системы.

2. *Макетирование.* Макет создаётся разработчиком и обсуждается с заказчиком.

3. *Создание тестов* для требований. Этот процесс очень часто выявляет проблемы в описании требований. Проблемы при создании тестов означают, что требования трудно выполнить и поэтому их нужно пересмотреть.

4. *Автоматизированная проверка непротиворечивости.* Если требования представлены как формализованные модели, то для проверки непротиворечивости моделей можно применить CASE-инструменты.

Аттестация требований — дорогой и сложный процесс, требующий высокой квалификации, хорошего кругозора и развитого воображения.

Редко удаётся выявить все проблемы требований, поэтому возвращаться к аттестации приходится многократно.

4. Спецификация требований

> ***Спецификация требований*** *— это документ, являющийся официальным предписанием для разработчиков ПО, он содержит описание требований заказчика (первичных требований) и разработчика (детальных требований).*

Первичные требования документируются при формировании требований, а детальные требования — при выполнении анализа требований.

Многие организации разрабатывали стандарты документирования требований. Наиболее полным и авторитетным стандартом считают стандарт института инженеров по электротехнике и радиоэлектронике IEEE Std 830-1998, который должен помочь:

- заказчикам ИС точно описать, что они хотят получить;
- разработчикам ИС точно понять, что хочет заказчик.

Этот стандарт утверждает, что качественно составленная спецификация должна принести заказчикам, разработчикам и другим участникам следующие выгоды:

- *Создаёт основу для соглашения между заказчиком и разработчиком по поводу функционала ИС.* Полное описание функций ПО, приведённое в спецификации, поможет потенциальным пользователям определить, насколько отвечает продукт их потребностям и как можно его улучшить.
- *Уменьшает объём работ по разработке.* Подготовка спецификации вынуждает заказчика рассмотреть все требования до начала проектирования, что сокращает время на повторное проектирование, кодирование и тестирование. Тщательный анализ требований, указанных в спецификации, может вскрыть упущения, неправильное понимание и противоречия на ранних стадиях цикла разработки, когда эти проблемы проще исправить.

- *Обеспечивает основу для оценки бюджета и графика работ.* Спецификация продута является практической основой для оценки затрат на разработку и может использоваться для формирования контракта и бюджетных ограничений.

- *Создаёт базу для аттестации (валидации) и верификации.* Как часть контракта на разработку, спецификация обеспечивает основу для проведения проверок. При использовании качественной спецификации повышается эффективность аттестации и верификации.

- *Облегчает передачу конечного продукта пользователям.* Спецификация упрощает передачу (установку) ПО новым пользователям. Таким образом, для заказчиков упрощается передача ПО другим подразделениям их организации, а для разработчиков упрощается передача ПО новым заказчикам.

- *Является основой для развития продукта.* Поскольку в спецификации обсуждается продукт, а не проект, то спецификация служит основой для последующего расширения готового продукта. Спецификация может несколько измениться, но останется базисом для дальнейшей эволюции продукта.

Стандарт IEEE Std 830-1998 предлагает следующую структуру спецификации требований к ПО.

1. Введение.

1.1. Назначение (назначение спецификации, аудитория).

1.2. Область действия (название ПО, его задачи, применение).

1.3. Определения и сокращения (терминология).

1.4. Публикации (список литературы).

1.5. Краткий обзор (характеристика всех разделов).

2. Полное описание.

2.1. Перспектива изделия (оценка продукта, связи с другими продуктами).

2.2. Функции изделия (основные функции ПО).

2.3. Характеристики пользователя (общие характеристики пользователей продукта).

2.4. Ограничения (ограничения возможностей разработчика).

2.5. Допущения и зависимости (факторы, влияющие на требования).

2.6. Распределение требований (требования, которые откладываются до появления будущих версий продукта).

3. Конкретные требования (охватывают функциональные, нефункциональные и интерфейсные требования). Это наиболее значимая часть документа, описывает детальные требования, организованные выбранным способом.

4. Приложения (оценка себестоимости, форматы ввода/вывода, проблемы).

5. Алфавитный указатель.

В стандарте указано, что он может использоваться в качестве основы для создания собственных стандартов по требованиям.

5. Управление требованиями

После создания начальной версии требований приходит новое, более глубокое понимание предметной области ИС, прорисовываются новые детали, которые раньше были незаметны. Так начинается новый виток разработки ИС, который начинается с работы над требованиями. Постоянство требований — это, скорее, исключение из общего правила. Чтобы изменения требований не похоронили проект раньше времени, процессом изменения требований надо управлять.

В ходе управления требованиями нужно решить ряд вопросов:

- *Распознавание и учёт требований.* Каждое требование должно быть индивидуально учтено, поскольку оно может пересекаться с другими требованиями и использоваться в оценках трассировки.
- *Управление внесением изменений.* Должна предусматриваться последовательность защитных действий для оценки воздействия изменения и стоимости изменения.
- *Стратегия трассировки.* Существуют зависимости между требованиями, а также между требованиями и проектными решениями системы. Трассировка должна обнаруживать зависимые требования, запоминать эти зависимости и отслеживать влияние требований друг на друга и на проектные решения.

Управление требованиями нуждается в автоматизированной поддержке, которую обеспечивают программные утилиты.

Шаги процесса управления изменениями:

1. *Распознавание проблемы.*

1.1. Фиксируется проблема в требованиях или прямой запрос на внесение изменения.

1.2. Проверяется обоснованность проблемы или запроса. Если обоснованность подтверждена, переходят к следующему шагу, иначе процесс прекращается.

2. *Анализ изменения.* При определении возможности изменения исходят из информации трассировки и общих представлений о требованиях к системе. Стоимость изменения определяется двумя параметрами:

- стоимостью изменения спецификации;
- стоимостью изменения проектного решения системы и программного кода.

По окончании анализа принимается решение об изменении.

3. *Выполнение изменения* — вносится изменение в спецификацию требований и, если необходимо, в проектное решение и программный код. Спецификация требований должна быть организована так, чтобы внесение изменения носило локальный характер и не потребовало реорганизации всего документа. Гибкость документов достигается минимизацией внешних ссылок и обеспечением модульности разделов.

Всегда существует соблазн внести сначала изменение в программную систему, а лишь затем изменить спецификацию требований. *В этом случае «рассинхронизация» ПО и требований почти неизбежна!*

В гибких процессах разработки требования меняются в течение всего процесса разработки. В этих условиях, когда заказчик предлагает изменение требований, эти изменения не проходят через официальный процесс управления изменениями. Напротив, заказчик сам присваивает изменению приоритет, и если он высокий, то заказчик решает, что свойства системы, запланированные на следующую итерацию, должны быть пересмотрены.

Глава 11

Архитектурное проектирование

1. Этап проектирования

В ходе анализа ищется ответ на вопрос: «Что должна делать будущая система?». Именно на этой стадии закладывается фундамент успеха всего проекта.

В процессе синтеза формируется ответ на вопрос: «Каким образом система будет реализовывать предъявляемые к ней требования?». Выделяют три этапа синтеза программной системы: проектирование ИС, кодирование ИС, тестирование ИС (рис. 11.1).

Рассмотрим информационные потоки процесса синтеза. Этап проектирования использует детальные требования к ИС, представленные информационной, функциональной и поведенческой моделями анализа. Информационная модель описывает информацию, которую, по мнению заказчика, должна обрабатывать ИС. Функциональная модель определяет перечень функций обработки. Поведенческая модель фиксирует желаемую динамику системы (режимы её работы). На выходе этапа проектирования — модель данных, модель архитектуры и модели подсистем.

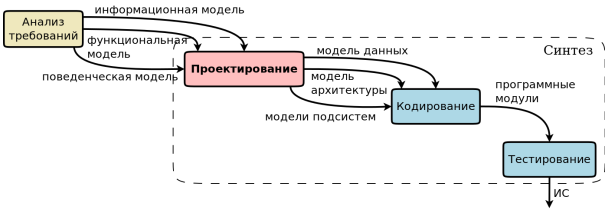


Рис. 11.1. Информационные потоки процесса синтеза

Модель данных — это результат преобразования информационной модели анализа в структуры данных, которые потребуются для реализации ИС.

Модель архитектуры выделяет основные структуры (подсистемы), фиксирует связи между ними и задаёт принципы взаимодействия между ними.

Модели подсистем описывают организацию подсистем, то есть определяют их содержание.

Далее выполняется кодирование программных модулей, проводится тестирование для объединения и проверки ИС.

На проектирование, кодирование и тестирование приходится более 75% стоимости разработки ИС. Принятые здесь решения оказывают решающее воздействие на успех реализации ИС, удобство эксплуатации и сопровождения ИС.

Следует отметить, что решения, принятые в ходе проектирования, делают его стержневым этапом процесса синтеза. Важность проектирования можно определить одним словом — качество. Проектирование — этап, на котором закладывается качество разработки ИС. Проектирование обеспечивает нас такими представлениями ИС, качество которых можно оценить.

Справедлива следующая аксиома разработки: *может быть плохая ИС при хорошем проектировании, но не может быть хорошей ИС при плохом проектировании.* Проектирование — единственный путь, обеспечивающий правильную трансляцию требований заказчика в конечный программный продукт.

2. Особенности архитектурного проектирования

Проектирование — итерационный процесс, при помощи которого требования к ИС транслируются в инженерные представления системы. Вначале эти представления дают только концептуальную информацию (на высоком уровне абстракции), последующие уточнения приводят к формам, которые близки к программному коду.

Обычно в проектировании выделяют две ступени: архитектурное проектирование и детальное проектирование (см. рис. 11.2). Архитектурное проектирование формирует абстракции высокого уровня, детальное проектирование уточняет эти абстракции, добавляет подробности алгоритмического уровня. Кроме того, во многих случаях выделяют интерфейсное проектирование, цель которого сформировать графический интерфейс пользователя (GUI).

Рис. 11.2. Информационные потоки процесса проектирования

Архитектурное проектирование обеспечивает понимание правильной организации системы и создаёт структуру под эту правильную организацию. Оно напрямик связывает весь этап проектирования с детальными требованиями, поскольку архитектура выделяет основные структурные компоненты системы и формирует отношения между ними.

Между созданием детальных требований и архитектурным проектированием имеется существенное перекрытие. В идеале в требованиях не должно быть информации о структуре системы. На самом же деле это справедливо только для малых систем. Архитектурное разделение системы на части просто необходимо для структуризации и организации спецификации требований. Поэтому перед созданием детальных требований формируется абстрактная архитектура системы, с подсистемами которой ассоциируются целые группы функций и характеристик. Эта же архитектура

используется для обсуждения требований и характеристик системы с заинтересованными лицами.

Многие специалисты отмечают решающее влияние архитектуры на качество программной системы:

- Архитектура является планом для переговоров по требованиям к системе и служит средством упорядочения обсуждений с клиентами, разработчиками и менеджерами.

- Архитектура — основной инструмент для управления сложностью и качеством системы. Подсистемы, составные части архитектуры, отвечают за реализацию функциональных требований и сильно влияют на нефункциональные требования. Именно архитектура определяет такие характеристики системы в целом, как производительность, защищённость, безопасность, устойчивость к отказами, сопровождаемость.

- Архитектура — высокоуровневый инструмент повторного использования программного обеспечения. Она обеспечивает целые семейства систем со схожей функциональностью, упрощая разработку конкретного экземпляра, с конкретными функциями и характеристиками.

Различным требованиям к интегральным характеристикам системы должны соответствовать разные варианты архитектуры.

Если главным требованием является *производительность системы*, следует проектировать архитектуру, которая локализует критические операции в пределах небольшого количества подсистем с минимальным числом взаимодействий между ними. Эти подсистемы следует развернуть на одном и том же компьютере, а не распределять по сети. Для сокращения внешних коммуникаций подсистемы должны быть крупными, с большой степенью автономности.

Для обеспечения *максимальной защищённости* архитектура должна быть многоуровневой, причём самые ценные подсистемы следует размещать на внутренних уровнях, с высокой степенью защиты.

При ориентации на *надёжную работу* нужно сосредоточить все вычислительные узлы в одной подсистеме (или малом числе подсистем). Это уменьшит стоимость проектирования и позволит применить эффективный механизм проверки надёжности.

Для создания ИС, *устойчивой к отказам* (обеспечения бесперебойности работы), в архитектуру вводятся избыточные (резервные) подсистемы. Их наличие позволяет выполнять замену отказавших элементов без остановки системы.

Удобство сопровождения повышается при проектировании архитектуры из малых подсистем, которые легче проверять и обновлять. Для облегчения диагностики ошибок желателен отказ от глобальных структур данных. При этом возникает конфликт с архитектурой, нацеленной на максимальную производительность. Там нужны большие подсистемы, здесь малые. Приходится идти на какой то компромисс.

Поиск компромиссного решения — типичная задача архитектурного проектирования, поскольку, как правило, требуется максимизация многих параметров систе-

мы.

Архитектура системы обычно моделируется с помощью UML диаграмм компонентов и диаграмм развёртывания (размещения).

Во время архитектурного проектирования системные архитекторы должны принимать такие решения, которые глубоко затрагивают систему и процесс её разработки. Основываясь на своих знаниях и опыте, они решают широкий спектр задач. Все эти задачи группируются вокруг трёх типов базисной деятельности.

Базисная деятельность архитектурного проектирования включает:

1. *Структурирование системы* — система декомпозируется на несколько подсистем (независимых программных компонентов), определяются взаимодействия подсистем.

2. *Моделирование управления* — формируется стратегия управления частями системы.

3. *Информационное моделирование* — на основе концептуальной модели данных проектируется логическая модель данных.

Рассмотрим вопросы структурирования и моделирования управления более подробно.

3. Типовые архитектуры

В ходе архитектурного проектирования создаётся структурная организация системы и архитектура управления, которая будет отвечать всем функциональным и нефункциональным требованиям.

Каждая программная система уникальна, но системы в одной и той же прикладной области часто имеют сходные архитектуры, отражающие фундаментальные положения этой области. Семейство продуктов — это приложения, которые строятся вокруг основной архитектуры с вариантами, удовлетворяющими специфическим требованиям клиента. Проектируя системную архитектуру, нужно выяснить общие черты создаваемой системы, а также более широкой категории приложений и решить, что можно позаимствовать из этой прикладной архитектуры.

Во встроенных системах имеется только один процессор и не нужно проектировать распределённую архитектуру. Однако современные ПК и, тем более, большие системы являются многоядерными распределёнными системами, в которых функционал ИС распределён по многим вычислительным узлам. Выбор распределённой архитектуры — серьёзное решение, которое влияет на производительность и надёжность системы.

Архитектура конкретной ИС может быть основана на определённой **типовой архитектуре** (иногда называемой **архитектурным шаблоном**).

> ***Архитектурный шаблон*** *— типовое решение, охватывающее собой архитектуру всей программной системы, основанное на лучших практиках предыдущих разработчиков.*

Архитектурный шаблон можно рассматривать как обобщённое описание хорошей практики, опробованной и проверенной в различных системах и средах. Он описывает системную организацию, которая была успешна в предыдущих системах.

Идея **архитектурных шаблонов**, как способа повторного использования знаний об архитектурах ИС, в настоящее время находит широко применение.

Поскольку **архитектурный шаблон** является образцом типового решения, который могут применять многие архитекторы, он должен содержать развёрнутое описание. Описание **архитектурного шаблона** должно включать:

Имя шаблона (должно характеризовать его суть).

Описание (краткое и понятное описание структуры шаблона и функциональных возможностей, структура поясняется диаграммой).

Пример (приводится пример типичного применения шаблона, поясняемый диаграммой).

Когда используется (описываются условия применения, даётся обобщённая характеристика предметных областей).

Преимущества (перечисляются преимущества применения).

Недостатки (указываются слабые стороны данного решения).

3.1. Модель-Представление-Контроллер (MVC)

Архитектурный шаблон MVC (от Model View Controller) разделяет работу приложения на три отдельные функциональные блока:

- **Модель** данных (**model**) — предоставляет знания: данные и методы работы с этими данными, реагирует на запросы, изменяя своё состояние. Не содержит информации, как эти знания можно визуализировать.
- **Представление** (**view**, пользовательский интерфейс) — отвечает за отображение информации (визуализацию). Часто в качестве представления выступает форма (окно) с графическими элементами.
- **Управляющая логика** (**controller**) — обеспечивает связь между пользователем и системой: контролирует ввод данных пользователем и использует модель и представление для реализации необходимой реакции.

Таким образом, изменения, вносимые в один из компонентов, оказывают минимально возможное воздействие на другие компоненты.

Важно отметить, что как **представление**, так и **контроллер** зависят от **модели**. Однако **модель** не зависит ни от **представления**, ни от **контроллера**, что делает возможным проектирование **модели** как независимого компонента и, например, создавать несколько **представлений** для одной **модели**.

Впервые этот шаблон был применён во фреймворке, разрабатываемом для языка *Smalltalk* в конце 1970-х годов. С этого момента он играет основополагающую роль

в большинстве фреймфорков с пользовательским интерфейсом. Он в корне изменил взгляд на проектирование приложений. Особенно часто он используется в проектировании веб-систем.

Большинство фреймворков для веб-программирования сейчас в основе своей содержат именно **MVC**.

Пример .33 (фреймворки с архитектурой MVC). Фреймворки, наиболее удачно реализующие этот шаблон:

- для языка *PHP* — *Zend Framework*, *Yii*, *cakePHP*;
- для языка *Python* — *Django*;
- для языка *Ruby* — *Ruby on Rails* (RoR).

Описание архитектурного шаблона MVC
Имя шаблона: MVC (Модель-Представление-Контроллер).

Описание: **модель** оповещает **представление** о том, что в ней произошли изменения, а **представления**, которые заинтересованы в оповещении, подписываются на эти сообщения. Это позволяет сохранить независимость **модели** как от **контроллера**, так и от **представления** (см. рис. 11.3).

Пример: системы, основанные на WWW — динамические веб-сайты (см. рис. 11.4, 11.5).

Когда используется: основная цель применения этой концепции состоит в разделении данных (**модели**) и бизнес-логики (**контроллера**) от её визуализации (**представления**, **вида**). За счёт такого разделения повышается возможность повторного использования. Наиболее полезно применение данной концепции в тех случаях, когда пользователь должен видеть те же самые данные одновременно в различных контекстах и/или с различных точек зрения.

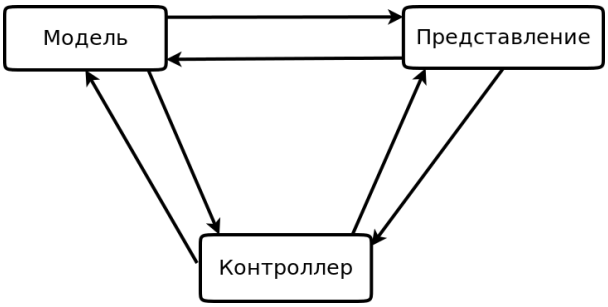


Рис. 11.3. Архитектура MVC (Модель-Представление-Контроллер)

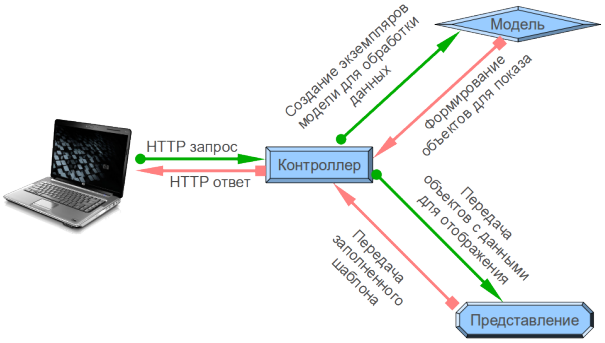


Рис. 11.4. Web-based система, построенная на архитектуре MVC

Рис. 11.5. Принцип работы веб-системы, построенной на архитектуре MVC

Преимущества:

1. К одной **модели** можно присоединить несколько **видов**, при этом не затрагивая реализацию **модели**. Например, некоторые данные могут быть одновременно представлены в виде электронной таблицы, гистограммы и круговой диаграммы.
2. Не затрагивая реализацию **видов**, можно изменить реакции на действия пользователя (нажатие мышью на кнопке, ввод данных), для этого достаточно использовать другой **контроллер**.
3. Ряд разработчиков специализируется только в одной из областей: либо разрабатывают графический интерфейс, либо разрабатывают бизнес-логику. Поэтому возможно добиться того, что программисты, занимающиеся разработкой бизнес-логики (**контролллера**), вообще не будут осведомлены о том, какое **представление** будет использоваться.

Недостатки: избыточность программного кода для простой модели данных и простой логики взаимодействия.

3.2. Архитектура файл-сервер

> ***Файл-сервер*** *(file-server) — архитектура, в которой приложения, схожие по своей структуре с локальными приложениями, используют сетевой ресурс для хранения данных в виде отдельных файлов.*

Функции сервера в таком случае обычно ограничиваются хранением данных (возможно также хранение исполняемых файлов), а обработка данных происходит исключительно на стороне клиента (см. рис. 11.6). Количество клиентов ограничено десятками, ввиду невозможности одновременного доступа на запись к одному файлу. Однако, клиентов может быть в разы больше, если они обращаются к файлам исключительно в режиме чтения.

Рис. 11.6. Файл-серверная архитектура

Достоинства:

- низкая стоимость разработки;
- высокая скорость разработки;
- невысокая стоимость обновления и изменения ПО.

Недостатки:

- рост числа клиентов резко увеличивает объём трафика и нагрузку на сети передачи данных;
- высокие затраты на модернизацию и сопровождение сервисов бизнес-логики на каждой клиентской рабочей станции;
- низкая надёжность системы.

Примеры: Устаревшие ИС, СПС Консультант+, *NFS*, *WebDAV*.

3.3. Архитектура клиент-сервер

> ***Клиент-сервер*** *(client-server) — вычислительная или сетевая архитектура, в которой задания или сетевая нагрузка распределены между поставщиками услуг, называемыми серверами, и заказчиками услуг, называемыми клиентами.*

Физически клиент и сервер — это программное обеспечение. Обычно они взаимодействуют через компьютерную сеть посредством сетевых протоколов и находятся на разных вычислительных машинах, но могут выполняться также и на одной машине. Программы-сервера, ожидают от клиентских программ запросы (см. рис. 11.7) и предоставляют им свои ресурсы в виде данных (например, загрузка файлов посредством HTTP, FTP, BitTorrent, потоковое мультимедиа или работа с базами данных) или сервисных функций (например, работа с электронной почтой, общение посредством систем мгновенного обмена сообщениями, просмотр веб-страниц в WWW).

Примеры: Классические ИС (бухгалтерские ИС, ERP, CASE, СПС Гарант).

Преимущества:

- *Централизованный доступ к данным*, как следствие — отсутствие дублирования кода программы-сервера программами-клиентами.
- *Простота обслуживания* — так как все вычисления выполняются на сервере, то требования к компьютерам, на которых установлен клиент, снижаются.
- *Бóльшая безопасность*: все данные хранятся на сервере, который, как правило, защищён гораздо лучше большинства клиентов. На сервере проще организовать контроль полномочий, чтобы разрешать доступ к данным только клиентам с соответствующими правами доступа.

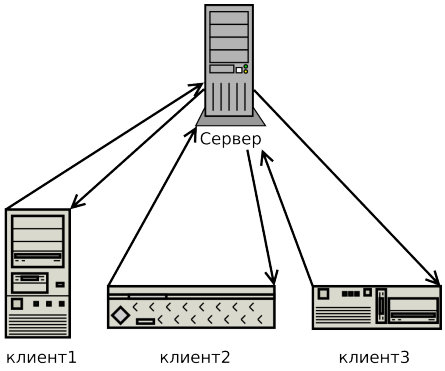


Рис. 11.7. Клиент-серверная (двухуровневая) архитектура

Недостатки:

- Неработоспособность сервера может сделать неработоспособной всю вычислительную сеть. Неработоспособным сервером следует считать сервер, производительности которого не хватает на обслуживание всех клиентов, а также сервер, находящийся на ремонте, профилактике и т. п.
- Поддержка работы данной системы требует отдельного специалиста — системного администратора.
- Высокая стоимость оборудования.

3.4. Многоуровневая архитектура клиент-сервер

Многоуровневая архитектура клиент-сервер *— разновидность архитектуры клиент-сервер, в которой функция обработки данных вынесена на один или несколько отдельных серверов.*

Это позволяет разделить функции хранения, обработки и представления данных для более эффективного использования возможностей серверов и клиентов.

3.5. Трёхуровневая архитектура

Трёхуровневая архитектура *(three-tier) — модель программного комплекса, предполагающая наличие в нём трёх компонентов: клиента, сервера приложений (к которому подключено клиентское приложение) и сервера баз данных (с которым работает сервер приложений).*

Клиент (слой клиента) — это интерфейсный компонент комплекса (обычно, имеющий графический интерфейс), предоставляемый конечному пользователю. Этот уровень не должен:

- иметь прямых связей с базой данных (по требованиям безопасности и масштабируемости);
- быть нагруженным основной бизнес-логикой (по требованиям масштабируемости);
- хранить состояние приложения (по требованиям надёжности).

На уровень клиента обычно выносится только простейшая бизнес-логика: интерфейс аутентификации, алгоритмы шифрования, проверка вводимых значений на допустимость и соответствие формату, несложные операции с данными (сортировка, группировка, подсчёт значений), загруженными на компьютер пользователя.

Сервер приложений (средний слой, связующий слой) располагается на втором уровне, на нём сосредоточена бóльшая часть бизнес-логики. Вне его остаются только фрагменты, экспортируемые на клиента (терминалы), а также элементы логики, размещённые в БД (хранимые процедуры и триггеры). Реализация данного компонента обеспечивается связующим программным обеспечением. Серверы приложений проектируются так, чтобы добавление дополнительных хостов обеспечивало горизонтальное масштабирование производительности ИС и не требовало внесения изменений в программный код приложения.

Сервер баз данных (слой данных) обеспечивает хранение данных и выносится на отдельный уровень, реализуется, как правило, средствами СУБД, подключение к этому компоненту обеспечивается только с уровня сервера приложений.

В простейших конфигурациях все компоненты или часть из них могут быть совмещены на одном вычислительном узле. В продуктивных конфигурациях используется выделенный вычислительный узел для сервера БД или кластер серверов баз данных, для серверов приложений — выделенная группа хостов, к которым непосредственно подключаются клиенты (см. рис. 11.8).

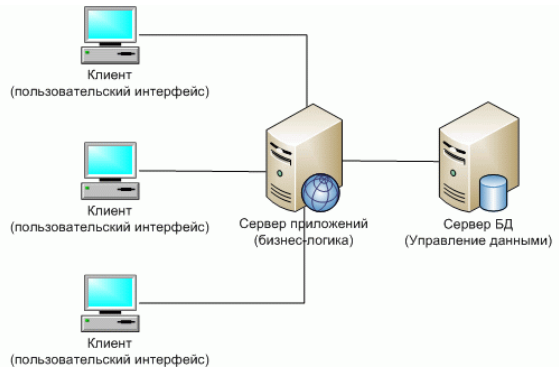


Рис. 11.8. Трёхуровневая архитектура

По сравнению с клиент-серверной или файл-серверной архитектурой трёхуровневая архитектура обеспечивает:

- бóльшую масштабируемость (за счёт горизонтальной масштабируемости сервера приложений и мультиплексирования соединений);
- бóльшую конфигурируемость (за счёт изолированности уровней друг от друга);

- более широкие возможности по обеспечению безопасности и отказоустойчивости;
- более низкие (в сравнении с клиент-серверными приложениями) требования к скорости и стабильности каналов связи между клиентом и серверной частью.

Реализация приложений, доступных из веб-браузера или из тонкого клиента, как правило, подразумевает развёртывание программного комплекса в трёхуровневой архитектуре. При этом, обычно разработка приложений для трёхуровневых программных комплексов сложнее, чем для клиент-серверных приложений, также наличие дополнительного связующего программного обеспечения может налагать дополнительные издержки в администрировании таких комплексов.

3.6. Облачная архитектура

> ***Облачные вычисления (cloud computing)*** *— технология распределённой обработки данных, в которой компьютерные ресурсы и мощности предоставляются пользователю как интернет-сервис.*

Облачный сервис представляет собой особую клиент-серверную технологию — использование клиентом ресурсов (процессорное время, оперативная память, дисковое пространство, сетевые каналы, специализированные контроллеры, программное обеспечение и т. д.) группы серверов в сети, взаимодействующих таким образом, что:

- для клиента вся группа выглядит как единый виртуальный сервер;
- клиент может прозрачно и с высокой гибкостью менять объёмы потребляемых ресурсов в случае изменения своих потребностей (увеличивать/уменьшать мощность сервера с соответствующим изменением оплаты за него).

При этом наличие нескольких источников используемых ресурсов, с одной стороны, позволяет повышать доступность системы клиент-сервер за счёт возможности масштабирования при повышении нагрузки (увеличение количества используемых источников данного ресурса пропорционально увеличению потребности в нём и/или перенос работающего виртуального сервера на более мощный источник, «живая миграция»), а с другой — снижает риск неработоспособности виртуального сервера в случае выхода из строя какого-либо из серверов, входящих в группу, обслуживающую данного клиента, так как вместо вышедшего из строя сервера возможно автоматическое переподключение виртуального сервера к ресурсам другого (резервного) сервера.

Термин «Облако» используется как метафора, основанная на изображении Интернета на диаграмме компьютерной сети, или как образ сложной инфраструктуры, за которой скрываются все технические детали. Широко распространённое формальное определение облачных вычислений было предложено Национальным институтом стандартов и технологий США — ANSI:

Облачные вычисления *представляют собой модель для обеспечения по требованию удобного сетевого доступа к общему пулу настраиваемых вычислительных ресурсов (например, сетей, серверов, систем хранения данных, приложений и услуг), которые можно быстро выделить и предоставить с минимальными управленческими усилиями или минимальным вмешательством со стороны поставщика услуг.*

Облачная обработка данных как концепция включает в себя понятия:

- «Инфраструктура как услуга» — **IaaS**;
- «Платформа как услуга» — **PaaS**;
- «Программное обеспечение как услуга» — **SaaS**;
- «Данные как услуга»;
- «Рабочее место как услуга»;
- «Всё как услуга»;
- другие технологические тенденции, общим в которых является уверенность, что сеть Интернет в состоянии удовлетворить потребности пользователей в обработке данных.

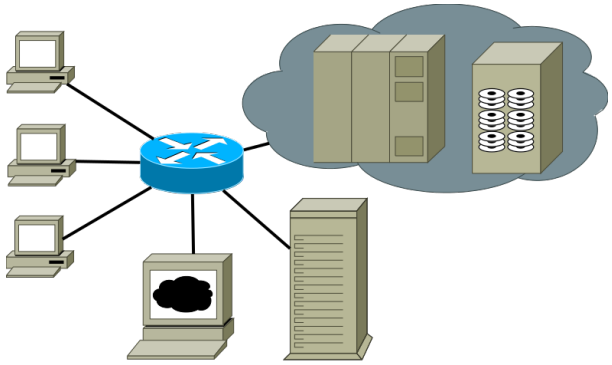


Рис. 11.9. ИС, построенная на облачной архитектуре

3.7. Разнообразие архитектур

Большая коллекция архитектурных шаблонов собрана Мартином Фаулером: design-pattern.ru.

3.8. Паттерны управления

Паттерны управления можно разделить на *паттерны централизованного управления* (то есть паттерны, в которых одна из подсистем полностью отвечает за управление, запускает и завершает работу остальных подсистем) и паттерны управления, подразумевающие децентрализованное реагирование на события (согласно этим паттернам на внешние события отвечает соответствующая подсистема).

Также следует упомянуть, что поскольку проектирование взаимодействия той или иной подсистемы с реляционной базой данных является неотъемлемой частью разработки информационных систем, среди паттернов управления выделена большая группа паттернов, описывающих организацию связи с базой данных.

Паттерны централизованного управления

Вызов — возврат

Описание: Вызов программных процедур осуществляется «сверху — вниз», то есть управление начинается на вершине иерархии процедур и через вызовы передаётся на нижние уровни иерархии.

Рекомендации: Паттерн применим только в последовательных системах, то есть в таких системах, в которых процессы должны происходить последовательно.

Преимущества: Простой анализ потоков управления. Последовательные системы легче проектировать и тестировать.

Недостатки: Сложно обрабатывать исключительные ситуации.

Пример .34 (Сценарий транзакции). Сценарий транзакции можно считать частным случаем паттерна **вызов — возврат**. Сценарий транзакции может быть рассмотрен как способ организации бизнес-логики («модель предметной области»).

Сценарий транзакции — процедура, которая получает на вход информацию от слоя представления, обрабатывает её, производя необходимые проверки и вычисления, сохраняет в базе данных и активизирует операции других систем.

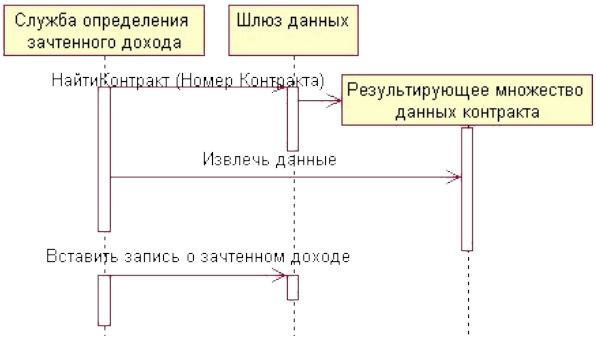


Рис. 11.10. Сценарий транзакции

Сценарий транзакции не годится для сложной бизнес-логики.

Диспетчер

Описание: Один системный компонент назначается *диспетчером* и управляет запуском и завершением других процессов системы и координирует эти процессы. Процессы могут протекать параллельно.

Рекомендации: Применяется в системах, в которых необходимо организовать параллельные процессы, но может использоваться также и для последовательных систем, в которых управляющая программа вызывает отдельные подсистемы в зависимости от значений некоторых переменных состояния.

Пример: Можно использовать в системах реального времени, где нет слишком строгих временных ограничений (в так называемых «мягких» системах реального времени).

Паттерны управления, основанные на событиях

Передача сообщений

Описание: В рамках данного паттерна событие представляет собой передачу сообщения всем подсистемам. Любая подсистема, которая обрабатывает данное событие, отвечает на него.

Рекомендации: Данный подход эффективен при интеграции подсистем, распределённых на разных компьютерах, которые объединены в сеть.

Управляемый прерываниями

Описание: При использовании данного паттерна внешние прерывания регистрируются *обработчиком прерываний*, а обрабатываются другим системным компонентом.

Рекомендации: Используются в системах реального времени со строгими временными требованиями. Можно комбинировать с паттерном **Диспетчер**: центральный диспетчер управляет нормальной работой системы, а в критических ситуациях используется управление, основанное на прерываниях.

Преимущества: Достаточно быстрая реакция системы на происходящие события.

Недостатки: При использовании данного подхода система сложна в программировании. При тестировании системы затруднительно имитировать все прерывания. Число прерываний ограничено используемой аппаратурой (после достижения предела, связанного с аппаратными ограничениями, никакие другие прерывания не обрабатываются).

4. Проектирование модулей

После проектирования системной структуры и определения принципов управления структурой следует выполнить разделение подсистем на модули.

Фактически эту работу можно считать введением в детальное проектирование, которое конкретизирует архитектурные решения.

Задача декомпозиции это задача определения внутреннего содержания каждой подсистемы. Результатом её решения является формирование структуры подсистемы в виде набора модулей и отношений их взаимодействия.

Известны два подхода и два типа моделей для декомпозиции подсистем на модули:

- модель потока данных;
- объектно-ориентированная модель.

В основе модели потока данных лежит разбиение по функциям. При выборе этой модели получается набор функциональных модулей с определёнными связями между ними. Например, результат может быть похож на реализацию диаграммы потоков данных.

Объектно-ориентированная модель основана на объектах (слабо сцепленных сущностях, имеющих собственные наборы данных, состояния, наборы операций) и их описаниях — классах.

Какой тип декомпозиции следует выбирать? Ответ па этот вопрос зависит от большого количества факторов, в том числе и от сложности разбиваемой подсистемы.

Однако, при любом подходе можно использовать готовые решения в виде **шаблонов** (**паттернов**) *проектирования*.

На этапе проектирования модулей применение **шаблонов проектирования** приобретает ещё большую актуальность (особенно в объектно-ориентированном подходе). Именно в связи с проектированием модулей такое понятие вошло в обиход в проектирование программных систем.

Глава 12

Проектирование интерфейса

1. Человекоориентированный интерфейс

1.1. Базовые понятия

> ***Интерфейс*** *— это способ, которым человек выполняет какую-либо задачу с помощью какого-либо продукта, а именно совершаемые им действия и то, что он получает в ответ.*

В основе разработки хороших интерфейсов лежат некоторые основные принципы, которые на сегодня не являются общеизвестными. И вопрос о необходимости изучения этих принципов не возникает, поскольку кажется, что уже определено, как должны выглядеть и работать интерфейсы: ведь они непрерывно совершенствовались примерно с 1990 г., основные разработчики программного обеспечения опубликовали руководства по созданию интерфейсов, чтобы обеспечить совместимость между ними, а существующие средства разработки позволяют быстро создавать любые интерфейсы, которые выглядят по современному.

Все эти интерфейсы неспособны выполнять многие важные для нас задачи. Например, чтобы записать какую-то мысль, Вы хотели бы просто подойти к компьютеру или другому устройству для обработки информации и начать набирать её — без всякой загрузки, без необходимости открывать текстовый процессор, создавать файл, вообще без использования операционной системы.

Пример .35 (ОС). Определение операционной системы Дж. Раскина:

> «То, с чем приходится возиться перед тем, как начать возиться с программой».

Чтобы добавить к репертуару системы несколько средств для выполнения простых операций, Вы не обязаны изучать целую прикладную программу. К сожалению в разработке интерфейсов изначально было взято неверное направление, и это привело к тому, что уровень их сложности стал неоправданно высоким с точки зрения как технологической, так и логической необходимости.

Миллионы из нас имеют противоречивые отношения с информационными технологиями. Мы не можем жить без них, и в то же время нам трудно жить с ними.

Тем не менее, проблема создания удобных и простых технологий имеет свои решения, хотя мы и не можем ими воспользоваться — они станут доступными, только если мы оставим груз прошлого. Привычный вариант интерфейса в виде рабочего стола, ориентированный на работу с прикладными программами, является частью этой проблемы.

Несмотря на рост количества специалистов по разработке интерфейсов, мало кто из потребителей заявляет, что новые продукты, например электронные четырёхкнопочные наручные часы, стали проще в использовании, чем несколько десятилетий назад. Сложность неоправданно возникает в отношении даже тех задач, которые раньше удавалось выполнять без усилий. *Простые задачи должны оставаться простыми независимо от уровня сложности всей системы!*

Сложные задачи могут требовать сложных интерфейсов, но это не оправдывает усложнения простых задач. Сравните, например, насколько труднее установить время на электронных наручных часах с четырьмя кнопками, чем выполнить то же самое действие на механической модели часов.

> ***Интерфейс*** *является* ориентированным на человека*, если он отвечает нуждам человека и учитывает его слабости.*

Чтобы создать такой интерфейс, необходимо иметь представление о том, как действуют люди и машины. Кроме того, следует развить в себе способность чувствовать те трудности, с которыми сталкиваются люди. И это не всегда просто. Мы настолько привыкли к тому, как работают программы, что соглашаемся принять их методы работы как данность, — даже в тех случаях, когда их интерфейсы неоправданно сложны, запутаны, не экономны и побуждают людей к ошибкам.

Многие из нас испытывают раздражение, например, от того, что для запуска (иначе говоря, загрузки) компьютера требуется какое-то время.

Пример .36 (загрузка ОС). Уже вышедший из употребления Apple Newton, Palm Pilot и другие карманные компьютеры могут запускаться мгновенно.

Появление на ПК «спящего режима» — состояния, в котором компьютер потребляет меньше энергии, чем в обычном режиме, и из которого он может быть быстро переведён в рабочее состояние, — это шаг в правильном направлении.

Инженерам удавалось с успехом решать и более сложные проблемы. Например, в ранних моделях телевизоров необходимо было ждать около минуты, пока разогревалась катодная трубка кинескопа. В некоторых моделях инженеры добавили специальную схему, которая поддерживала катодную трубку в тёплом состоянии, что позволило сократить время достижения рабочей температуры. (Поддержание катодной трубки в разогретом состоянии потребовало бы большого расхода электричества и уменьшило бы срок её службы.) В другом варианте был разработан кинескоп с катодной трубкой, которая разогревалась в течение нескольких секунд. И в том и в другом случае интересы пользователя были удовлетворены. В начале двадцатого столетия был создан автомобиль на паровой тяге, называвшийся Стенли Стимер (Stanley Steamer). Несмотря на все свои очевидные достоинства, этот механизм не имел успеха из-за одного недостатка: чтобы тронуться с места, от момента

зажигания до достижения необходимого давления в котле требовалось подождать 20 минут.

Принцип разработки, согласно которому программные продукты не должны вынуждать пользователя ждать без необходимости, можно считать очевидным и ориентированным на человека. Таким же является и стремление не подгонять пользователя. В общем виде этот принцип можно было бы сформулировать следующим образом: Ритм взаимодействия должен устанавливаться самим пользователем.

Пример .37. Настройка скорости двойного нажатия клавиш для клавиатуры и мышки. Настройка чувствительности перемещения мышки.

Не требуется обладать большими техническими знаниями, чтобы понять, что большая пропускная способность коммуникационных линий может ускорить передачу веб-страниц. Однако другие взаимосвязи иногда бывают не столь очевидны. Поэтому для разработчиков интерфейсов «человек — машина» важно знать внутренние механизмы технологии. В противном случае у них не будет возможности оценивать достоверность утверждений, высказанных, например, программистами или специалистами по аппаратной разработке относительно осуществимости тех или иных элементов интерфейса.

При всей сложности компьютеров и других продуктов современной технологии «машинная» часть интерфейса «человек — машина» легче поддаётся пониманию, чем человеческая — намного более сложная и изменчивая. Тем не менее, многие (возможно, даже очень многие) факторы человеческой производительности не зависят от возраста, пола, культурного происхождения или уровня компетентности пользователя. Эти свойства человеческой производительности и способности к обучению имеют непосредственное отношение к основам разработки любого интерфейса.

Используйте машину или инструмент в соответствии с их возможностями и ограничениями, и они сослужат Вам хорошую службу. Разрабатывайте интерфейс «человек — машина» в соответствии с возможностями и слабостями человека, и Вы поможете пользователю не только справиться с работой, но и сделать его более счастливым, более продуктивным человеком.

Руководства по разработке продуктов, взаимодействующих с нами физически, обычно содержат конкретную информацию, основанную на свойствах и возможностях человеческого скелета и органов чувств. Совокупность сведений в этой области составляет науку **эргономику**.

> ***Эргономика*** *— это научная дисциплина, комплексно изучающая физиологические стороны человека в конкретных условиях его деятельности в современном производстве.*

Основной объект исследования эргономики: система «человек — машина». На основе этих знаний можно проектировать стулья, столы, клавиатуры или дисплеи, которые с высокой степенью вероятности будут удобны для своих пользователей. Тем не менее, нельзя пренебрегать тщательным тестированием разрабатываемых продуктов. Вы не станете проектировать машину, обслуживание которой предусматривает, чтобы один человек оперировал двумя переключателями, расположенными в трёх метрах друг от друга. Очевидно, что людей с такими физическими размерами не бывает.

Большая часть машин, созданных нашей цивилизацией, были механическими и взаимодействовали с нами главным образом физически. Соответственно, наши физические ограничения сравнительно легко учесть. Постепенно человеческие изобретения стали иметь всё большее отношение к области интеллектуальных задач, нежели физических.

Для создания хороших интерфейсов необходимо овладеть «эргономикой сознания». Мы часто не замечаем собственные ментальные ограничения, поэтому для определения границ возможностей нашего сознания мы должны прибегнуть к тщательному наблюдению и экспериментированию.

Изучение прикладной сферы наших ментальных способностей называется когнитивным проектированием, или когнетикой.

> ***Когнетика*** *или* ***когнитивное проектирование*** *— это научная дисциплина, изучающая прикладные сферы ментальных способностей человека.*

Некоторые когнитивные ограничения очевидны: например, нельзя ожидать от обычного пользователя способности перемножать в уме 30-значные числа за 5 секунд, поэтому нет смысла разрабатывать интерфейс, который требовал бы от пользователя такой способности. Однако мы часто не учитываем другие ментальные ограничения, которые оказывают неблагоприятное влияние на нашу продуктивность при работе с интерфейсами «человек — машина», хотя эти ограничения присущи каждому человеку. Интересно отметить, что все известные компьютерные интерфейсы, а также многие некомпьютерные интерфейсы «человек — машина» разработаны с рассчётом на некие когнитивные способности, которыми, как показывают эксперименты, мы на самом деле не обладаем. Большая часть трудностей, связанных с использованием компьютеров и подобных устройств, возникает скорее из-за низкого качества интерфейса, чем из-за сложности самой задачи или же недостатка старания или умственных способностей у пользователя. Есть ещё предположение, что это может быть спланированный маркетинговый ход (к этому часто прибегает Microsoft, с целью провести масштабное платно переобучение всех пользователей).

Когнетика, так же как и эргономика, учитывает статистическую природу различий между людьми. Тем не менее, следует прежде всего рассмотреть сами ограничения, присущие нашим когнитивным способностям.

> ***Локус*** *— область (предмет) сознательного внимания.*

Термин **фокус**, который иногда используется в этом контексте, может вызвать неправильное представление о том, как работает внимание, потому что может быть понят как действие. Когда Вы находитесь в бодрствующем и сознательном состоянии, Вашим локусом внимания является какая-то деталь или объект окружающего мира или идея, о которой Вы целенаправленно и активно думаете. Различие между **фокусом** и **локусом** внимания можно понять на примере следующего предложения: Мы можем целенаправленно сфокусировать наше внимание на каком-либо **локусе**. Тогда как **фокусировать** означает волевое действие, мы, тем не менее, не

можем полностью управлять содержанием локуса нашего внимания. Если Вы слышите, как позади вас внезапно взорвалась петарда, Ваше внимание будет направлено на источник звука.

Фокус *— выбранный на экране объект.*

Ваше внимание может быть, или не быть направлено на такого рода **фокус**, когда Вы пользуетесь тем или иным интерфейсом. Из всех объектов или явлений окружающего мира, которые Вы воспринимаете с помощью своих чувств или воображения, в каждый момент времени Вы можете сконцентрироваться только на одном. Чем бы ни был этот объект, деталь, воспоминание, мысль или понятие, он становится локусом Вашего внимания. В данном случае имеется в виду не только то внимание, которое можно активно обращать на что-либо, но также и пассивное восприятие потока поступающей информации или просто переживание происходящего.

Когда Вы выполняете какую-то задачу многократно, то с каждым разом делать это становится всё проще.

Задача дизайнеров в том, чтобы создавать интерфейсы, которые не позволяют привычкам вызывать проблемы у пользователей.

Печатать на компьютере слепым методом, так же как и ездить на велосипеде или ходить пешком по тропинке, лучше всего получается, если об этом не задумываться. Как только Вы задумаетесь, Вы можете сбиться. Чтобы совершать известные Вам действия, требуется всего лишь расслабить мышцы и нервы, которые отвечают за выполнение каждого отдельного шага, предоставить их самим себе и не вмешиваться в их работу. Конечно, это не означает, что Вы отказываетесь от собственной воли, потому что решение о совершении действия остается за Вами, и Вы можете в любой момент вмешаться, чтобы, например, изменить технику исполнения. Если Вы захотите, то можете научиться ездить на велосипеде задом наперед или ходить экстравагантной хромающей походкой, подпрыгивая на каждом четвертом шаге и одновременно насвистывая какую-нибудь мелодию (такими примерами напичкана книга рекордов Гиннесса). Но если Вы станете концентрировать свое внимание на деталях, на движении каждой мышцы, чуть ли не падая на каждом шаге и в последний момент вовремя выставляя ногу, чтобы все-таки не свалиться, то, в конце концов, Вы вообще не сможете двигаться и будете только дрожать от напряжения.

В случае идеального человекоориентированного интерфейса *доля участия самого интерфейса в работе пользователя должна сводится к формированию полезных привычек.*

Любая привычка означает отказ от внимания к деталям. Тем не менее, привычки необходимы всем высшим формам жизни, представленным на Земле. С другой стороны, жизнь возможна даже при отсутствии какого бы то ни было сознания, как, например, жизнь микробов — по крайней мере, насколько мы знаем или хотя бы можем предполагать.

С точки зрения когнитивной психологии,

> *любая задача, которую пользователь научился выполнять без участия сознания, становится для него* ***автоматичной****.*

Зачастую невозможно изменить привычку волевым действием. Как бы часто или настойчиво Вы не говорили себе не делать то или иное привычное действие, Вы не всегда можете остановить себя. Предположим, к примеру, что в следующее воскресенье педали тормоза и газа на Вашей машине поменяются местами. Специальная красная лампочка на приборной доске будет сигнализировать Вам об этом изменении. Хотя, возможно, Вам и удастся проехать несколько кварталов без аварии, тем не менее, большинство из нас в такой ситуации не смогло бы избежать ошибок. Как только Ваш локус будет отвлекаться от нововведения в конструкции машины, например, в случае если на дороге окажется ребенок, Ваша реакция, обусловленная привычкой, заставит вас нажать не на ту педаль. И даже специальная красная лампочка будет здесь бесполезна. Причина в том, что привычку нельзя изменить однократным волевым действием. Для этого требуется тренировка в течение некоторого периода времени. Разработчик может устроить, в том числе и ненамеренно, ловушку для пользователя, если сделает так, что на одном компьютере будут интенсивно использоваться два или более приложения, интерфейсы которых отличаются только несколькими часто применяемыми деталями. В таких обстоятельствах у пользователя, скорее всего, сформируются привычки, которые будут приводить к ошибкам при попытках применить в одном приложении команды, свойственные другому.

Человек, по-видимому, имитирует одновременное выполнение нескольких задач, требующих сознательного контроля, через последовательное переключение внимания с одной задачи на другую. Действительная одновременность достигается, когда все задачи, кроме разве что одной, становятся **автоматичными**. Например, Вы можете одновременно не спеша идти, что-нибудь есть и при этом решать какую-нибудь математическую задачу. (В это же время можно бессознательно обдумывать и ещё одну математическую задачу, но по определению когнитивного бессознательного Вы не заметите этого процесса. Вы не можете сознательно работать над двумя разными математическими задачами одновременно.) Для большинства людей все эти действия, за исключением поиска решения математической задачи, настолько знакомы, что могут выполняться «на автопилоте». Однако если при одновременном выполнении всех этих действий Вы внезапно почувствуете какой-нибудь неприятный на вкус кусочек Вашей походной еды, Вы станете думать только о том, что Вы такое съели, тогда как математическая задача перестанет быть Вами осознаваемой.

Не менее важным является тот факт, что человек не может избежать формирования автоматических реакций. Эта невозможность не зависит от повторения: никаким количеством повторений нельзя научиться не формировать привычки при регулярном использовании того или иного интерфейса. Формирование привычек является неотъемлемой частью нашего ментального аппарата. Его невозможно остановить волевым действием. Наверное, когда-нибудь в воскресенье утром Вы нечаянно приезжали на занятия, хотя собирались поехать в какое-то другое место. Сделали Вы это по привычке, которая сформировалась через повторение определённой последовательности действий. Когда Вы учились читать, то поначалу проговаривали по отдельности каждый слог и обращали внимание на произношение каждой буквы. Теперь же Вы можете читать без необходимости сознательного контроля над процессом составления слов из букв.

Любая последовательность действий, которую Вы регулярно выполняете, становится, в конце концов, **автоматичной**. Набор действий, составляющих последова-

тельность, становится как бы одним действием. Как только Вы начнёте выполнять некоторую последовательность, требующую не более 1 или 2 секунд времени, Вы не сможете остановиться и проделаете все действия вплоть до завершения последовательности. Вы также не сможете прервать последовательность, выполнение которой занимает больше нескольких секунд, если она не стала локусом внимания.

1.2. Режимы

Режимы (modes) являются важным источником ошибок, путаницы, ненужных ограничений и сложности в интерфейсе. Многие проблемы, создаваемые режимами, хорошо известны. Тем не менее, практика создания систем без режимов почему-то не находит широкого применения в разработке интерфейсов. Чтобы понимать **режимы**, следует сначала определить понятие **жеста**.

> ***Жест*** *— это последовательность действий человека, которая выполняется автоматически.*

В большинстве интерфейсов допускается несколько интерпретаций конкретного **жеста**. Например, в одном случае с помощью клавиши Return можно вставить в текст пустую строку (точнее, символ возврата каретки), тогда как в другом случае нажатие этой клавиши приводит к исполнению текстовой строки в качестве команды.

Режимы проявляются в том, как реагирует интерфейс на тот или иной **жест**.

> *Состояние интерфейса, при котором интерпретация данного конкретного жеста остаётся неизменной, называется* ***режимом****.*

Если **жест** получает другую интерпретацию, это значит, что интерфейс находится в другом **режиме**.

Например, исправный фонарик, управляемый с помощью однокнопочного выключателя, может быть либо включен, либо выключен. Два состояния фонарика соответствуют двум режимам интерфейса. В одном режиме нажатие кнопки включает свет. В другом режиме нажатие кнопки выключает свет. Если Вы не знаете текущее состояние фонарика, Вы не можете предсказать, к чему приведёт нажатие кнопки. Невозможность определить текущее состояние фонарика — это пример классической проблемы, возникающей в интерфейсе, в котором есть режимы.

По состоянию управляющего механизма невозможно сказать, какое действие следует выполнить, чтобы достичь поставленной цели. Оперируя с управляющим механизмом без одновременной проверки состояния системы, Вы не сможете предсказать, какой результат будет иметь данная операция.

Пример .38 (переключение раскладки клавиатуры).

- keyrus — драйвер клавиатуры для MS DOS. Неплохое решение: в режиме русской раскладки периметр экрана имеет ярко красную окраску, смена режима сопровождается звуковым сигналом.

- PuntoSwitcher — современное решения для автоматической смены режима (на основе синтаксического анализа и словаря исключений).

К переключателям трудно подобрать надписи. Например, интерфейс, в котором кнопка на экране подписана Lock (Блокировать). Когда пользователи в первый раз видели эту кнопку, они считали, что должны нажать на неё для блокировки данных в этом окне. Когда они делали так, подпись кнопки изменялась на Unlock (Разблокировать), показывая, что при нажатии на эту кнопку данные разблокировались бы. После этого многие пользователи удивлялись, что данные оказывались разблокированными, поскольку кнопка показывала Unlock, что они считали индикатором текущего состояния, как это часто бывает при использовании переключателей на кнопках и в меню. Естественно, это приводило к путанице: на самом деле кнопка показывала Lock, когда данные были заблокированы, и Unlock — когда они были разблокированы. Очевидно, что проблему нельзя решить, просто поменяв надписи таким образом, чтобы разблокированные данные обозначались как Unlock, а заблокированные как Lock. В данном случае следует, во-первых, использовать радиокнопку, а во вторых, применить более точные формулировки — Locked (Заблокировано) вместо Lock (Блокировка), что будет восприниматься более правильно: если флажок установлен, значит, данные заблокированы, если же флажок сброшен — данные не заблокированы. Для переключения режимов следует использовать *радиокнопки* с правильными подписями, обозначающими соответствующий режим. Переключатели стали стандартным средством для выбора одной из нескольких возможностей. Используйте переключатели вместо выключателей. Выключатели могут приводить к ошибкам. Такие ошибки обычно имеют кратковременный характер и легко исправляются. Тем не менее, их нельзя не учитывать при разработке интерфейсов. Выключатели можно использовать только в том случае, когда можно видеть значение контролируемого состояния и оно находится в локусе внимания пользователя или в кратковременной памяти. Изменения подписей в этом случае не требуется. Можно использовать более развёрнутую формулировку, например: «Щёлкните по этой кнопке, чтобы разблокировать данные» или даже: «Данные в настоящее время заблокированы. Щёлкните по этой кнопке, чтобы разблокировать их». Однако на кнопке или около флажка, или в меню трудно разместить полное объяснение, если интерфейс не снабжён функцией увеличения изображения. Для этих целей можно также использовать всплывающие подсказки.

Ларри Кларк ещё в 1979 г. заметил, что режимы создают проблемы по той причине, что привычные действия приводят к неожиданным результатам.

Пример .39 (Caps Lock). Трудность, которая особенно изматывает всех пользователей компьютеров, связана с клавишей «Caps Lock», присутствующей на большинстве типов клавиатур. Часто первым признаком случайного нажатия этой клавиши оказывается то, что Вы видите, что набранное Вами предложение напечатано заглавными буквами, и только после этого Вы замечаете также, что включён индикатор «Caps Lock», находящийся на противоположном конце клавиатуры.

> Ошибки *пользователя, связанные с режимами, называют* ***модальными.***

Для предотвращения **модальных ошибок** надо *не использовать режимы.* Если не удаётся этого избежать, то следует обеспечить чёткое различие между режимами. Можно, например изменять форму курсора.

Не использование одинаковых команд в разных режимах уменьшает последствия модальных ошибок.

Данный элемент интерфейса может для одного пользователя быть модальным, а для другого — нет. Более полное определение режима должно включать в себя то, как пользователь рассматривает интерфейс.

Интерфейс является модальным по отношению к данному жесту, если:

1. текущее состояние интерфейса не находится в локусе внимания пользователя;
2. в ответ на некоторый жест интерфейс может выполнить одно из нескольких возможных действий, в зависимости от текущего состояния системы.

Интерфейс может быть *модальным* по отношению к одному жесту и *немодальным* по отношению к другому.

> ***Истинно немодальный интерфейс*** *должен быть немодальным для любого жеста.*

Пример .40 (немодальный интерфейс). *Горячие клавиши* в FAR не зависят от языка ввода.

Степень модальности Q интерфейса определяется через классификацию каждого жеста, допустимого в интерфейсе, как модального или немодального. С учётом вероятности $p(N_i)$ применения данного немодального жеста N_i и вычисленной для данного пользователя (группы пользователей) имеем

$$Q = \sum_i p(N_i).$$

Q изменяется от 0 (полностью модальный) до 1 (полностью немодальный).

> *Набор состояний, в которых конкретный жест имеет конкретную интерпретацию, называют* ***диапазоном этого жеста****.*

Квазирежимы

Удерживание кнопки в нажатом состоянии (Shift), т. е. физическое удержание интерфейса в определённом состоянии, не приводит к возникновению *модальных* ошибок (в отличие от Caps Lock).

Для обозначения режимов, которые удерживаются пользователем кинестетически, *Дж. Раскин* ввёл термин **квазирежим** и его прилагательное **квазимодальный**.

> *Режим, удерживаемый пользователем называют* ***квазирежимом****.*

Квазирежимы являются эффективным средством борьбы с *модальными* ошибками. Для сохранения эффективности *количество* квазирежимов *должно быть от 3 до 7.*

Пример .41 (квазирежимы).

- Выпадающие меню по наведению мышки с зажатой кнопкой (на нужном пункте кнопка отпускается) [Macintosh а не Windows].
- Циклические списки (при неизменном порядке) [Adobe, KDE, Gnome, . . . , а не Windows].
- Адаптивные меню или палитры (исходя из того предположения, что если оставлять часто используемый элемент на виду, без необходимости поиска его в меню, то это может ускорить работу пользователя). В первом методе выбранный элемент убирается из общего списка и помещается в основную палитру или меню. Во втором, более правильном методе выбранный элемент копируется в основную палитру или меню.

Монотонность

> ***Монотонность*** *— свойство интерфейса, выполняющего задачу единственным способом.*

Общераспространённым мифом является то, что интерфейсы для начинающих и опытных пользователей системы должны быть совершенно различными.

Чем более монотонным является интерфейс применительно к конкретной задаче, тем легче у пользователя формируется *автоматичность* её выполнения.

Когда требуется сделать выбор между несколькими методами, Ваш локус внимания временно смещается с текущей задачи на принятие решения о выборе.

Составные модели

Если команда предусматривает применение некоторого *действия* к некоторому *объекту*, то рекомендуется в интерфейсе использовать *немодальную* последовательность — *объект-действие*.

В большинстве руководств по разработке интерфейсов рекомендуется именно модель взаимодействия существительное-глагол (Apple, 1987; Hewlett Packard, 1987; IBM, 1988; Microsoft. 1995). Анализ, проведённый с точки зрения локуса внимания пользователя, показывает преимущества такой модели.

Применение метода *действие-объект* должно ограничиваться только выбором из палитры для непосредственного использования.

Видимость и состоятельность

> *Элемент интерфейса считается* ***видимым****, если он либо доступен в данный момент для органов восприятия человека, либо он был недавно воспринят и сохранился в кратковременной памяти пользователя.*

Для нормальной работы интерфейса «должны быть видимы только необходимые вещи: те, что идентифицируют части работающих систем, и те, что отображают способ, которым пользователь может взаимодействовать с устройством. **Видимость** отображает связь между предпринимаемыми действиями и их реальной отдачей» (Norman, 1988).

Задача разработчика, — сделать каждый элемент интерфейса *видимым*.

Если каждая функция и способ её использования очевидны по одному внешнему виду для большинства людей из той культуры, на которую этот интерфейс ориентирован, то выполняется ***принцип видимости*** *для этого интерфейса.*

Если интерфейс вынуждает вас запоминать существование того или иного его элемента, это означает, что этот элемент является невидимым. Если Вы вынуждены углубляться в недра интерфейса, чтобы случайно или в результате собственной настойчивости натолкнуться на последовательность действий, которая активизирует какой-то его элемент, то этот элемент является невидимым. Если Вам приходится обращаться за помощью к справочной системе для того, чтобы узнать, как выполнить то или иное действие, это означает, что методы выполнения этого действия невидимы. Многие компьютерные игры являются, по сути дела, недокументированными интерфейсами, в которых способы управления или пути достижения желаемых результатов невидимы. Стоит добавить к этим играм документацию, и они станут неинтересными. Однако большинство людей не хотят играть в игры, когда им требуется выполнить свою работу, поэтому перед разработчиком интерфейсов стоит задача сделать каждый элемент своего продукта видимым.

Элемент управления, который имеет свойство видимости называется ***состоятельным****.*

«Состоятельность является хорошим средством для связки элемента интерфейса с его целью. Ручки используются для настройки; гнёзда — для вставки чего-то; мячи — для их бросания или пинания» (Norman, 1988). При анализе интерфейсов следует всегда спрашивать себя, каким образом пользователь может узнать, что то или иное действие возможно; также следует всегда стремиться к тому, чтобы каждый видимый элемент был состоятельным. Пиктограммы часто рассматриваются как символические обозначения состоятельности, но они не всегда выполняют эту функцию.

Руководство пользователя должно содержать однозначные инструкции. Так, например, для «горячих клавиш» можно ввести дополнительные обозначения (↓↑).

Пример .42 (↓↑). $Shift\downarrow\ n\ Control\downarrow\ k\ Shift\uparrow\ w\ Control\uparrow$; $Control\downarrow\ n\ \downarrow\uparrow\uparrow$.

1.3. Квантификация интерфейса

Модель скорости печати GOMS

Модель GOMS *— правила для целей, объектов, методов и выделений (the model of Goals, Objects, Methods and Selection rules).*

Номенклатура модели GOMS:

$K = 0{,}2\,\mathbf{с}$ — *нажатие клавиши.*

$P = 1{,}1\,\mathbf{с}$ — *указание.* Время, необходимое пользователю для того, чтобы указать на какую-то позицию на экране.

$H = 0{,}4\,\mathbf{с}$ — *перемещение.* Время, необходимое пользователю для того, чтобы переместить руку с клавиатуры на графический манипулятор или наоборот.

$M = 1{,}35\,\mathbf{с}$ — *ментальная подготовка.*

R — *ответ.* Время, в течение которого пользователь должен ожидать ответ от компьютера.

На практике указанные значения могут варьироваться в широких пределах. Для опытного пользователя, способного печатать со скоростью 135 слов/мин., значение K может составлять 0,08 с, для обычного пользователя, имеющего скорость 55 слов/мин., — 0,2 с, для среднего неопытного пользователя, имеющего скорость 40 слов/мин., — 0,28 с, а для начинающего — 1,2 с. Нельзя сказать, что скорость набора не зависит от того, что именно набирается. Для того чтобы набрать одну букву из группы случайно взятых букв, большинству людей требуется около 0,5 с. Если же это какой-то запутанный код (например, адрес электронной почты или код активации), то у большинства людей скорость набора составит около 0,75 символов в секунду. Значение K включает в себя и то время, которое необходимо пользователю для исправления сразу замеченных ошибок. Клавиша «Shift» считается за отдельное нажатие.

Широкая изменяемость каждой из представленных мер объясняет, почему эта упрощённая модель не может использоваться для получения абсолютных временных значений с какой-либо степенью точности. Тем не менее, с помощью типичных значений мы можем сделать правильную сравнительную оценку между какими-то двумя интерфейсами по уровню эффективности их использования. Если оцениваются сложные интерфейсы, включающие пересекающиеся временные зависимости, или если должны быть с точностью достигнуты определённые временные интервалы, то следует применять более сложные модели (например, CPM-GOMS (critical-path method) или NGOMSL (natural GOMS language)).

Вычисление времени, необходимого на выполнение требуемого действия начинаются с перечисления операций из списка жестов модели GOMS.

Правила расстановки ментальных операций:

0. *Начальная расстановка операторов M.* Оператор M следует устанавливать перед всеми операторами K, а также перед всеми операторами P, предназначенными для выбора команд (перед операторами P, предназначенными для указания на аргументы этих команд, M ставить не следует).

1. *Удаление ожидаемых операторов M.* Если следующий после M оператор является ожидаемым относительно предшествующему M оператору, то этот оператор M убирается (например, $P\,M\,K \to P\,K$).

2. *Удаление операторов M внутри когнитивных единиц.* Если строка вида $M\ K\ M\ K\ M\ K\ \ldots$ принадлежит когнитивной единице, то следует удалить все операторы M, кроме первого.

   > ***Когнитивной единицей*** *является непрерывная последовательность вводимых символов, которые могут образовывать название команды или аргумент или просто связанный текст.*

3. *Удаление операторов M перед последовательными разделителями.* Если оператор K означает лишний разделитель, стоящий в конце когнитивной единицы (например, разделитель команды, следующий сразу за разделителем аргумента этой команды), то следует удалить оператор M, стоящий перед ним.

Разделителем считается символ, которым обозначено начало или конец значимого фрагмента текста, такого как, например, слово естественного языка или телефонный номер. Например, пробелы являются разделителями для большинства слов. Точка является наиболее распространённым разделителем, который используется в конце предложений. Скобки используются для ограничения пояснений и замечаний и т. д.

Операторами являются K, P и H. Если для выполнения команды требуется дополнительная информация (как, например, в случае когда для установки будильника пользователю требуется указать время его включения), эта информация называется *аргументом* данной команды. *Удаление операторов M, — прерывателей команд.* Если оператор K является разделителем, стоящим после постоянной строки (например, название команды), следует удалить оператор M, стоящий перед ним. В этих правилах под строкой понимается некоторая последовательность символов. *Удаление перекрывающих операторов M.* Любую часть оператора M, которую перекрывает оператор R (ожидание ответа компьютера), следует исключить.

Измерение эффективности интерфейса

Для начала следует построить модель интерфейса. GOMS позволяет оценить время, необходимое пользователю на выполнение любой чётко сформулированной задачи.

Однако модели анализа не могут дать ответ на вопрос о том, насколько быстро *должен* работать интерфейс. Следующий вопрос, — определить, насколько быстро работает тот интерфейс, который отвечает поставленным требованиям. Чтобы ответить на него, мы можем воспользоваться мерой, применяемой в теории информации.

4. ***Информационная производительность интерфейса*** *E — это отношение минимального количества информации, необходимого для выполнения задачи, к количеству информации, которую должен ввести пользователь.*

Чтобы сделать правильную оценку времени, необходимого на выполнение задачи с помощью самого быстрого интерфейса, прежде всего следует определить минимальное количество информации, которое пользователь должен ввести, чтобы выполнить задачу. Это минимальное количество не зависит от модели интерфейса.

Если методы работы, используемые в предполагаемом интерфейсе, требуют введения такого количества информации, которое превышает минимальное, это означает, что пользователь делает лишнюю работу, и поэтому интерфейс можно усовершенствовать. С другой стороны, если от пользователя требуется ввести именно то количество информации, которое необходимо для выполнения задачи, то для этой задачи интерфейс нельзя сделать более производительным путём изменения количества информации. В этом случае пути улучшения интерфейса (а также много путей для ухудшения) всё же остаются, но по крайней мере данная цель повышения производительности будет уже достигнута.

E может изменяться в пределах от 0 до 1: $E = 0$ — в случае, когда пользователь должен ввести совершенно бесполезную информацию (как в случае окна сообщения *Alert* JavaScript); $E = 1$ — в случае, когда работа не производится (как в случае прозрачного окна сообщения).

В параметре E учитывается только информация, необходимая для задачи, и информация, вводимая пользователем. Два или более методов действия могут иметь одинаковую производительность E, но иметь разное время выполнения. Возможно даже, что один метод имеет более высокий показатель E, но действует медленнее, чем другой метод.

Пример .43. $M\,K\,M\,K$ и $M\,K\,K\,K$

При использовании первого метода должно быть введено только два символа. При использовании второго метода требуется ввести три символа, но времени на всё действие тратится меньше.

Как правило, чем более производительным является интерфейс, тем более продуктивным и более человекоориентированным он является.

Из теории информации известно, что информация измеряется в битах. Один бит представляет собой один из двух альтернативных вариантов (таких как `0` или `1`, `да` или `нет`). Двух двоичных выборов, или двух битов, достаточно для выбора одного элемента из четырёх. Чтобы сделать выбор из группы восьми элементов, потребуется 3 бита. Из шестнадцати элементов — 4 бита, и т. д. В общем случае при количестве n равновероятных вариантов суммарное количество передаваемой информации определяется как b: $2^b = n$.

Количество информации (по Шеннону) для каждого равновероятного варианта из n возможных:

$$\frac{1}{n}\log_2 n. \tag{12.1}$$

В общем случае, если $p(i)$ — вероятность i-ой альтернативы, то количество информации для i-го варианта:

$$p(i)\log_2\left(\frac{1}{p(i)}\right). \tag{12.2}$$

Так, для двух равновероятных альтернатив ($p = 1 - p = 0{,}5$):

$$E = p\log_2\left(\frac{1}{p}\right) + (1-p)\log_2\left(\frac{1}{1-p}\right) = 0{,}5\cdot 1 + 0{,}5\cdot 1 = 1. \tag{12.3}$$

Если $p \neq 0{,}5$, то $E < 1$.

Мы можем оценить объём информации, содержащейся в сообщении, только в контексте всего набора возможных сообщений. Чтобы подсчитать количество информации, передаваемой некоторым полученным сообщением, необходимо знать вероятность, с которой это сообщение может быть отправлено. Количество информации в любом сообщении не зависит от других сообщений, которые были в прошлом или могут быть в будущем, не связано со временем или продолжительностью и не зависит от каких-либо иных событий.

Нельзя путать понятие информации с понятием смысла, информация является мерой свободы выбора сообщения.

Однако действия, которые совершает пользователь при выполнении задачи, можно с большей точностью смоделировать в виде процесса Маркова, в котором вероятность последующих действий зависит от уже совершённых пользователем действий. Тем не менее, для данного рассмотрения достаточно использовать упомянутые вероятности отдельных, единичных событий, при этом будем исходить из того, что все сообщения являются независимыми друг от друга и равновероятными.

Провести быстрый анализ интерфейса можно путём вычисление различных жестов на основе количества информации, передаваемого одним нажатием клавиши или одной операцией мыши. При передаче информации нажатием клавиши её количество зависит от общего количества клавиш и относительной частоты использования каждой из них. Если на клавиатуре имеется 128 клавиш, и каждая из них используется с одинаковой частотой, то нажатие любой из них будет передавать 7 бит информации. В действительности частота использования клавиш существенно различается. Например, пробел или буква `e` используются чаще, чем `й` или \, поэтому в большинстве приложений на каждое нажатие клавиши приходится в среднем около 5 битов.

Для анализа эффективности интерфейса часто используют более простую меру, чем E, — **символьную эффективность**.

> ***Символьная эффективность интерфейса*** *— это отношение минимального количества символов, необходимого для выполнения задачи, к количеству символов, которое в данном интерфейсе должен ввести пользователь.*

Закон Фиттса (Fitts' Law) позволяет определить количественно тот факт, что чем дальше находится объект от текущей позиции курсора, или чем меньше размеры этого объекта, тем больше времени потребуется пользователю для перемещения к нему курсора.

Представим, что Вы перемещаете курсор к кнопке, изображённой на экране. Кнопка является целью данного перемещения. На основе данных о размерах объекта и расстоянии от начальной позицию курсора до ближайшей точки кнопки закон Фиттса позволяет найти среднее время, за которое пользователь сможет переместить курсор к кнопке.

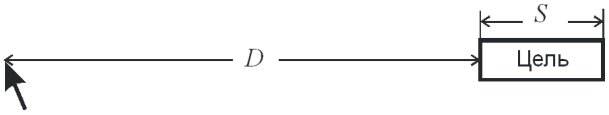
Закон 1 (Фиттса)**.** *Время достижения цели прямо пропорционально дистанции до цели и обратно пропорционально размеру цели.*

Paul Fitts. The Information Capacity of the Human Motor System in Controlling Amplitude of Movement, 1954.

Пол Фиттс был первым главой психологического отдела Аэродинамической лаборатории ВВС США и профессором двух университетов (много сделал для авиационной безопасности). Закон был сформулирован в 1954, а впервые применён в психологии взаимодействия компьютер-человек в 1978 году (также применён ко многим указательным устройствам таким как мыши, джойстики, и т. п.), но к сожалению *Пол Фиттс* не дожил до этих дней и умер в 1965 в возрасте 53 лет.

Определим длину прямой, соединяющую начальную позицию курсора и ближайшую точку целевого объекта, как *дистанцию*.

Рассмотрим одномерный случай.



Размер объекта вдоль линии перемещения курсора обозначим S, а дистанцию — D.

Закон 2 (Фиттса)**.** *Время от момента, когда курсор начинает движение к целевому объекту до момента, когда пользователь щёлкает по нему мышью находится, как*

$$t[мс] = a + b \log_2 \left(\frac{D}{S} + 1 \right), \tag{12.4}$$

константы a и b устанавливаются опытным путём по параметрам производительности человека.

Демонстрация закона Фиттса.

Пример .44 (сравнение эффективности меню Apple Macintosh и MS Windows). Положим $a = 50$ и $b = 150$ (Джеф Раскин). В Macintosh меню очень крупное и располагается *на краю экрана*. По измерениям, пользователи, в среднем останавливают курсор в 50 мм от границы экрана. В Windows меню имеет размер (в основном) 5 мм. Для 14"монитора среднее расстояние перемещения курсора $D = 80$ мм.

Таким образом для Macintosh:

$$t_{\text{Mac}} = 50 + 150 \log_2 \left(\frac{80}{50} + 1 \right) = 257 \text{ мс}.$$

Для Windows:

$$t_{\text{Win}} = 50 + 150 \log_2 \left(\frac{80}{5} + 1 \right) = 663 \text{ мс}.$$

Закон Хика (Hick's Law) позволяет количественно определить тот факт, что чем больше количество вариантов заданного типа предоставляется пользователю, тем больше времени требуется на выбор.

Время выбора одного из n равноправных вариантов:

$$t[\text{мс}] = a + b \log_2(n + 1). \tag{12.5}$$

Если вероятность выбора i-го варианта равна $p(i)$, то

$$t[\text{мс}] = a + b \sum_i p(i) \log_2 \left(\frac{1}{p(i)} + 1 \right). \tag{12.6}$$

Здесь a, b зависят от многих условий, в том числе и от того, насколько хорошо пользователь знаком с системой (наличие навыков и привычек снижают b) и от того, как представлены варианты.

При отсутствии более точных данных для проведения быстрых и приблизительных вычислений можно воспользоваться теми же значениями a и b, что и для закона Фиттса.

Независимо от значений a и b из закона Хика следует, что предоставление пользователю сразу всех вариантов одновременно является более эффективным, чем организация этих вариантов в иерархическое меню.

Пример .45 (эффективность меню). Сравним эффективности меню с 8 элементами и двухуровнего меню (по 4 элемента на каждом уровне).

$$t_8 = a + b \log_2 8 = a + 3b,$$

$$t_{2\times4} = 2(a + b \log_2 4) = 2(a + 2b).$$

Очевидно, что $t_8 < t_{2\times4}$.

При проектировании меню следует руководствоваться правилом «5 ± 2»: количество элементов на одном уровне должно быть от 3 до 7.

Это правило связано с возможностями кратковременной памяти среднестатистического человека, оно является универсальным (используется во многих областях).

Рассмотрение **законов Фиттса** и **Хика** нельзя считать полным. Например, следует обратить внимание на то, что эти законы не случайно принимают ту же форму, что и теорема Шэннона-Хартли (Shannon-Hartley). Тем не менее, этого рассмотрения вполне достаточно для того, чтобы отметить их ценность с точки зрения разработки интерфейсов. Они могут быть полезными даже в том случае, когда эмпирические значения коэффициентов a и b не известны.

Теорема Шеннона-Хартли — применение теоремы кодирования канала с шумом к архетипичному случаю непрерывного временного аналогового канала коммуникаций, искажённого гауссовским шумом. Теорема устанавливает шенноновскую ёмкость канала, верхнюю границу максимального количества безошибочных цифровых данных (то есть, информации), которое может быть передано по такой связи коммуникации с указанной полосой пропускания в присутствии шумового вмешательства, согласно предположению, что мощность сигнала ограничена, и гауссовский шум характеризуется известной мощностью или мощностью спектральной плотности.

Рассматривая все возможные многоуровневые и многофазные методы шифрования, теорема Шеннона-Хартли утверждает, что пропускная способность канала C, означающая теоретическую верхнюю границу скорости передачи данных, которые можно передать с данной средней мощностью сигнала S через аналоговый канал связи, подверженный аддитивному белому гауссовскому шуму мощности N равна:

$$C = B \log_2 \left(1 + \frac{S}{N} \right), \tag{12.7}$$

где C — пропускная способность канала, бит/с; B — полоса пропускания канала, Гц; S — полная мощность сигнала над полосой пропускания, Вт; N — полная шумовая мощность над полосой пропускания, Вт; S/N — частное от деления отношения сигнала к его шуму (SNR) на гауссовский шум, выраженное как отношение мощностей.

2. UX/UI

> ***UX*** *(User eXperience, опыт взаимодействия пользователя) — это восприятие пользователем всех аспектов системы (сайта, приложения, продукта, услуги, сообщества и т. д.).*

Компании стремятся добиться позитивных, последовательных, прогнозируемых и желаемых результатов при помощи пользовательского опыта, в состав которого может входить интерфейс, промышленный дизайн, физическое взаимодействие и многое другое.

UI-дизайн, как правило, играет важную роль в работе UX-дизайнера, однако это не единственный её элемент. *Опыт пользователя (UX) — это не пользовательский интерфейс (UI)!*

$$\text{UX} \neq \text{UI}.$$

UX-дизайн представляет собой многоступенчатый процесс стратегического проектирования, целью которого является создание продукта, который потребители/пользователи считают привлекательным, лёгким в использовании и понятным. Используя UX-дизайн, мы в конечном итоге получаем оптимальный для нас пользовательский интерфейс.

В комплексном процессе UX-проектирования можно выделить как минимум 10 шагов, необходимых для создания готового пользовательского интерфейса:

- Анализ коммерческих целей и технических характеристик.
- Отчёты о результатах анализа конкурентной среды.
- Разработка персонажей и изучение опыта пользователя.
- Карта сайта и информационная архитектура.
- Карты пользовательского опыта, пути и потоки пользователей.
- Эскизы и каркасные модели.
- Макеты и проектирование взаимодействия.
- Интерактивные прототипы.
- Тестирование удобства использования.
- Окончательный визуальный дизайн.

3. Проектирование экранных форм электронных документов

Под электронными формами документов понимается не изображение бумажного документа, а изначально электронная (безбумажная) технология работы; она предполагает появление бумажной формы только в качестве твёрдой копии документа.

> ***Электронная форма документа*** *(ЭД) — это страница с пустыми полями, предназначенная для заполнения пользователем.*

Формы могут допускать различный тип входной информации и содержать командные кнопки, флажки, переключатели, списки.

Создание форм электронных документов требует использования специального программного обеспечения. На сегодняшний день наиболее удобной программой визуального проектирования форм является NetBeans фирмы Oracle.

К недостаткам электронных документов можно отнести юридическую неполноту, хотя разработан единый процесс их утверждения (электронной подписи), не во всех случаях она применима.

Технология обработки электронных документов требует использования специализированного программного обеспечения — программ *управления документооборотом*, которые зачастую встраиваются в корпоративные ИС.

Проектирование форм электронных документов включает в себя следующие этапы:

определение перечня макетов экранных форм — проектировщик анализирует постановку каждой задачи, в которой приводятся перечни используемых входных и выходных документов;

определение содержания макетов — выполняется на основе анализа состава реквизитов первичных документов с *постоянной* и *оперативной* информацией и *результатных* документов.

создание структуры ЭД — подготовка внешнего вида с помощью графических средств проектирования, задание различных способов введения данных (поля, списки, селекторы, ...);

определение содержания формы ЭД — обеспечение списков данными (составление списков, составление и связывание запросов к БД, ...);

программирование и тестирование макетов экранных форм.

4. Макетирование

Достаточно часто заказчик не может сформулировать подробные требования по вводу, обработке или выводу данных для будущего программного продукта. С другой стороны, разработчик может сомневаться в совместимости продукта с операционной

системой и др. ИС, в пользовательском интерфейсе или в эффективности реализуемого алгоритма. В этих случаях целесообразно использовать **макетирование**. Основная цель **макетирования** (**прототипирования**) — снять неопределённости в требованиях заказчика.

> ***Макетирование (прототипирование)*** *— это процесс создания модели требуемого программного продукта.*

Модель может принимать одну из трёх форм:

1. *бумажный макет или макет на основе ПК* (изображает работу программной системы или человеко-машинное взаимодействие);
2. *работающий макет* (выполняет некоторую часть требуемых функций);
3. *существующая программа* (характеристики которой затем должны быть изменены).

Как показано на рис. 12.1, макетирование основывается на многократном повторении итераций, в которых участвуют заказчик и разработчик.

Рис. 12.1. Последовательность действий при макетировании

Макетирование начинается со сбора и уточнения требований к создаваемой ИС. Разработчик и заказчик встречаются и определяют все цели ИС, устанавливают, какие требования известны, а какие предстоит доопределить. Затем выполняется быстрое проектирование. В нём внимание сосредоточивается на тех характеристиках ИС, которые должны быть видимы пользователю. Быстрое проектирование приводит к построению макета. Макет оценивается заказчиком и используется для уточнения требований к ИС. Итерации повторяются до тех пор, пока макет не выявит все требования заказчика и, тем самым, не даст возможность разработчику понять, что должно быть сделано.

Достоинство макетирования: *обеспечивает определение полных требований к ИС.*

Недостатки макетирования:

- разработчик может принять макет за продукт;
- заказчик может принять макет за продукт.

Когда заказчик видит работающую версию ИС, он перестаёт сознавать, что детали макета скреплены «жевательной резинкой и проволокой»; он забывает, что в погоне за работающим вариантом оставлены нерешёнными вопросы качества и удобства сопровождения ИС. Когда заказчику говорят, что продукт должен быть перестроен, он начинает возмущаться и требовать, чтобы макет «в три приёма» был превращён в рабочий продукт. Очень часто это отрицательно сказывается на управлении разработкой ИС.

4.1. Инструменты быстрого прототипирования

Для того чтобы оценить инструменты или техники прототипирования, в первую очередь необходимо определить критерии эффективного прототипа. Лучшими являются те прототипы, которые вливаются в процесс дизайна. Мы хотим иметь возможность быстро переходить от набросков к интерактивному воплощению.

Эффективные прототипы быстры. Мы хотим использовать методики, которые помогают делать итерации короче. Этап создания прототипа не должен быть жёстко закреплён в конце процесса дизайна. Включение создания прототипа в ежедневную дизайнерскую работу способствует возникновению идей и быстрой проверке концепций.

Эффективные прототипы являются одноразовыми. Как и любой другой дизайн, предназначенный для реализации, — это артефакт, предназначенный для того, чтобы донести идею до кого-то ещё (заинтересованного лица, разработчика, пользователя и т.д.). После того как дизайнерская идея была донесена, прототип реализации может быть отвергнут. Мы не должны переживать тяжёлое чувство, что создаём шедевр, который обязательно будет реализован, и безусловно не должны создавать прототип, работая на уровне кода.

Эффективные прототипы являются сфокусированными. Мы хотим выбирать интерактивные элементы дизайна, которые действительно должны быть прототипированы. Следует искать части своего дизайна, которые сложны. Ищите паттерны взаимодействия, которые давно известны в теории *опыта взаимодействия пользователя*. Ищите элементы взаимодействия, которые принесут пользу вашему продукту. Прототип, который продемонстрирует эти элементы, будет наилучшим.

Классический прототип представляет из себя набор страниц, связанных друг с другом ссылками.

В большинстве студий и агентств прототипы ограничивается базовыми экранами и состояниями. Их выбирают по основным пользовательским сценариям, например, поиск товара в интернет-магазине и покупка. Второстепенные сценарии на прототипах часто пропускают либо показывают поверхностно. Всё потому, что небольшие компании не могут себе позволить отвести лишнюю неделю работы специалиста, чтобы добавить несколько состояний.

Диазайнеры изучают результаты анализа поведения клиента (пользователя), проводят свои исследования, выясняют основной принцип взаимодействия: мобильные или десктопные устройства. Исходя из этого, выбирается базовое разрешение.

Затем отрисовываваются в базовом разрешении все состояния страниц, явный и неявный функционал, а также не очевидные, но возможные, сценарии. В остальных разрешениях, как правило, готовятся только ключевые и нестандартные страницы.

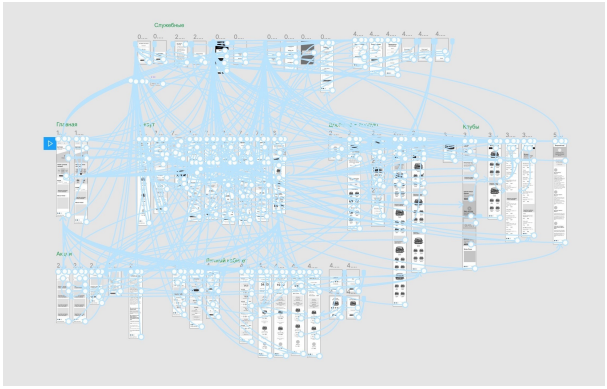


Рис. 12.2. Прототип мобильной версии сайта

Наиболее популярны следующие инструменты макетирования:

- Axure RP — инструмент, ориентированный на создание прототипов веб-сайтов (Axure.com). Генерирует кликабельный HTML и документацию в формате Word. Поддерживает комплексное взаимодействие. Работает на платформе Windows и Mac OS X, гибкая стоимость (от $29 в месяц на пользователя до $895 для командной работы). Актуальная версия 9.

- Balsamiq Wireframes — позволяет очень быстро создавать макеты ПО (www.balsamiq.com). Сгенерированное содержимое выглядит как скетчи, что поможет не вводить в заблуждение тех, кто может подумать, что программа уже готова. Предназначен прежде всего для прототипирования мобильных приложений. В нём представлены шаблоны для нескольких ОС и некоторых популярных смартфонов. Работает как облачный сервис (Balsamiq Cloud, 30 дней триал, далее от $9 в месяц или $90 в год до $199 в месяц или $1 990 в год), десктоп приложение для Windows и Mac OS (от $89 для одного пользователя до $9 790 для 220 пользователей) или плагин для Google Drive, Confluence, Jira (за $5 в месяц или $50 в год).

- Cacoo — инструмент, позиционируемый, как `cloud-based diagrams` (cacoo.com), позволяет создавать диаграммы различного типа. Для макетирования следует использовать раздел Wireframe. Для веб-разработки также полезен будет раздел Sitemap. Работает через браузер, использую *HTML5*. Стоимость сервиса: $4,95 в месяц (для одного пользователя), $15 в месяц (для командной работы), бесплатно (через Google Drive).

- Adobe XD (Adobe Experience Design) — ПО для разработки интерфейсов от Adobe Systems. Поддерживает векторную графику и веб-вёрстку, позволяет создавать небольшие активные прототипы.

- Sketch — векторный графический редактор для macOS, разработанный голландской компанией Bohemian Coding. Используется для проектирования интерфейсов мобильных приложений и веб-сайтов. Поддерживает возможность

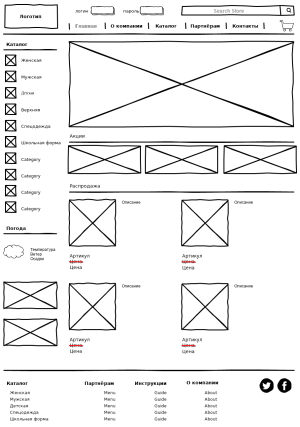


Рис. 12.3. Макет в Cacoo

создания интерактивных прототипов. Sketch имеет большие возможности интеграции с другим ПО и веб-сервисами InVision, Marvel, Jira, Zeplin и Avocode.

- Pencil Project — бесплатный сервис с открытым исходным кодом. В него встроены коллекции популярных элементов, есть набор готовых классических решений. Инструмент используется для создания макетов мобильных приложений и веб-страниц.

- Figma — онлайн-сервис для разработки интерфейсов и прототипирования с возможностью организации совместной работы в режиме реального времени. Сервис имеет широкие возможности для интеграции с корпоративным мессенджером Slack и инструментом для высокоуровневого прототипирования Framer. Позиционируется создателями как основной конкурент программным продуктам компании Adobe.

  Сервис доступен по подписке, предусмотрен бесплатный тарифный план для одного пользователя. Ключевой особенностью *Figma* является её облачность, у сервиса нет оффлайн-версии. За счёт этого также достигается принцип кроссплатформенности, который не могут гарантировать ближайшие конкуренты — Sketch и Adobe XD.

  Figma подходит как для создания простых прототипов и дизайн-систем, так и сложных проектов (мобильные приложения, порталы). В 2018 году платформа стала одним из самых быстро развивающихся инструментов для разработчиков и дизайнеров. В ноябре 2019 года появилась функция *Auto Layout*.

  Устойчивая модель развития *Figma*, упростившая сотрудничество во всём процессе создания цифровых продуктов для дизайнеров, разработчиков, менеджеров и маркетологов, позволила в начале 2018 года привлечь дополнительные 25 миллионов долларов инвестиций от партнёров.

  В 2019 году создатели редактора привлекли 40 миллионов долларов инвестиций от венчурного фонда Sequoia Capital при оценке самой компании в 400 миллионов долларов. Решение об инвестициях было принято на основе факта, что около половины портфельных компаний фонда используют редактор

в работе. К началу 2019 года *Figma* вышла на 1 миллион зарегистрированных пользователей, став серьёзным конкурентом для популярных средств прототипирования.

Глава 13

Обеспечение качества программного обеспечения

1. Основные понятия

Рассмотрим понятие качества ПО и все аспекты его обеспечения: цели, факторы, количественный подход к определению уровня качества, деятельности по обеспечению качества.

В словаре программной инженерии IEEE Std 610.12-90 «IEEE Standard Glossary of Software Engineering Terminology» есть два определения, придуманные двумя отцами-основателями, архитекторами современного представления о качестве — Филиппом Кросби и Джозефом Джураном. Первое:

> ***Качество*** *— это степень соответствия системы, компонента или процесса определённым требованиям.*

В 1979 году Ф. Кросби писал: «**качество** означает соответствие требованиям».

Определение Кросби отсылает к степени соответствия написанного ПО тем спецификациям требований, которые были подготовлены заказчиком. Это означает, что погрешности спецификации не учитываются и не снижают программное качество, оцениваемое в пределах погрешностей подхода.

Второе определение:

> ***Качество ПО*** *— это степень соответствия системы, компонента или процесса нуждам или ожиданиям заказчика, пользователя.*

Дж. Джуран в 1988 году ввёл такое определение: «1) **качество** определяется теми характеристиками продукта, которые удовлетворяют потребности заказчика и таким образом обеспечивают удовлетворение продуктом; 2) **качество** означает отсутствие недостатков».

Определение Джурана нацелено на полное удовлетворение заказчика и объявляет истинной целью достижения качества *удовлетворение реальных нужд заказчика.* В этом случае разработчик обязан приложить существенные усилия к исследованию и корректировке спецификации требований заказчика. Основное допущение этого

определения состоит в том, что заказчик освобождается от любой ответственности за точность и завершённость программной спецификации. Мало того, заказчику позволяют выражать его реальные потребности (которые могут существенно отличаться от проектной спецификации) на весьма поздней стадии проекта, даже на заключительной стадии. Как результат, затруднения могут возникнуть на протяжении всего процесса разработки, особенно при попытках доказать, насколько хорошо программа реализует потребности заказчика.

Дополнительные черты качества включены в определение Роджера Прессмана (2000 г.):

> ***Качество ПО*** *— это соответствие точно определённым функциональным требованиям и требованиям производительности, точно документированным стандартам разработки, а также подразумеваемым характеристикам, ожидаемым от всего профессионально разработанного ПО. (Роджер Прессман, 2000)*

Здесь внимание акцентируется на трёх важных моментах:

1. Функциональные и нефункциональные требования к ПО являются фундаментом для измерения качества. Несоответствие требованиям свидетельствует о недостатке качества.

2. Стандарты определяют порядок разработки, проверки, управление изменениями, контроль версий и набор **метрик** для количественного измерения важных параметров ПО, показывающих уровень качества. Задаваемый стандартами процесс разработки позволяет избежать проектного хаоса — главного виновника плохого качества. Игнорирование стандартов всегда отражается на качестве продукта.

3. Существует множество подразумеваемых требований, которые часто не упоминаются (например, пожелание лёгкой сопровождаемости, удобство использования) и являются следствиями применения хороших практик программной инженерии, отражающих достигнутый мировой уровень технологий. Эти требования ориентированы на максимальное удовлетворение всех заинтересованных лиц. Если ПО их не обеспечивает, качество считают сомнительным.

Измерения помогают понять как процесс разработки продукта, так и сам продукт. Измерения процесса производятся в целях его улучшения, измерения продукта — для повышения его качества. В результате измерения определяется *мера* — количественная характеристика какого-либо свойства объекта. Путём непосредственных измерений могут определяться только опорные свойства объекта. Все остальные свойства оцениваются в результате вычисления тех или иных функций от значений опорных характеристик. Эти функции (формулы), дающие числовые значения называются **метриками**.

Из IEEE Standard Glossary of Software Engineering Terms:

> ***Метрика*** *— это мера степени обладания свойством, имеющая числовое значение.*

В программной инженерии понятия **мера** и **метрика** очень часто рассматривают как синонимы.

2. Определение и цели обеспечения качества ПО

Наиболее часто используют определение «обеспечения качества ПО» (SQA — Software Quality Assurance) из словаря IEEE (1990 г.).

> ***Обеспечение качества ПО*** *— это:*
>
> 1. *планируемый и систематичный шаблон всех действий, которые обеспечивают полную уверенность в том, что элемент или продукт соответствует определённым техническим требованиям;*
> 2. *набор видов деятельности, проектируемый для оценки процесса, по которому продукты разрабатываются или производятся (отличается от понятия «контроль качества»).*

Разберём данное определение.

- Определение IEEE относится лишь к процессу разработки ПО.
- Определение IEEE связывает сопровождение только со спецификациями на технические требования.
- В определении говорится о систематическом планировании и реализации.

  SQA основано на планировании и применении множества действий, которые интегрируются во все этапы процесса разработки ПО. Это придаёт заказчику уверенность в том, что программный продукт будет соответствовать всем техническим требованиям.

Чем отличается контроль качества от обеспечения качества? Эти термины представляют отдельные и различные понятия.

Контроль качества определён как «набор видов деятельности, проектируемых для оценки качества разрабатываемого или производимого продукта» (IEEE, 1990). Другими словами, главная цель этих видов деятельности — отбраковать продукты, не прошедшие оценку. Контроль качества применяется к разработке продукта до момента её завершения, то есть до отправки продукта заказчику. Типичной деятельностью контроля качества является *тестирование*.

Главная цель обеспечения качества — минимизировать стоимость гарантированного качества за счёт множества действий, выполняемых на различных этапах процесса разработки. Эти действия предотвращают ошибки, выявляют и корректируют их на ранних стадиях процесса разработки. В результате действия по обеспечению качества существенно уменьшают процент продуктов, непригодных для отправки, и в то же время сокращают стоимость гарантированного качества в большинстве случаев.

Следовательно, действия контроля качества являются лишь частью в общем списке действий по обеспечению качества.

Акцентируя внимание на планируемой и систематической реализации, определение IEEE ограничивает область применения SQA, исключая вопросы сопровождения, а также ограничения по времени и бюджету. Для улучшения результатов и достижения более полного удовлетворения заказчика следует учесть следующие расширения:

- Распространить область действия SQA на весь жизненный цикл программной системы. Такое решение позволяет интегрировать вопросы качества в функции сопровождения ПО.

- Не ограничивать SQA техническими вопросами функциональных требований, а распространить его на вопросы планирования и учёта бюджета. Известно, что вопросы планирования и бюджетных ограничений существенно влияют на реализацию функциональных технических требований. Очень часто ограничения по времени препятствуют внесению «опасных» изменений в план-график проекта, невзирая на функциональные требования. Нечто подобное может происходить и в условиях серьёзных бюджетных ограничений, налагающих запрет на дополнительные ресурсы.

Таким образом можно дать следующее расширенное определение:

> ***Обеспечение качества ПО*** — *систематический, планируемый набор действий, необходимых для формирования приемлемого уровня уверенности в том, что процесс разработки и сопровождения программной системы соответствует установленным функциональным техническим требованиям, а также организаторским требованиям соблюдения план-графика и бюджетных ограничений.*

Действия SQA покрывают функциональные, организационные и экономические аспекты разработки и сопровождения ПО.

Сгруппируем цели обеспечения качества по этапам жизненного цикла ПО:

- На этапе разработки ПО (ориентированы на процесс):

  1. Обеспечение приемлемого уровня уверенности в том, что ПО будет соответствовать функциональным и техническим требованиям.
  2. Обеспечение приемлемого уровня уверенности в том, что ПО будет создаваться в соответствии с план-графиком и бюджетными требованиями.
  3. Инициализация и управление действиями по совершенствованию и повышению эффективности разработки ПО, а также действий SQA. Это означает улучшение перспектив достижения функциональных и организационных (менеджерских) требований (при снижении стоимости разработки ПО и действий SQA).

- На этапе сопровождения ПО (ориентированы на продукт):

4. Обеспечение приемлемого уровня уверенности в том, что действия по сопровождению ПО будут соответствовать функциональным и техническим требованиям.
5. Обеспечение приемлемого уровня уверенности в том, что действия по сопровождению ПО будут соответствовать план-графику и бюджетным требованиям.
6. Инициализация и управление действиями по совершенствованию и повышению эффективности сопровождения ПО, а также действий SQA. Это означает улучшение перспектив достижения функциональных и организационных (менеджерских) требований при сокращении стоимости.

3. Факторы качества ПО

Качество ПО определяется множеством факторов, которые сильно зависят от предметной области и требований заказчика. Классической моделью факторов качества считают **модель Джима МакКолла**, предложенную в 1977 г. (Jim McCall). Дж. МакКолл сгруппировал 11 факторов в три категории:

- *Операционные факторы* (правильность, надёжность, эффективность, целостность, практичность).
- *Факторы изменяемости* (сопровождаемость, гибкость, тестируемость).
- *Факторы перемещаемости* (мобильность, многократность использования, способность к взаимодействию).

Во многих случаях нельзя прямо измерить приведённые факторы. Факторы можно выразить через метрики следующим образом:

$$F_j = \sum_{i=1}^{n} c_i \times m_i,$$

где F_j — j-й фактор качества ПО, c_i — весовой коэффициент, m_i — i-я метрика, оказывающая влияние на фактор качества ПО.

Многие из подобранных метрик измеряются только косвенным путём. Для их вычислений используют методы экспертных оценок, ранжирование и т. п.

В настоящее время для определения факторов качества применяют серию из четырёх международных стандартов:

- ISO/IEC 9126-1:2001 «Quality model»,
- ISO/IEC TR 9126-2:2003 «External metrics»,
- ISO/IEC TR 9126-3:2003 «Internal metrics»,
- ISO/IEC TR 9126-4:2004 «Quality in use metrics».

В этих стандартах определены три группы метрик качества: *внутренние метрики*, *внешние метрики* и *метрики «при использовании»*.

Внутренние метрики (internal metrics) фиксируют внутреннее качество, проявляющееся в ходе разработки ПО.

Внешние метрики (external metrics) определяют внешнее качество, заданное требованиями заказчика и демонстрируемое характеристиками конечной системы.

Метрики «при использовании» (quality in use metrics) измеряют качество в ходе нормальной эксплуатации программной системы конечными пользователями.

Внутренние и внешние метрики ориентированы на шесть характеристик качества, в свою очередь, каждая характеристика детализируется своим набором атрибутов.

Функциональность (functionality) — определяет способность продукта обеспечивать требуемые функции и операции.

Атрибут	**Описание.**
Пригодность (suitability)	Пригодность к обеспечению требуемой функциональности.
Точность (accuracy)	Обеспечение заданной точности результатов.
Способность к взаимодействию (interoperability)	Способность к взаимодействию с внешними системами.
Защищённость (security)	Защищённость внутренней информации от неавторизованного использования.
Согласованность (compliance)	Отсутствие противоречий с ограничениями стандарта.

Надёжность (reliability) — показывает возможности продукта для обеспечения достаточного уровня работоспособности в реальных условиях.

Атрибут	**Описание.**
Зрелость (maturity)	Способность продукта избегать ошибок при генерации исключений и ошибках данных.
Устойчивость к отказам (fault tolerance)	Способность продукта поддерживать определённый уровень производительности при генерации исключений и ошибках данных.
Восстанавливаемость (recoverability)	Способность продукта восстанавливать определённый уровень производительности при генерации исключений и ошибках данных.
Согласованность (compliance)	Отсутствие противоречий с ограничениями стандарта.

Практичность (usability) — означает понятность, лёгкость использования и изучения продукта.

Атрибут	**Описание.**
Понятность (understandability)	Обеспечение понимания пользователем порядка решения конкретной задачи с помощью продукта.
Обучаемость (learnability)	Возможность обучения пользователя работе с помощью продукта.
Удобство работы (operability)	Обеспечение пользователя возможностью решать все требуемые задачи.
Привлекательность (attractiveness)	Притягательность продукта с точки зрения пользователя.
Согласованность (compliance)	Отсутствие противоречий с ограничениями стандарта.

Эффективность (efficiency) — демонстрирует эффективность продукта при разумном задействовании ресурсов.

Атрибут	**Описание.**
Временная эффективность (time behavior)	Обеспечение быстрого взаимодействия и достаточной скорости решения задачи.
Ресурсоёмкость (resource behavior)	Задействование разумного объёма ресурсов при решении задачи.
Согласованность (compliance)	Отсутствие противоречий с ограничениями стандарта.

Сопровождаемость (maintainability) — измеряет возможности продукта быть изменчивым и обновляемым, а также удобным в сопровождении.

Атрибут	**Описание.**
Анализируемость (analysability)	Обеспечение возможности самоанализа при поиске причины ошибочного поведения.
Изменяемость (changeability)	Способность изменять структуру программы.
Стабильность (stability)	Сохранение стабильности даже при изменении структуры продукта.
Тестируемость (testability)	Поддержка валидации продукта с помощью тестирования.
Согласованность (compliance)	Отсутствие противоречий с ограничениями стандарта.

Переносимость (portability) — означает способность продукта быть системой, переносимой из одной аппаратно-программной среды в другую.

Атрибут	**Описание.**
Адаптируемость (adaptability)	Возможность адаптироваться к другой среде без привлечения дополнительной функциональности.
Простота установки (installability)	Способность инсталлироваться в конкретной среде.
Способность к сосуществованию (co-existence)	Способность продукта разделять среду с другой системой.
Взаимозаменяемость (replaceability)	Замещаемость компонентов при корректировке системы.
Согласованность (compliance)	Отсутствие противоречий с ограничениями стандарта.

Примеры метрик для оценки атрибутов качества

Полнота реализации функций — процент реализованных функций по отношению к количеству, заданному требованиями. Позволяет измерить *функциональную пригодность*.

Корректность реализации функций — правильность их реализации по отношению к требованиям. Позволяет измерить *функциональную пригодность*, *зрелость*.

Отношение числа проведённых тестов к общему их числу. Позволяет измерить *зрелость*.

Наглядность и полнота документации. Позволяет измерить *понятность*.

Метрики «при использовании» генерируются для четырёх характеристик качества, перечисленных в табл. 13.1.

Таблица 13.1.

Характеристики для метрик «при использовании»

Характеристика	**Описание.**
Эффективность (effectiveness)	Возможность для пользователя достичь цели с достаточной точностью и полнотой.
Продуктивность (productivity)	Возможность для пользователя достичь цели при использовании достаточного объёма ресурсов, обеспечивающих нужную производительность.
Безопасность (safety)	Обеспечение приемлемого уровня рисков (относящихся к людским ресурсам, данным, среде и т. д.).
Удовлетворённость запросов (satisfaction)	Возможность полного решения задачи с помощью продукта.

Схемы вычисления метрик качества для перечисленных характеристик подробно описаны во второй — четвертой частях стандарта. Стандарт определяет возможность

вычисления 80 базовых и более 250 производных метрик, предлагает обоснованный выбор мер и шкал для формируемых значений.

Порядок применения стандартов ISO/IEC 9126-1, -2, -3, -4 к обеспечению качества ПО задаёт другая линейка стандартов, — ISO/IEC 14598-1, -2, -3, -4, -5, -6, связывая измерения с конкретными этапами жизненного цикла программной системы.

В 2005 году начат процесс создания линейки международных стандартов второго поколения по качеству ПО — стандартов серии 25 000. Данная серия вбирает в себя весь положительный опыт линейки 9126. В частности, в ней уточнён и расширен до восьми набор характеристик качества для внутренних и внешних метрик, существенно обновлён набор характеристик (и введено 13 атрибутов) для метрик «при использовании». Процесс создания серии 25 000 завершён в 2014 г. Стандарт ISO/IEC 25000:2014 «Проектирование систем и разработка программного обеспечения. Требования к качеству систем и программного обеспечения и их оценка (SQuaRE). Руководство» опубликован 15 марта 2014 г.

Стандарт ГОСТ Р ИСО/МЭК 25021-2014 «Информационные технологии (ИТ). Системная и программная инженерия. Требования и оценка качества систем и программного обеспечения (SQuaRE). Элементы показателя качества», дата введения 1 июня 2015 г.

Стандартизованные характеристики качества оказывают существенную помощь заинтересованным лицам при формировании требований, они могут служить шаблоном требований. Ведь они определяют:

- Что должно делать ПО (например, поддерживать полный цикл покупки товара в интернет-магазине).
- Степень надёжности ПО (например, работать круглосуточно, 365 дней в году, с простоем не более 1 сут. в год).
- Степень удобства использования (например, обучать продавца всем функциям и возможностям за один рабочий день).
- Степень эффективности (например, поддерживать параллельную работу 100 покупателей).
- Удобство сопровождения (например, сохранять работоспособность при добавлении новых функций заказа, оплаты, доставки).
- Степень переносимости (например, работать в среде операционных систем GNU/Linux, Windows 7, Windows 8, Windows 10, формировать отчёты в форматах MS Word, Excel, ODF, HTML, XML).

При необходимости требования могут детализироваться с использованием полного набора атрибутов качества.

4. Модели качества процессов разработки

В современных условиях, условиях жёсткой конкуренции, очень важно гарантировать высокое качество процесса разработки ПО. Такую гарантию даёт сертификат

качества процесса, подтверждающий его соответствие принятым международным стандартам. Каждый такой стандарт фиксирует свою модель обеспечения качества. Наиболее авторитетны модели стандартов качества процесса разработки:

ISO 9001:2000 — ориентирована на процессы разработки из любых областей человеческой деятельности.

ISO/IEC 15504 — специализируется на процессах программной разработки и отличается более высоким уровнем детализации. Достаточно сказать, что объём этого стандарта превышает 500 страниц. Значительная часть идей ISO/IEC 15504 взята из модели CMM.

CMM (Capability Maturity Model) — модель зрелости процесса разработки ПО института программной инженерии при американском университете Карнеги-Меллон.

Базовым понятием модели CMM считается зрелость компании.

> ***Незрелой*** *называют компанию, где процесс конструирования ПО и принимаемые решения зависят только от таланта конкретных разработчиков.*

Как следствие, здесь высока вероятность превышения бюджета или срыва сроков окончания проекта.

Напротив, в **зрелой** компании работают ясные процедуры управления проектами и построения программных продуктов. По мере необходимости эти процедуры уточняются и развиваются. Оценки длительности и затрат разработки точны, основываются на накопленном опыте. Кроме того, в компании имеются и действуют корпоративные стандарты на процессы взаимодействия с заказчиком, процессы анализа, проектирования, программирования, тестирования и внедрения программных продуктов. Все это создаёт среду, обеспечивающую качественную разработку программного обеспечения.

Таким образом, модель CMM фиксирует критерии для оценки зрелости компании и предлагает рецепты для улучшения существующих в ней процессов. Иными словами, в ней не только сформулированы условия, необходимые для достижения минимальной организованности процесса, но и даются рекомендации по дальнейшему совершенствованию процессов.

Очень важно отметить, что модель CMM ориентирована на построение системы постоянного улучшения процессов. С точки зрения CMM, компания — это живая система. В CMM зафиксированы пять уровней зрелости и предусмотрен плавный, поэтапный подход к совершенствованию процессов — можно поэтапно получать подтверждения об улучшении процессов после каждого уровня зрелости.

1. *Начальный уровень* (*уровень 1*): «самоорганизующийся хаос», — процесс осуществляется случайным образом. Означает, что процесс в компании не формализован. Он не может строго планироваться и отслеживаться, его успех носит случайный характер. Результат работы целиком и полностью зависит от личных качеств отдельных сотрудников, при увольнении таких сотрудников проект останавливается.

2. *Повторяемый уровень* (*уровень 2*): процесс планируется и отслеживается.

   Для перехода на повторяемый уровень (уровень 2) необходимо внедрить формальные процедуры для выполнения основных элементов процесса конструирования. Результаты выполнения процесса соответствуют заданным требованиям и стандартам. Основное отличие от *уровня 1* состоит в том, что выполнение процесса планируется и контролируется. Применяемые средства планирования и управления дают возможность повторения ранее достигнутых успехов.

3. *Определённый уровень* (*уровень 3*): процесс полностью определён и организован на основе единого стандарта компании.

   Этот уровень требует, чтобы все элементы процесса были определены, стандартизованы и задокументированы. Основное отличие от уровня 2 заключается в том, что элементы процесса уровня 3 планируются и управляются на основе единого стандарта компании. Качество разрабатываемого ПО уже не зависит от способностей отдельных личностей.

4. *Управляемый уровень* (*уровень 4*): количественное управление процессом и его качеством. С переходом на управляемый уровень в компании принимаются количественные показатели качества как программных продуктов, так и процесса. Это обеспечивает более точное планирование проекта и контроль качества его результатов. Основное отличие от уровня 3 состоит в более объективной, количественной оценке продукта и процесса.

5. *Оптимизирующий уровень* (*уровень 5*): планомерное улучшение и повышение качества процесса.

   Этот уровень подразумевает, что главной задачей компании становится постоянное улучшение и повышение эффективности существующих процессов, ввод новых технологий. Основное отличие от уровня 4 заключается в том, что технология создания и сопровождения программных продуктов планомерно и последовательно совершенствуется.

Каждый уровень CMM характеризуется **областью ключевых процессов** (**ОКП**), причём считается, что каждый последующий уровень включает в себя все характеристики предыдущих уровней. Иначе говоря, для 3-го уровня зрелости рассматриваются ОКП 3-го уровня, ОКП 2-го уровня и ОКП 1-го уровня.

> ***Область ключевых процессов*** *(ОКП) образуют процессы, которые при совместном выполнении приводят к достижению определённого набора целей.*

Пример .46 (ОКП 5-го уровня). ОКП 5-го уровня образуют процессы:

- предотвращения дефектов;
- управления изменениями технологии;
- управления изменениями процесса.

Если все цели ОКП достигнуты, компании присваивается *сертификат* данного уровня *зрелости*. Если хотя бы одна цель не достигнута, то компания не может соответствовать данному *уровню CMM*.

Проверяет компанию и выносит решение о сертификации CMM соответствующего уровня сертифицированный оценщик (Lead Appraiser). Сертификацию оценщиков проводит Software Engineering Institute (SEI) в Университете Карнеги-Меллона (Carnegie Mellon University).

Для повышения квалификации, обучения членов команды новым технологиям, можно использовать доступные сервисы.

Пример .47 (учебные материалы).

- Запись и публикация обучающих роликов: Screencast-O-Matic (до 15 мин. бесплатно); Zoom, Telemost, Google Meet, ... В Screencast-O-Matic запись экрана работает на Windows, Mac, iPhone, iPad, Android и Chromebook.
- Публикация слайдов (PPT): SlideShare.
- Публикация слайдов (PDF): Speaker Deck.

5. Деятельность по обеспечению качества ПО

Обеспечение качества ПО (SQA) — защитная деятельность, применяемая на всём протяжении жизненного цикла системы, включающая следующие виды действий:

- Применение технических методов и средств анализа, проектирования, кодирования и сопровождения.

  Технические методы и средства помогают аналитику получить высококачественную спецификацию, а проектировщику (программисту) — провести высококачественное проектирование (программирование).

- Проведение технических проверок и аудитов на каждом шаге разработки.

  После того как спецификация и проектное решение (программный код) созданы, осуществляется оценка их качества. Для этого используют самые разные действия: технические проверки и аудиты.

  Техническая проверка и аудит организуются как встреча персонала с единственной целью — обнаружить проблемы качества.

- Верификация и валидация продукта.

  Основными средствами верификации и валидации являются инспектирование и тестирование. Многие разработчики используют тестирование ПО как спасательную сеть для обеспечения качества. Иными словами, они полагают, что полное тестирование обнаружит большинство ошибок, вследствие чего ослабляется необходимость в других действиях по обеспечению качества. К сожалению, тестирование эффективно не для всех классов ошибок.

  Верификация и валидация — это не единичный акт, а процесс, проверяющий соответствие ПО своей спецификации и требованиям заказчиков. Верификация

и валидация охватывает полный жизненный цикл ПО, начинается на этапе анализа требований и завершается проверкой готовой программной системы.

- Внедрение в жизнь стандартов.

  Диапазон стандартов и процедурных норм, применяемых к процессу разработки ПО разными компаниями, существенно различен. Во многих случаях стандарты задаются заказчиком или контрактом. В других случаях они выбираются самими разработчиками. Хорошо организованный процесс SQA создаёт в компании атмосферу культивирования качества, стимулирующий команду к качественной работе и поиску путей повышения качества.

  Если стандарты приняты, то действия по обеспечению качества должны гарантировать их соблюдение. Оценка соответствия стандартам может выполняться разработчиками как часть технической проверки, или в ситуациях, где требуется независимая верификация, специальная группа качества может проводить свою собственную ревизию.

- Контроль изменений.

  Наиболее вероятным источником угрозы для качества ПО являются изменения. Каждое изменение в ПО рассматривается как потенциальный источник ошибок или как предпосылка для распространения ошибок. Процесс контроля изменений (он входит в процесс управления конфигурацией) прямо способствует качеству ПО:

  — формализацией запросов на изменения;

  — оценкой сущности изменения;

  — контролем влияния изменения.

  Контроль изменений осуществляется как на стадии разработки, так и на стадии сопровождения.

- Проведение измерений показателей качества и их оценка — действие, которое интегрируется в любую инженерную дисциплину. Важной целью обеспечения качества считается отслеживание качества и оценка влияния методологических и процедурных изменений на улучшение качества ПО. Для достижения этой цели должны собирать результаты измерений, проводимых по метрикам качества ПО.

- Сохранение записей и формирование отчётности предусматривает процедуры для сбора и распространения информации. Результаты проверок, аудитов, инспектирований, контроля изменений, тестирования и другой деятельности группы контроля качества должны стать частью архива записей по проекту и распространяться среди разработчиков по мере необходимости. Например, результаты каждой технической проверки по проектированию записываются и могут помещаться в брошюру, которая содержит всю техническую информацию о системе и информацию по качеству.

5.1. Технические проверки и аудиты

Проверки ПО позволяют выявить дефекты таких видов деятельности, как анализ, проектирование, кодирование. Существует много типов проверок, которые могут проводиться как часть процесса SQA (неформальная встреча у кофейного автомата, формальное представление программного проекта аудитории из заказчиков, руководства, технического персонала и т. д.).

Назначение технической проверки:

1. Обнаружение ошибок и противоречий в функциях, логике или реализации для любой формы представления ПО, включая документацию.
2. Проверка соответствия ПО требованиям.
3. Обеспечение представления ПО согласно определённым стандартам.
4. Формирование более управляемого проекта.

Проверяется основная документация (*спецификация*, *программный код*, *диаграммамы* по основным процессам разработки) и такие документы, как *план-график разработки*, *планы тестирования*, *управление конфигурацией*, *стандарты* и *документация* для пользователя.

В группу проверки включаются такие специалисты, которые могут принести максимальную пользу. Например, при проверке артефактов проектирования следует привлечь проектировщиков из схожих проектов.

Группа проверяющих состоит из 3–4 человек, один из них назначается старшим. Кроме этого, по частным вопросам могут привлекаться дополнительные сотрудники. Проверяемые документы раздаются до начала проверки, для предварительного ознакомления. Сама проверка длится не более двух часов. По окончании проверки все замечания заносятся в отчёт. Отчёт подписывается всеми проверяющими.

Аудит отличается от технической проверки тем, что не ограничивается по времени, а в состав группы входят лица из внешних организаций. Аудит, как правило, разработку не рассматривает с точки зрения контрактных обязательств, привязывая выполненную работу к план-графику проекта.

Очевидная польза *технических проверок* — раннее обнаружение программных дефектов. Обнаруженные дефекты исправляются перед следующим шагом разработки.

Статистика показывает, что 50–60% всех ошибок разработки ПО приходится на действия по проектированию. Технические проверки обеспечивают обнаружение до 75% недостатков проектирования. Таким образом, обнаруживая и устраняя большой процент ошибок, процесс проверки существенно сокращает стоимость последующих шагов разработки и сопровождения.

Анализ большого числа программных проектов показал что устранение ошибки, обнаруженной перед тестированием, обходится в 6,5; в течение тестирования — в 15; после завершения проекта — в 60–100 раз дороже, чем на этапе проектирования.

5.2. Инспектирование

В отличие от тестирования инспектирование является статическим способом поиска ошибок. **Инспектирование** заключается в просмотре и проверке артефактов проекта на наличие ошибок, его цель — обнаружение дефектов, а не исследование общих проблем проекта. Такими артефактами являются требования, результаты проектирования, программный код и т. д. *Дефектами* являются либо ошибки во фрагменте системы, либо несоответствие фрагмента принятым стандартам.

Идея инспектирования принадлежит сотруднику фирмы IBM [Майкл Фаган]Майклу Фагану ([Майкл Фаган]Michael Fagan). В настоящее время данный метод верификации ПО получил широкое применение. Подразумевается, что инспектирование осуществляется коллегами автора, которые анализируют исходный программный код и помогают автору найти в нём дефекты.

Процесс инспектирования формализован, выполняется группой разработчиков, они должны выполнять определённые роли:

Автор — несёт ответственность за свою работу (артефакт), после завершения инспектирования самостоятельно исправляет все обнаруженные дефекты.

Рецензент — комментирует (рецензирует) программный код, отвечает за правильное проведение инспектирования.

Корректор — ищет в программе (или документе) ошибки, противоречия и упущения.

Регистратор — отвечает за учёт описаний и классификацию обнаруженных дефектов, записывает результаты собрания группы инспектирования.

Содержание этапов процесса инспектирования:

1. *Планирование. Рецензент* составляет план инспектирования исходя из диалога с *автором* и другими членами группы инспектирования.

2. *Предварительный просмотр.* На собрании *автор* знакомит группу с назначением и основными особенностями своего артефакта (объекта инспектирования).

3. *Индивидуальная подготовка.* Участникам необходимо тщательно подготовиться к инспектированию. Инспектирующие должны понимать программу на том же уровне детализации, что и автор. Каждый член инспекционной группы самостоятельно вникает в программу и её спецификацию, фиксируя обнаруженные недостатки. Этот этап требует существенных усилий.

4. *Собрание инспекционной группы.* Как только каждый из участников группы готов, проводится собрание, в ходе которого все участники выполняют свои роли. На собрании активно используются *проверочные списки*, содержащие вопросы, на которые следует обратить особое внимание. Собрание длится не более двух часов и выявляет дефекты, аномалии и несоответствия стандартам. Способы исправления обнаруженных дефектов не рассматриваются.

5. *Исправление ошибок.* После инспектирования автор изменяет артефакт, исправляя обнаруженные ошибки.

6. *Доработка.* Рецензент принимает решение о необходимости повторного инспектирования. Если оно не требуется, все обнаруженные дефекты оформляются документально, а документ подписывается. Группа собирается в последний раз, подводит итоги, обновляет проверочные списки.

Вопросы для выявления ошибок

Далее приводится список примерных вопросов.

- Все ли переменные получили значения до начала их использования?
- Все ли константы именованы?
- Корректно ли заданы границы массивов?
- Выполняются ли условия для каждого условного оператора?
- Все ли циклы завершаются?
- Правильно ли расставлены скобки в составных операторах?
- Все ли варианты выбора реализуются в операторах?
- Используются ли в программе входные переменные?
- Все ли выходные переменные получают значения перед выводом на печать?
- Могут ли какие-нибудь входные данные привести к нарушению данных системы?
- Все ли вызовы процедур (подпрограмм и функций) содержат правильное количество параметров?
- Согласованы ли типы формальных и фактических параметров?
- В правильном ли порядке расположены параметры?
- Если модули обращаются к разделяемым ресурсам, обеспечены ли взаимное исключение и условная синхронизация?
- Если связанная структура данных изменяется, правильно ли переопределяются её связи?
- Если используется динамическая память, правильно ли она распределяется?
- Происходит ли сбор мусора и решена ли проблема повисших указателей?
- Все ли возможные ошибки рассмотрены в условиях, определяющих исключительные ситуации?

В программной индустрии ведут учёт времени, потраченного на инспектирование, и объёма проверенной работы. *Статистика показывает, что на инспектирование уходит порядка 10–15% бюджета разработки. Но всё же инспектирование вполне окупает себя.*

5.3. Верификация и валидация

Верификация и **валидация** (*V&V* — verification and validation) являются составной частью процесса контроля качества. Определения этих терминов из стандарта IEEE Std 1012-2004 «IEEE Standard for Software Verification and Validation»:

> ***Верификация*** *— процесс оценки системы или компонента для выявления того, удовлетворяют ли продукты данного этапа разработки условиям, изложенным в начале этапа.*

> ***Верификация*** *— процесс подтверждения того, что ПО и ассоциированные с ним продукты соответствуют требованиям (например, по правильности, законченности, совместимости, точности) ко всем видам деятельности жизненного цикла; соответствуют стандартам, практикам и соглашениям для всех процессов жизненного цикла; полностью и успешно завершена каждая деятельность жизненного цикла и соответствует всем критериям для инициализации последующих видов деятельности жизненного цикла.*

Верификация отвечает на вопрос: правильно ли строится, создаётся ПО?

> ***Валидация*** *— оценка системы или компонента во время или в конце процесса разработки для определения — удовлетворяет ли она (он) указанным требованиям.*

> ***Валидация*** *— процесс подтверждения того, что ПО и ассоциированные с ним продукты удовлетворяют системным требованиям к программному продукту, определённым для завершения каждого вида деятельности жизненного цикла, правильно решают проблемы (например, учитывают закономерности предметной области, бизнес-правила, используют правильные системные допущения), а также соответствуют назначению и нуждам пользователей.*

Валидация проверяет: правильно ли работает система, отвечает ли построенная система пожеланиям и нуждам заказчика.

Верификация проверяет соответствие ПО функциональным и нефункциональным требованиям. **Валидация** является более общим процессом, в ходе которого надо убедиться, что программная система соответствует ожиданиям заказчика. **Валидация** проводится после **верификации**, для того чтобы определить, насколько система соответствует не только спецификации, но и ожиданиям заказчика.

Конечно, в требованиях тоже встречаются ошибки и упущения, из-за которых конечный продукт может не совсем соответствовать ожиданиям заказчика. Поэтому требования тоже рекомендуется подвергать валидации.

В ходе **верификации** и **валидации** используют два основных действия:

Инспектирование ПО. Осуществляется на всех этапах разработки программной системы. Параллельно с инспектированием может выполняться автоматический анализ программного кода и других артефактов. Инспектирование и автоматический анализ являются статическими методами верификации и валидации, поскольку им не требуется работающая система.

Тестирование ПО. Требует запуска фрагментов исполняемого кода. Тестирование считается динамическим методом верификации и валидации, так как применяется к работающей системе.

Инспектирование можно выполнять на всех этапах разработки системы, а **тестирование** — лишь в тех случаях, когда создан исполняемый код.

Инспектирование и тестирование не следует считать конкурирующими инструментами верификации и валидации. У каждого из них есть свои достоинства и недостатки, поэтому целесообразно их совместное применение в верификация и валидации.

К методам инспектирования относят *инспектирование программ* и *автоматический анализ исходного кода*, проверяющий его синтаксическую и семантическую корректность. Но статичность ограничивает область действия инспектирования проверкой спецификаций. К сожалению, инспектировать функционирование продукта просто невозможно. Нельзя применить инспектирование и к проверке многих нефункциональных характеристик (например, производительности, надёжности). Поэтому для этих целей используют тестирование.

Базовым методом верификации и валидации является **тестирование**.

Увы, создание идеального ПО может быть лишь мечтой. Верификация и валидация выявляют лишь степень приближения к идеалу заказчика, степень соответствия программной системы запланированным целям. Степень приближения к идеальному ПО определяется:

- назначением системы (для критических систем степень соответствия максимальна, например для системы управления ядерным реактором);
- ожиданиями пользователей (хотят заплатить мало, но терпеть недостатки, или платят много и недостатков не признают);
- условиями на рынке программных продуктов (есть конкуренция для необходимого семейства продуктов, или она отсутствует).

6. План обеспечения качества ПО

План обеспечения качества ПО (Software Quality Assurance, SQA) отражает широкую панораму действий, ориентированных на формирование качества. Разрабатываемый группой SQA, план служит шаблоном действий, которые должны быть внедрены в каждый программный проект.

В стандарте IEEE Std 730-2002 предлагается следующая структура плана SQA:

1. Цель.

2. Упоминаемые документы.

3. Руководство.

3.1. Организация.

3.2. Задачи.

3.3. Обязанности.

3.4. Оцениваемые ресурсы обеспечения качества.

4. Документация.

4.1. Цель.

4.2. Минимальные требования к документации.

4.3. Прочее.

5. Стандарты, практики, соглашения и метрики.

5.1. Цель.

5.2. Содержание.

6. Проверки.

6.1. Цель.

6.2. Минимальные требования:

- проверка спецификаций ПО;
- проверка архитектурного проектирования;
- проверка детального проектирования;
- проверка плана верификации и валидации;
- аудит функциональности;
- аудит физических компонентов;
- внутренние аудиты процесса;
- проверка организации;
- проверка плана управления конфигурацией;
- проверка пост-реализации.

6.3. Прочие проверки и аудиты.

7. Тестирование. Может ссылаться на документацию по тестированию ПО.

8. Отчёты о проблемах и корректирующие действия.

9. Инструменты, технологии и методологии. Может ссылаться на план управления программным проектом.

10. Контроль носителей информации.

11. Контроль поставщиков.

12. Сбор, сопровождение и хранение протоколов.

13. Обучение.

14. Управление рисками. Может ссылаться на план управления программным проектом.

15. Словарь (тезаурус).

16. Процедура изменения плана и версии.

При работе над планом SQA важно придерживаться предельно лаконичного стиля изложения. Если документ окажется слишком большим, то вряд ли его дочитают до конца, что сведёт на нет саму идею плана обеспечения качества.

Глава 14

Тестирование программного обеспечения

1. Основные понятия

> ***Тестирование*** *— процесс выполнения программы с целью обнаружения ошибок.*

Шаги процесса задаются **тестами**.
Каждый **тест** определяет:

- свой набор исходных данных и условий для запуска программы;
- набор ожидаемых результатов работы программы.

Другое название **теста — тестовый вариант**.

Полную проверку программы гарантирует **исчерпывающее тестирование**. Оно требует проверить все наборы исходных данных, все варианты их обработки и включает большое количество тестовых вариантов. Увы, но **исчерпывающее тестирование** во многих случаях остаётся только мечтой — срабатывают ресурсные ограничения (прежде всего, ограничения по времени).

Хорошим считают **тестовый вариант** с высокой вероятностью обнаружения ещё не раскрытой ошибки. *Успешным* называют **тест**, который обнаруживает до сих пор не раскрытую ошибку.

Целью проектирования **тестовых вариантов** является систематическое обнаружение различных классов ошибок при минимальных затратах времени и стоимости.

Тестирование обеспечивает:

- обнаружение ошибок;
- демонстрацию соответствия функций программы её назначению;
- демонстрацию реализации требований к ПО;
- оценку надёжности как показателя качества программы.

Тестирование не может показать отсутствие дефектов, оно может показать только их присутствие. Важно помнить это утверждение при проведении тестирования.

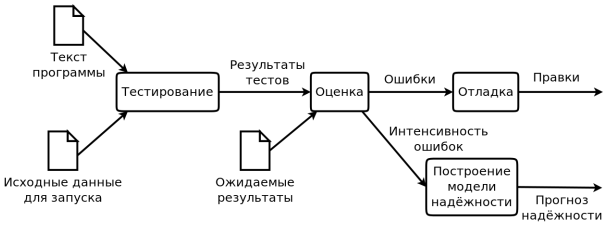


Рис. 14.1. Информационные потоки процесса тестирования

Рассмотрим информационные потоки процесса тестирования (рис. 14.1).

На вход процесса тестирования поступает три потока от следующих артефактов разработки:

- текст программы;
- исходные данные для запуска программы;
- ожидаемые результаты.

Выполняются тесты, все полученные результаты оцениваются. Это значит, что реальные результаты тестов сравниваются с ожидаемыми результатами. Когда обнаруживается несовпадение, фиксируется ошибка — начинается отладка. Процесс отладки непредсказуем по времени. На поиск места дефекта и исправление может потребоваться час, день, месяц. Неопределённость в отладке приводит к большим трудностям в планировании действий.

После сбора и оценивания результатов тестирования начинается оценка качества и надёжности ПО.

Если регулярно встречаются серьезные ошибки, требующие проектных изменений, то качество и надёжность ПО сомнительны, констатируется необходимость усиления тестирования. С другой стороны, если функции ПО реализованы правильно, а обнаруженные ошибки легко исправляются, может быть сделан один из двух выводов:

- качество и надёжность ПО удовлетворительны;
- тесты не способны обнаруживать серьезные ошибки.

В конечном счёте, если тесты не обнаруживают ошибок, появляется сомнение в том, что тестовые варианты достаточно продуманы и что в ПО нет скрытых ошибок. Такие ошибки будут, в конечном итоге, обнаруживаться пользователями и корректироваться разработчиком на этапе сопровождения (когда стоимость исправления возрастает в 60–100 раз по сравнению с этапом разработки).

Для формальной оценки результатов тестирования используют **модели надёжности ПО**, выполняющие *прогноз надёжности* по данным об интенсивности ошибок.

Существуют 2 принципа тестирования программы:

- **функциональное тестирование** (тестирование «чёрного ящика»);
- **структурное тестирование** (тестирование «белого ящика»).

1.1. Тестирование «чёрного ящика»

Известны: функции программы.
Исследуется: работа каждой функции на всей области определения.
Как показано на рис. 14.2, основное место приложения тестов «чёрного ящика» — интерфейс ПО.

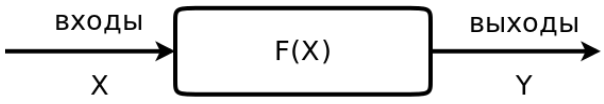


Рис. 14.2. Тестирование «чёрного ящика»

Эти тесты проверяют:

- как выполняются функции программ;
- как принимаются исходные данные;
- как вырабатываются результирующие данные;
- как сохраняется целостность внешней информации.

При тестировании «чёрного ящика» рассматриваются системные характеристики программ, игнорируется их внутренняя логическая структура. Исчерпывающее тестирование, как правило, невозможно. Например, если в программе 10 входных величин и каждая принимает по 10 значений, то потребуется 10^{10} тестовых вариантов. Следует отметить, что тестирование «чёрного ящика» не реагирует на многие особенности программных ошибок.

1.2. Тестирование «белого ящика»

Известна: внутренняя структура программы.
Исследуются: элементы программы и связи между ними (рис. 14.3).

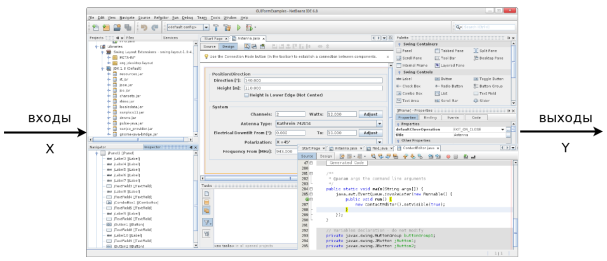


Рис. 14.3. Тестирование «белого ящика»

Объектом тестирования здесь является не внешнее, а внутреннее поведение программы. Проверяется корректность построения всех элементов программы и правильность их взаимодействия друг с другом. Обычно анализируются управляющие связи элементов, реже — информационные связи. В таком тестировании вместо «белого ящика» иногда применяют термин «стеклянный ящик», «прозрачный ящик». Такое тестирование характеризуется степенью **покрытия** логики (кода) программы. Исчерпывающее тестирование также затруднительно.

2. Функциональное тестирование ПО

Функциональное тестирование программного обеспечения, основанное на принципе «чёрного ящика», может применяться и на уровне программных модулей и на уровне программной системы.

2.1. Особенности тестирования «чёрного ящика»

Функциональное тестирование, или тестирование «чёрного ящика», позволяет получить комбинации входных данных, обеспечивающих полную проверку всех *функциональных требований* к программе. Программный продукт здесь рассматривается как «чёрный ящик», чьё поведение можно определить только исследованием его входов и соответствующих выходов.

При таком подходе желательно иметь:

- набор A, образуемый такими *входными данными*, которые приводят к *аномалиям* поведения программы;
- набор D, образуемый такими *выходными данными*, которые демонстрируют *дефекты* программы.

Любой способ тестирования «чёрного ящика» (рис. 14.4) должен:

- выявить такие входные данные, которые с высокой вероятностью принадлежат набору A;
- сформулировать такие ожидаемые результаты, которые с высокой вероятностью являются элементами набора D.

Во многих случаях определение таких тестовых вариантов основывается на предыдущем опыте инженеров тестирования. Они используют своё знание и понимание области определения для идентификации тестовых вариантов, которые эффективно обнаруживают дефекты. Тем не менее систематический подход к выявлению тестовых данных может использоваться как полезное дополнение к эвристическому знанию. Особенно для начинающих тестировщиков.

Принцип «чёрного ящика» не альтернативен принципу «белого ящика», это дополняющий подход, который обнаруживает другой класс ошибок. Тестирование «чёрного ящика» обеспечивает поиск следующих категорий ошибок:

1. некорректных или отсутствующих функций;

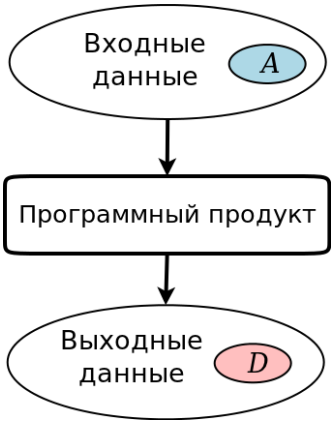


Рис. 14.4. Процесс тестирования «чёрного ящика»

2. ошибок интерфейса;

3. ошибок во внешних структурах данных или в доступе к внешней базе данных;

4. ошибок характеристик (необходимая ёмкость памяти и т. д.);

5. ошибок инициализации и завершения.

Подобные категории ошибок способами «белого ящика» не выявляются.

В отличие от тестирования «белого ящика», которое выполняется на ранней стадии разработки (кодирования), тестирование «чёрного ящика» применяют на поздних стадиях.

При тестировании «чёрного ящика» пренебрегают управляющей структурой программы. Здесь внимание концентрируется на *информационной области определения программной системы.*

Техника «чёрного ящика» ориентирована на решение следующих задач:

- *сокращение необходимого количества тестовых вариантов* благодаря проверке не статических, а динамических аспектов системы;
- *выявление классов ошибок*, а не отдельных ошибок.

2.2. Разбиение по эквивалентности

Разбиение по эквивалентности — самый популярный способ тестирования «чёрного ящика». В этом способе входная область данных программы делится на **классы эквивалентности**.

Класс эквивалентности *— набор данных с общими свойствами.*

Для каждого **класса эквивалентности** разрабатывается один тестовый вариант. Обрабатывая разные элементы одного **класса**, программа должна вести себя

одинаково. Иначе говоря, при обработке любого набора из класса эквивалентности в программе задействуется один и тот же набор операторов (и связей между ними).

На рис. 14.5 каждый класс эквивалентности показан эллипсом. Здесь выделены входные классы эквивалентности допустимых и недопустимых исходных данных, а также классы результатов.

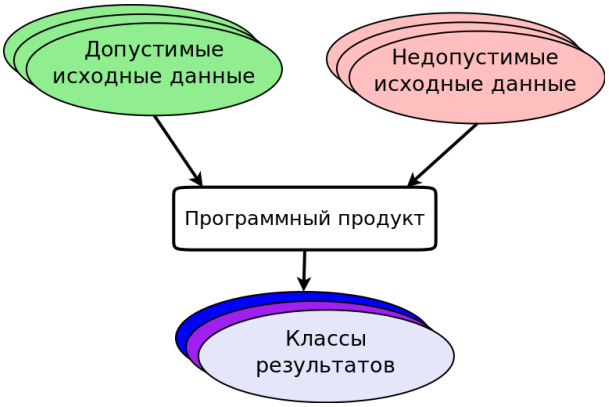


Рис. 14.5. Разбиение по эквивалентности

Классы эквивалентности могут быть определены по спецификации на программу.

Пример .48. Если спецификация задаёт в качестве допустимых входных величин 5-разрядные целые числа в диапазоне 15 000 . . . 70 000, то класс эквивалентности допустимых исходных данных (ИД) включает величины от 15 000 до 70 000, а два класса эквивалентности недопустимых ИД составляют:

- числа меньшие, чем 15 000;
- числа большие, чем 70 000.

Класс эквивалентности включает множество значений данных, допустимых или недопустимых по условиям ввода. Условие ввода может задавать:

1. определённое значение;
2. диапазон значений;
3. множество конкретных величин;
4. булево значение.

Правила формирования классов эквивалентности

1. Если условие ввода задаёт диапазон $[m; n]$, то определяются один допустимый и два недопустимых класса эквивалентности: `V_Class` $= \{m, \ldots, n\}$ — допустимый класс эквивалентности; `Inv_Class1` $= \{x|$для любого x: $x < m\}$ — первый недопустимый класс эквивалентности; `Inv_Class2` $= \{y|$для любого y: $y > n\}$ — второй недопустимый класс эквивалентности.

2. Если условие ввода задаёт конкретное значение a, то определяется один допустимый и два недопустимых класса эквивалентности: `V_Class` $= \{a\}$; `Inv_Class1` $= \{x|$для любого x: $x < a\}$; `Inv_Class2` $= \{y|$для любого y: $y > a\}$.

3. Если условие ввода задаёт множество значений $\{a, b, c\}$, то определяются один допустимый и один недопустимый класс эквивалентности: `V_Class` $= \{a, b, c\}$; `Inv_Class` $= \{x|$для любого x: $(x \neq a)\&(x \neq b)\&(x \neq c)\}$.

4. Если условие ввода задаёт булево значение, например `true`, то определяются один допустимый и один недопустимый класс эквивалентности: `V_Class`=`{true}`; `Inv_Class`=`{false}`.

Классы плохих данных

Программу можно тестировать на предмет нескольких других классов *плохих данных*. Типичными примерами плохих данных можно считать:

- недостаток данных (или их отсутствие);
- избыток данных;
- неверные значения данных (некорректные данные);
- неверный размер данных;
- неинициализированные данные.

Некоторые из этих случаев могут быть уже покрыты имеющимися тестами.
Для «неверного размера данных» тесты придумать трудно.

Пример .49 (тестовые варианты на классах плохих данных).

- *Отрицательная зарплата* — неверные значения данных.
- *Отрицательное число сотрудников* — неверные значения данных.
- *Дата рождения сотрудника 01.01.1801* — неверные значения данных.
- *Размер массива для сотрудников организации 10 000 000 000* — избыток данных.
- *Не задано количество рабочих дней* — неинициализированные данные.
- *Количество дней в командировке 0* — неверный размер данных.
- *Размер «суточных» для командировки не задан* — недостаток данных.

После построения классов эквивалентности разрабатываются тестовые варианты. **Тестовый вариант** выбирается так, чтобы проверить сразу наибольшее количество свойств класса эквивалентности.

Используйте тесты, позволяющие легко проверить результаты вручную.

Пример .50 (удобные числа). Допустим, вы пишете тест для проверки расчёта зарплаты; вам нужно ввести оклад, и один из способов сделать это — ввести числа, которые попадаются под руку. Например, 123456789 или, лучше 1234567,89. Это довольно большая зарплата — более ста миллионов. Теперь предположим, что этот тест успешен, т. е. указывает на ошибку. Как узнать,что вы обнаружили ошибку? Вы можете вычислить правильный результат вручную и сравнить его с результатом, полученным на самом деле. Однако если в вычислениях фигурируют такие неприятные числа, как 1 234 567,89, вероятность допустить ошибку в вычислениях не менее высока, чем вероятность обнаружения ошибки в программе.

С другой стороны, удобные круглые числа вроде 20 000 делают вычисления очень простыми. Нули легко набирать, а умножение на 2 большинство программистов способны выполнять в уме.

Некоторые считают, что неудобные числа чаще приводят к обнаружению ошибок, но это не так. При использовании любого числа из одного и того же **класса эквивалентности** вероятность нахождения ошибки одинакова!

2.3. Анализ граничных значений

Как правило, бóльшая часть ошибок происходит на границах области ввода, а не в центре.

> ***Анализ граничных значений*** *заключается в получении тестовых вариантов с данными, лежащими на границах* ***классов эквивалентности.***

Данный способ тестирования дополняет способ **разбиения по эквивалентности**.

Основные отличия **анализа граничных значений** от **разбиения по эквивалентности**:

1. тестовые варианты создаются для проверки только рёбер **классов эквивалентности**;
2. при создании **тестовых вариантов** учитывают не только условия ввода, но и область вывода.

Правила анализа граничных значений

1. Если условие ввода задаёт диапазон $[m; n]$, то тестовые варианты должны быть построены:
   - для значений m и n;
   - для значений чуть левее m и чуть правее n на числовой оси.

   Пример .51. Если задан входной диапазон $(-1{,}0; +1{,}0)$, то создаются тесты для значений $-1{,}0$, $+1{,}0$, $-1{,}001$, $+1{,}001$.
2. Если условие ввода задаёт дискретное множество значений, то создаются тестовые варианты:

- для проверки минимального и максимального из входных значений;
- для значений чуть меньше минимума и чуть больше максимума.

Пример .52. Если входной файл может содержать от 1 до 255 записей, то создаются тесты для 0, 1, 255, 256 записей.

3. Правила 1 и 2 применяются к условиям области вывода.

   Пример .53. В программе требуется выводить таблицу значений. Количество строк и столбцов в таблице меняется. Создаётся тестовый вариант для минимального вывода (по объёму таблицы), а также тестовый вариант для максимального вывода (по объёму таблицы).

4. Если внутренние структуры данных программы имеют предписанные границы, то разрабатываются тестовые варианты, проверяющие эти структуры на их границах.

5. Если входные или выходные данные программы являются упорядоченными множествами (например, последовательным файлом, линейным списком, таблицей), то надо тестировать обработку первого и последнего элементов этих множеств.

Большинство разработчиков используют этот способ интуитивно. При использовании описанных правил тестирование границ будет более полным, в связи с чем возрастёт вероятность обнаружения ошибок.

2.4. Функциональные диаграммы причинно-следственных связей

Функциональные диаграммы причинно-следственных связей *(**cause-effect graphs**) — способ проектирования тестовых вариантов, который обеспечивает формальную запись логических условий и соответствующих действий.*

Не путать с диаграммой причинно-следственных связей Исикавы (диаграммой «рыбьей кости»). Используется автоматный подход к решению задачи.

Шаги способа:

1. Для каждого модуля перечисляются причины (условия ввода или классы эквивалентности условий ввода) и следствия (действия или условия вывода). Каждой причине и следствию присваивается свой *идентификатор*.

2. Строится **граф причинно-следственных связей**.

3. Граф преобразуется в **таблицу решений**.

4. Столбцы **таблицы решений** преобразуются в **тестовые варианты**.

Функциональная диаграмма причинно-следственных связей представляет собой связанный граф, узлы которого символизируют булевы состояния элементов программного продукта (программы). При этом используются следующие соглашения:

- *причины* обозначаются символами c_i (cause), а *следствия* — символами e_i (effect);
- каждый узел графа может находиться в состоянии 0 или 1 (0 — состояние отсутствует, 1 — состояние присутствует).

Рассмотрим базовые символы для записи графов причин и следствий (cause-effect graphs).

Функция тождество (рис. 14.6) устанавливает, что если значение c_1 есть 1, то и значение e_1 есть 1; в противном случае значение e_1 есть 0.

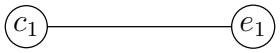


Рис. 14.6. Функция тождество

Функция не (рис. 14.7) устанавливает, что если значение c_1 есть 1, то значение e_1 есть 0; в противном случае значение e_1 есть 1.

Рис. 14.7. Функция не

Функция или (рис. 14.8) устанавливает, что если c_1 или c_2 есть 1, то e_1 есть 1, в противном случае e_1 есть 0.

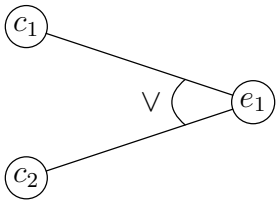


Рис. 14.8. Функция или

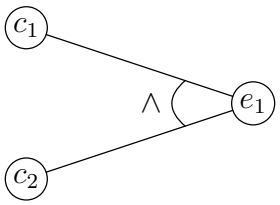


Рис. 14.9. Функция и

Таблица 14.1.

Таблица истинности для ограничения E

Ограничение E	a	b
true	0	0
	0	1
	1	0
false	1	1

Функция и (рис. 14.9) устанавливает, что если и c_1 и c_2 есть 1, то e_1 есть 1, в противном случае e_1 есть 0.

Часто определённые комбинации причин невозможны из-за синтаксических или внешних ограничений. Используются перечисленные ниже обозначения ограничений.

Ограничение E (Exclusive, исключает, рис. 14.10) устанавливает, что E должно быть истинным, если причины нулевые или хотя бы одна из причин (a или b) принимает значение 1 (a и b не могут принимать значение 1 одновременно, см. табл. 14.1).

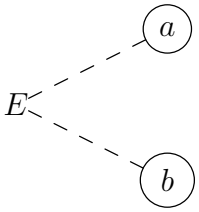


Рис. 14.10. Ограничение E (Exclusive)

Иными словами, E будет иметь истинное значение, если не более, чем одна причина имеет истинное значение (1).

Ограничение I (Inclusive, включает, рис. 14.11, табл. 14.3) устанавливает, что по крайней мере одна из величин, a, b, или c, всегда должна быть равной 1 (a, b, и c не могут принимать значение 0 одновременно).

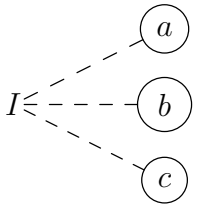


Рис. 14.11. Ограничение I (Inclusive)

Таблица 14.2.

Таблица истинности для ограничения I

Ограничение I	a	b	c
false	0	0	0
true	0	0	1
	0	1	0
	0	1	1
	1	0	0
	1	0	1
	1	1	0
	1	1	1

Ограничение O (Only one, одно и только одно, рис. 14.12, табл. 14.3) устанавливает, что одна и только одна из величин a или b должна быть равна 1.

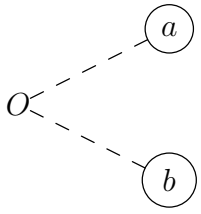


Рис. 14.12. Ограничение O (Only one)

Ограничение R (Requires, требует, рис. 14.13) устанавливает, что если a принимает значение 1, то и b должна принимать значение 1 (нельзя, чтобы a было равно 1, а $b = 0$).

Часто возникает необходимость в ограничениях для следствий.

Ограничение M (Masks, скрывает, рис. 14.14) устанавливает, что если следствие e_1 имеет значение 1, то следствие e_2 должно принять значение 0.

Таблица 14.3.

Таблица истинности для ограничения O

Ограничение O	a	b
false	0	0
true	0	1
	1	0
false	1	1

Рис. 14.13. Ограничение R (Requires)

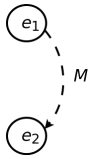


Рис. 14.14. Ограничение M (скрывает, Masks)

3. Организация процесса тестирования ПО

3.1. Методика тестирования программных систем

Процесс тестирования объединяет различные способы тестирования в спланированную последовательность шагов, которые приводят к успешному построению программной системы. Методика тестирования ПС может быть представлена в виде разворачивающейся спирали (рис. 14.15).

Рис. 14.15. Спираль процесса тестирования ПС

В начале осуществляется тестирование элементов (модулей), проверяющее результаты этапа кодирования ПС. На втором шаге выполняется тестирование интеграции, ориентированное на выявление ошибок этапа проектирования ПС. На тре-

тьем обороте спирали производится тестирование правильности, проверяющее корректность этапа анализа требований к ПС. На заключительном витке спирали проводится системное тестирование, выявляющее дефекты этапа системного анализа ПС.

Охарактеризуем каждый шаг процесса тестирования.

1. *Тестирование элементов.* Цель — индивидуальная проверка каждого модуля. Используются способы тестирования «белого ящика».

2. *Тестирование интеграции.* Цель — тестирование сборки модулей в программную систему. В основном применяют способы тестирования «чёрного ящика».

3. *Тестирование правильности.* Цель — проверить реализацию в программной системе всех функциональных и поведенческих требований, а также требования эффективности. Используются исключительно способы тестирования «чёрного ящика».

4. *Системное тестирование.* Цель — проверка правильности объединения и взаимодействия всех элементов компьютерной системы, реализации всех системных функций.

3.2. Тестирование элементов

Объектом тестирования элементов (юнитов, unit) является наименьшая единица проектирования ПС — **модуль**. Для обнаружения ошибок в рамках **модуля** тестируются его важнейшие управляющие пути. Относительная сложность тестов и ошибок определяется как результат ограничений области тестирования элементов. Принцип тестирования — «белый ящик», шаг может выполняться для набора модулей параллельно.

Тестированию подвергаются:

- интерфейс модуля;
- внутренние структуры данных;
- независимые пути;
- пути обработки ошибок (обработка исключительных ситуаций try — catch);
- граничные условия.

Интерфейс модуля тестируется для проверки правильности ввода-вывода тестовой информации. Если нет уверенности в правильном вводе-выводе данных, нет смысла проводить другие тесты.

Исследование внутренних структур данных гарантирует целостность сохраняемых данных.

Тестирование независимых путей гарантирует однократное выполнение всех операторов модуля. При тестировании путей выполнения обнаруживаются следующие категории ошибок: ошибочные вычисления, некорректные сравнения, неправильный поток управления.

Наиболее общими *ошибками вычислений* являются:

1. неправильный или непонятый приоритет арифметических операций;
2. смешанная форма операций (целочисленное деление там, где предполагается вещественное число);
3. некорректная инициализация;
4. несогласованность в представлении точности (связано с особенностями машинной арифметики);
5. некорректное символическое представление выражений.

Источниками *ошибок сравнения* и *неправильных потоков управления* являются:

1. сравнение различных типов данных;
2. некорректные логические операции и приоритетность;
3. ожидание эквивалентности в условиях, когда ошибки точности делают эквивалентность невозможной;
4. некорректное сравнение переменных;
5. неправильное прекращение цикла;
6. отказ в выходе при отклонении итерации;
7. неправильное изменение переменных цикла.

Обычно при проектировании модуля предвидят некоторые ошибочные условия. Для защиты от ошибочных условий в модуль вводят *пути обработки ошибок*. Такие пути тоже должны тестироваться. Тестирование *путей обработки ошибок* можно ориентировать на следующие ситуации:

1. донесение об ошибке невразумительно;
2. текст донесения не соответствует, обнаруженной ошибке;
3. вмешательство системных средств регистрации аварии произошло до обработки ошибки в модуле;
4. обработка исключительного условия некорректна;
5. описание ошибки не позволяет определить её причину.

Модули часто вызывают ошибки на «границах»:

1. при обработке n-го элемента n-элементного массива;
2. при выполнении m-й итерации цикла с m проходами;
3. при появлении минимального (максимального) значения.

Тестовые варианты, ориентированные на данные ситуации, имеют высокую вероятность обнаружения ошибок.

Тестирование элементов обычно рассматривается как дополнение к этапу кодирования. Этот процесс начинается после кодирования модуля.

Так как модуль (юнит) не является автономной системой, то для реализации тестирования требуются дополнительные средства, представленные на рис. 14.16.

Дополнительными средствами являются **драйвер тестирования** и **заглушки**.

> ***Драйвер тестирования*** *— управляющая программа, которая принимает исходные данные и ожидаемые результаты тестовых вариантов, запускает в работу тестируемый модуль, получает от модуля выходные данные и формирует отчёты о тестировании.*

Большинство модулей невозможно запустить без такого драйвера. Например, в C, C++, Java исполняются только main функции (классы).

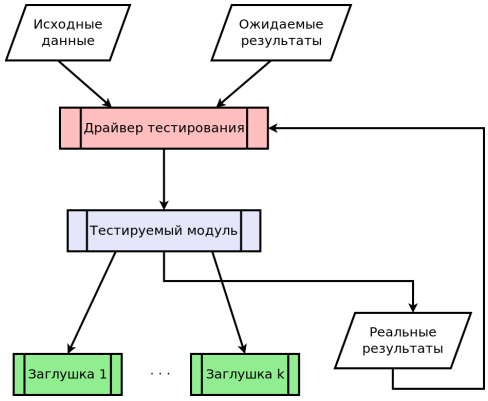


Рис. 14.16. Программная среда для тестирования модуля

Заглушки замещают модули, которые вызываются тестируемым модулем.

> ***Заглушка*** *реализует интерфейс подчинённого модуля, может выполнять минимальную обработку данных, имитирует приём и возврат данных.*

Создание **драйвера** и **заглушек** подразумевает дополнительные затраты, так как они не поставляются с конечным программным продуктом. Если эти средства просты, то дополнительные затраты невелики. Но многие модули не могут быть адекватно протестированы с помощью простых дополнительных средств. В этих случаях *полное тестирование* может быть отложено до шага тестирования интеграции (где драйверы или заглушки также используются).

Тестирование юнита просто осуществить, если он имеет высокую **связность**.

> ***Связность*** *или* ***сцепление*** *(cohesion) — характеристика внутренней взаимосвязи между частями одного модуля.*

При реализации модулем только одной функции количество тестовых вариантов уменьшается, а ошибки легко предсказываются и обнаруживаются.

Связность противоположна **связанности**.

> ***Связанность*** *(coupling) или* ***зависимость*** *(dependency) — характеристика взаимосвязи модуля с другими модулями.*

Это степень, в которой каждый программный модуль полагается на другие модули.

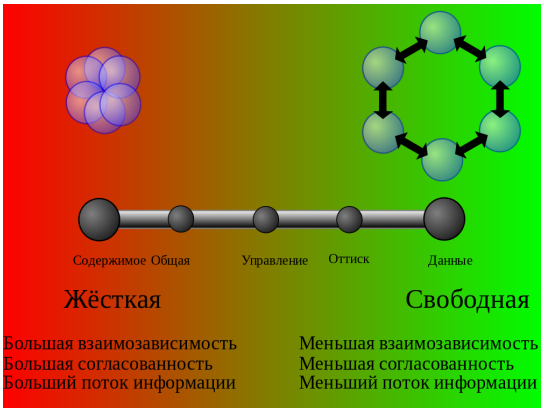


Рис. 14.17. Концептуальная модель связанности

Отладка

> ***Отладка*** *— это локализация и устранение ошибок.*

Отладка является продолжением успешного тестирования. Это значит, что если тестовый вариант обнаруживает ошибку, то процесс отладки уничтожает её.

Процессу отладки предшествует выполнение тестового варианта. Результаты тестового прогона оцениваются, регистрируется несоответствие между ожидаемым и реальным результатами. Несоответствие является симптомом скрытой причины. Процесс отладки пытается сопоставить симптом с причиной, вследствие чего становится возможным обнаружение и исправление ошибки.

Возможны два исхода процесса отладки:

1. причина найдена, исправлена, устранена;
2. причина не найдена.

Во втором случае отладчик может предполагать причину. Для проверки этой причины он просит разработать дополнительный тестовый вариант, который поможет проверить предположение. Таким образом, запускается итерационный процесс коррекции ошибки.

Возможны разные способы проявления ошибок:

1. программа завершается нормально, но выдаёт неверные результаты;
2. программа зависает;
3. программа завершается по прерыванию;
4. программа завершается, выдаёт ожидаемые результаты, но хранимые данные испорчены (это самый неприятный вариант).

Характер проявления ошибок также может меняться. Симптом ошибки может быть:

- постоянным;
- мерцающим;
- пороговым (проявляется при превышении некоторого порога в обработке); Например, 200 самолетов на экране отслеживаются, а 201-й — нет.
- отложенным (проявляется только после исправления маскирующих ошибок).

В ходе отладки встречаются ошибки в широком диапазоне: от мелких неприятностей до катастроф.

Следствием (и причиной) увеличения ошибок часто является усиление давления на отладчика. Часто из-за этого давления разработчик устраняет одну ошибку и вносит две новые ошибки.

Термин debugging (отладка) дословно переводится как «ловля тараканов», который отражает специфику процесса — погоню за объектами отладки, тараканами. Такое название связано с тем, что ошибки в работе первых программных систем были действительно связаны с тем, что тараканы замыкали контакты на платах и реле.

Различают две группы методов отладки:

- *аналитические*;
- *экспериментальные*.

Аналитические методы базируются на анализе выходных данных для тестовых прогонов. *Экспериментальные методы* базируются на использовании вспомогательных средств отладки (отладочные печати, трассировки), позволяющих уточнить характер поведения программы при тех или иных исходных данных.

Общая стратегия отладки — обратное прохождение от замеченного симптома ошибки к исходной аномалии (месту в программе, где ошибка совершена). В простейшем случае место проявления симптома и ошибочный фрагмент совпадают. Но чаще всего они далеко отстоят друг от друга.

Цель отладки — найти оператор программы, при исполнении которого правильные аргументы приводят к неправильным результатам. Если место проявления симптома ошибки не является искомой аномалией, то один из аргументов оператора должен быть неверным. Поэтому надо перейти к исследованию предыдущего оператора, выработавшего этот неверный аргумент. В итоге пошаговое обратное прослеживание приводит к искомому ошибочному месту.

В разных методах прослеживание организуется по-разному. В *аналитических методах* — на основе логических заключений о поведении программы, шаг за шагом уменьшается область программы, подозреваемая в наличии ошибки. Здесь определяется корреляция между значениями выходных данных и особенностями поведения. Основное преимущество *аналитических методов отладки* состоит в том, что исходная программа остаётся без изменений.

В *экспериментальных методах* для прослеживания выполняется:

1. Выдача значений переменных в указанных точках.
2. Трассировка переменных (выдача их значений при каждом изменении).
3. Трассировка потоков управления (имён вызываемых процедур, меток, на которые передаётся управление, номеров операторов перехода).

Преимущество *экспериментальных методов отладки* состоит в том, что основная рутинная работа по анализу процесса вычислений перекладывается на компьютер.

Многие трансляторы имеют встроенные средства отладки для получения информации о ходе выполнения программы, а многие среды разработки имеют встроенный отладчик или используют внешний отладчик для удобного просмотра этой информации и гибкого управления процессом отладки.

Недостаток *экспериментальных методов отладки* — в программу вносятся изменения, которые могут стать причиной новых ошибок. Впрочем, некоторые системы программирования создают специальный отладочный экземпляр программы, а в основной экземпляр не вмешиваются.

Рефакторинг

Отлаженный программный модуль впоследствии подвергается реорганизации кода или **рефакторингу**.

> ***Рефакторинг*** *— процесс улучшения внутренней структуры программы без изменения её функциональности.*

Более широко **рефакторинг** определяют как технику разработки ПО через множество изменений кода, направленных на добавление функциональности и улучшение структуры.

Цель **рефакторинга** — сделать код программы легче для понимания (без этого рефакторинг нельзя считать успешным).

В основе **рефакторинга** лежит последовательность небольших эквивалентных (то есть сохраняющих поведение) преобразований. Поскольку каждое преобразование маленькое, программисту легче проследить за его правильностью, и в то же время вся последовательность может привести к существенной перестройке программы и улучшению её согласованности и чёткости.

Рефакторинг следует отличать от оптимизации производительности. Как и **рефакторинг**, оптимизация обычно не изменяет поведение программы, а только ускоряет её работу. Но оптимизация часто затрудняет понимание кода, что противоположно рефакторингу. *Не жертвуйте лёгкостью чтения программы ради эффек-*

тивности, так как это затруднит последующие тестирования, отладку и сопровождение. Для оптимизации используйте *оптимизирующий компилятор*.

Рефакторинг нужно применять постоянно при разработке кода. Основными стимулами его проведения являются следующие задачи:

1. необходимо добавить новую функцию, которая недостаточно укладывается в принятое архитектурное решение;
2. необходимо исправить ошибку, причины возникновения которой сразу не ясны;
3. преодоление трудностей в командной разработке, которые обусловлены сложной логикой программы.

Во многом при рефакторинге лучше полагаться на интуицию, основанную на опыте. Тем не менее имеются некоторые видимые проблемы в коде (code smells), требующие рефакторинга:

1. дублирование кода;
2. длинный метод;
3. большой класс;
4. длинный список параметров;
5. «завистливые» функции — это метод, который чрезмерно обращается к данным другого объекта;
6. избыточные временные переменные;
7. классы данных;
8. несгруппированные данные.

В экстремальном программировании и других гибких методологиях **рефакторинг** является неотъемлемой частью цикла разработки ПО: разработчики попеременно то создают новые тесты и функциональность, то выполняют **рефакторинг** кода для улучшения его логичности и прозрачности. Автоматическое *юнит-тестирование* позволяет убедиться, что **рефакторинг** не разрушил существующую функциональность.

Иногда под **рефакторингом** неправильно подразумевают стилизацию кода, — коррекцию кода с заранее оговоренными правилами отступа, перевода строк, внесения комментариев и прочими визуально значимыми изменениями, которые никак не отражаются на процессе компиляции, с целью обеспечения лучшей читаемости кода.

Рефакторинг изначально не предназначен для исправления ошибок и добавления новой функциональности, он вообще не меняет поведение программного обеспечения и это помогает избежать ошибок и облегчить добавление функциональности. Он выполняется для улучшения понятности кода или изменения его структуры, для удаления «мёртвого кода» — всё это для того, чтобы в будущем код было легче

поддерживать и развивать. В частности, добавление в программу нового поведения может оказаться сложным в связи с существующей структурой — в этом случае разработчик может выполнить необходимый **рефакторинг**, а уже затем добавить новую функциональность.

Это может быть перемещение атрибута из одного класса в другой, вынесение фрагмента кода из метода и превращение его в самостоятельный метод или даже перемещение кода по иерархии классов. Каждый отдельный шаг может показаться элементарным, но совокупный эффект таких малых изменений в состоянии радикально улучшить проект или даже предотвратить распад плохо спроектированной программы.

Методы рефакторинга

Наиболее употребимые методы **рефакторинга**:

- Изменение сигнатуры метода (Change Method Signature).

  Суть изменения сигнатуры метода заключается в добавлении, изменении или удалении параметра метода. Изменив сигнатуру метода, необходимо скорректировать обращения к нему в коде всех клиентов. Это изменение может затронуть внешний интерфейс программы, кроме того, не всегда разработчику, изменяющему интерфейс, доступны все клиенты этого интерфейса, поэтому может потребоваться та или иная форма регистрации изменений интерфейса для последующей передачи их вместе с новой версией программы.

- Инкапсуляция поля (Encapsulate Field).

  В случае, если у класса имеется открытое поле, необходимо сделать его закрытым и обеспечить методы доступа. После «Инкапсуляции поля» часто применяется «Перемещение метода».

- Выделение класса (Extract Class).

- Выделение интерфейса (Extract Interface).

- Выделение локальной переменной (Extract Local Variable).

- Выделение метода (Extract Method).

  Выделение метода заключается в выделении из длинного и/или требующего комментариев кода отдельных фрагментов и преобразовании их в отдельные методы, с подстановкой подходящих вызовов в местах использования. В этом случае действует правило: если фрагмент кода требует комментария о том, что он делает, то он должен быть выделен в отдельный метод. Также правило: один метод не должен занимать более чем один экран (25-50 строк, в зависимости от условий редактирования), в противном случае некоторые его фрагменты имеют самостоятельную ценность и подлежат выделению. Из анализа связей выделяемого фрагмента с окружающим контекстом делается вывод о перечне параметров нового метода и его локальных переменных.

- Генерализация типа (Generalize Type).

- Встраивание (Inline).
- Введение фабрики (Introduce Factory).
- Введение параметра (Introduce Parameter).
- Подъём метода (Pull Up Method).
- Спуск метода (Push Down Method).
- Переименование метода (Rename Method).
- Перемещение метода (Move Method).

  Перемещение метода применяется по отношению к методу, который чаще обращается к другому классу, чем к тому, в котором сам располагается (так называемые завистливые функции).
- Замена наследования делегированием (Replace Inheritance with Delegation).
- Замена кода типа подклассами (Replace Type Code with Subclasses).
- Замена условного оператора полиморфизмом (Replace Conditional with Polymorphism).

  Условный оператор с несколькими ветвями заменяется вызовом полиморфного метода некоторого базового класса, имеющего подклассы для каждой ветви исходного оператора. Выбор ветви осуществляется неявно, в зависимости от того, экземпляру какого из подклассов оказался адресован вызов.

Основные шаги:

1. вначале следует создать базовый класс и нужное число подклассов;
2. в некоторых случаях следует провести оптимизацию условного оператора путём «Выделения метода»;
3. возможно использование «Перемещения метода», чтобы поместить условный оператор в вершину иерархии наследования;
4. выбрав один из подклассов, нужно конкретизировать в нём полиморфный метод базового класса и переместить в него тело соответствующей ветви условного оператора;
5. повторить предыдущее действие для каждой ветви условного оператора;
6. заменить весь условный оператор вызовом полиморфного метода базового класса.

Проблемы, возникающие при проведении рефакторинга

- проблемы, связанные с базами данных;
- проблемы изменения интерфейсов;
- трудности при изменении дизайна.

Средства автоматизации рефакторинга

Технические критерии для инструментов **рефакторинга**:

- базы данных программы;
- деревья синтаксического разбора;
- точность.

Практические критерии для инструментов **рефактор**инга:

- скорость;
- отмена модификаций;
- интеграция с другими инструментами.

3.3. Тестирование интеграции

> ***Тестирование интеграции*** *предназначено для проверки работы программной системы в цельной сборке.*

Цель сборки и **тестирования интеграции**: взять модули, протестированные как элементы, и построить программную структуру, требуемую проектом.

Тесты проводятся для обнаружения ошибок интерфейса. Вот некоторые категории ошибок интерфейса:

- потеря данных при прохождении через интерфейс;
- отсутствие в модуле необходимой ссылки;
- неблагоприятное влияние одного модуля на другой;
- подфункции при объединении не образуют требуемую главную функцию;
- отдельные (допустимые) неточности при интеграции выходят за допустимый уровень;
- проблемы при работе с глобальными структурами данных.

Сборка модулей в программный комплекс может осуществляться двумя методами:

- **монолитным**,
- **пошаговым**.

Пошаговая сборка может, в свою очередь, быть

- *восходящей* (снизу-вверх);
- *нисходящей* (сверху-вниз).

Существует два варианта тестирования, поддерживающих процесс интеграции: **нисходящее тестирование** и **восходящее тестирование**.

Нисходящее тестирование интеграции

В данном подходе модули объединяются движением *сверху вниз* по управляющей иерархии, начиная от главного управляющего модуля. Подчинённые модули добавляются в структуру или в результате **поиска в глубину**, или в результате **поиска в ширину**. Интеграция **поиском в глубину** будет подключать все модули, находящиеся на **главном управляющем пути** структуры (по вертикали).

Выбор **главного управляющего пути** отчасти произволен и зависит от характеристик, определяемых приложением.

При интеграции **поиском в ширину** структура последовательно проходится по уровням-горизонталям.

Пример .54 (интеграция поиском в глубину и в ширину). Интеграция **поиском в глубину** будет подключать все модули, находящиеся на **главном управляющем пути** структуры (по вертикали). При выборе левого пути (рис. 14.18) прежде всего будут подключены модули M1, M2, M5. Следующим подключается модуль M8 или M6 (если это необходимо для правильного функционирования M2). Затем строится центральный или правый управляющий путь.

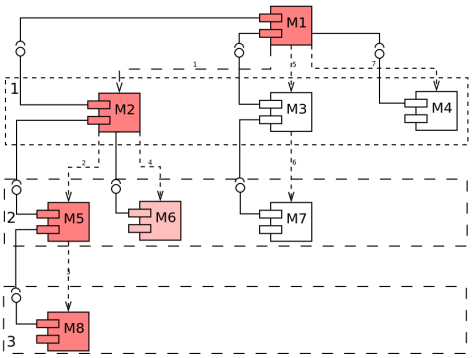


Рис. 14.18. Нисходящая интеграция системы

При интеграции **поиском в ширину** структура последовательно проходится по уровням-горизонталям. На каждом уровне подключаются модули, непосредственно подчинённые управляющему модулю — начальнику. В этом случае прежде всего подключаются модули M2, M3, M4. На следующем уровне — модули M5, M6 и т. д.

Шаги процесса нисходящей интеграции

1. Главный управляющий модуль (находится на вершине иерархии) используется как **тестовый драйвер**. По сути это unit-test главного модуля. Все непосредственно подчинённые ему модули временно замещаются **заглушками**.

2. Одна из **заглушек** заменяется реальным модулем. Модуль выбирается *поиском* в **ширину** или в **глубину**.

3. После подключения каждого модуля (и установки на нём заглушек) проводится набор тестов, проверяющих полученную структуру.

4. Если в модуле-драйвере уже нет заглушек, производится смена модуля-драйвера (поиском в ширину или в глубину).

5. Выполняется возврат на шаг 2 (до тех пор, пока не будет построена вся структура).

Достоинство нисходящей интеграции: ошибки в главной, управляющей части системы выявляются в первую очередь.

Недостаток: трудности в ситуациях, когда для полного тестирования на верхних уровнях нужны результаты обработки с нижних уровней иерархии.

Существуют 3 возможности борьбы с этим недостатком:

1. Откладывать некоторые тесты до замещения заглушек модулями. При этом возникают сложности в оценке результатов тестирования.

2. Разрабатывать заглушки, частично выполняющие функции модулей.

3. Подключать модули движением снизу вверх. Эту возможность обсудим отдельно.

В тестах тестируется конкретный, определённый объект и то, как именно он взаимодействует с внешними зависимостями.

> ***Внешняя зависимость*** *— это объект, с которым взаимодействует код и над которым нет прямого контроля.*

Для ликвидации внешних зависимостей используются **тестовые объекты**, например такие как stubs (заглушки).

Существует классический труд под названием «xUnit test patterns: refactoring test code», автор *Жерард Месарош* (*Gerard Meszaros*) в котором автор вводит 5 видов тестовых объектов:

dummy object — обычно *передаётся в тестируемый класс в качестве параметра*, но не имеет поведения, с ним ничего не происходит, никакие методы не вызываются.

Пример .55 (dummy-объекты). Примером таких dummy-объектов являются:

```
new object();
null;
"Ignored String";
```

test stub (заглушка) — используется *для получения данных из внешней зависимости*, подменяя её. При этом игнорирует все данные, могущие поступать из тестируемого объекта в **заглушку**. Один из самых популярных видов тестовых объектов.

К примеру, если тестируемый объект использует чтение из конфигурационного файла, тогда заглушаем его `ConfigFileStub`, возвращающим тестовые строки конфигурации для избавления зависимости от файловой системы.

test spy (тестовый шпион) — используется для тестов взаимодействия, основной функцией является *запись данных и вызовов, поступающих из тестируемого объекта* для последующей проверки корректности вызова зависимого объекта. Позволяет проверить логику тестируемого объекта, без проверок зависимых объектов.

mock object (мок-объект) — реализет заданные аспекты моделируемого программного окружения. Мок-объект представляет собой конкретную фиктивную реализацию интерфейса, предназначенную исключительно для тестирования. Он очень похож на **тестовый шпион**, однако не записывает последовательность вызовов с переданными параметрами для последующей проверки, а может сам *выдавать исключения при некорректно переданных данных. Именно* ***мок-объект*** *проверяет корректность поведения тестируемого объекта.* В процедурном программировании аналогичная конструкция называется «dummy» (заглушка). Функция, выдающая константу, или случайную величину из допустимого диапазона значений.

fake object (фальшивый объект) — используется в основном для того, чтобы быстрее запускать тесты и для ускорения их работы. Эдакая замена тяжеловесного внешнего зависимого объекта его легковесной реализацией. Основные примеры — эмулятор для конкретного приложения БД в памяти (fake database) или фальшивый вебсервис.

Восходящее тестирование интеграции

При **восходящем тестировании интеграции** сборка и тестирование системы начинаются с модулей-атомов, располагаемых на нижних уровнях иерархии. Модули подключаются движением снизу вверх. Подчинённые модули всегда доступны, и нет необходимости в заглушках.

Шаги методики **восходящей интеграции**.

1. Модули нижнего уровня объединяются в кластеры (группы, блоки), выполняющие определённую программную подфункцию.
2. Для координации вводов-выводов тестового варианта пишется драйвер, управляющий тестированием кластеров.
3. Тестируется кластер.
4. Драйверы удаляются, а кластеры объединяются в структуру движением вверх.

Пример *восходящей интеграции системы* приведён на рис. 14.19.

Пример .56 (восходящая интеграция системы). Модули объединяются в кластеры 1,2,3 (рис. 8.6). Каждый кластер тестируется своим драйвером. Модули в кластерах 1 и 2 подчинены модулю Ma, поэтому драйверы D1 и D2 удаляются и кластеры подключают прямо к Ma. Аналогично драйвер D3 удаляется перед подключением кластера 3 к модулю Mb. В последнюю очередь к модулю Mc подключаются модули Ma и Mb.

По мере продвижения интеграции вверх необходимость в выделении **драйверов** уменьшается. Как правило, в двухуровневой структуре **драйверы** не нужны.

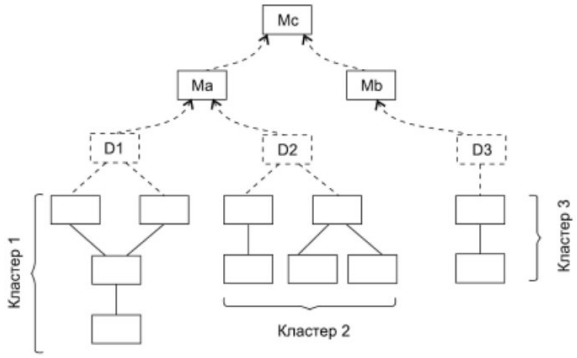


Рис. 14.19. Восходящая интеграция системы

Сравнение нисходящего и восходящего тестирования интеграции

Нисходящее тестирование:

- *основной недостаток* — необходимость **заглушек** и связанные с ними трудности тестирования;
- *основное достоинство* — возможность раннего тестирования главных управляющих функций.

Восходящее тестирование:

- *основной недостаток* — необходимость **драйверов** и связанные с ними трудности тестирования;
- *основное достоинство* — упрощается разработка тестовых вариантов, отсутствуют **заглушки** (**драйверов**, как правило, требуется меньше, чем **заглушек**).

Возможен комбинированный подход: для верхних уровней иерархии применяют *нисходящую стратегию*, а для нижних уровней — *восходящую стратегию тестирования*. При проведении тестирования интеграции очень важно выявить **критические модули**.

Признаки **критического модуля**:

- реализует несколько требований к программной системе;
- имеет высокий уровень управления (находится достаточно высоко в программной структуре);
- имеет высокую сложность или склонность к ошибкам (как индикатор может использоваться цикломатическая сложность — её верхний разумный предел составляет 10);
- имеет определённые требования к производительности обработки.

Критические модули должны тестироваться как можно раньше. Кроме того, к ним должно применяться **регрессионное тестирование** (повторение уже выполненных тестов в полном или частичном объёме).

3.4. Тестирование правильности

После окончания тестирования интеграции программная система собрана в единую систему, интерфейсные ошибки уже обнаружены и откорректированы. Теперь начинается последний шаг программного тестирования — **тестирование правильности**. Цель — подтвердить, что реализованные в системе функции (описанные в спецификации требований к ПС) соответствуют ожиданиям заказчика. Подтверждение правильности ПС выполняется с помощью тестов «чёрного ящика», демонстрирующих соответствие требованиям. При обнаружении отклонений от спецификации требований создаётся список недостатков.

Как правило, отклонения и ошибки, выявленные при подтверждении правильности, требуют изменения сроков разработки продукта. Важным элементом подтверждения правильности является проверка **конфигурации** ПС.

Конфигурацией ПС называют совокупность всех элементов информации, вырабатываемых в процессе конструирования ПС. Минимальная конфигурация ПС ранее была представлена на стр. 27.

Проверка конфигурации гарантирует, что все элементы конфигурации ПС правильно разработаны, учтены и достаточно детализированы для поддержки этапа сопровождения в жизненном цикле ПС.

Разработчик не может предугадать, как заказчик будет реально использовать ПС. Для обнаружения ошибок, которые способен найти только конечный пользователь, используют процесс, включающий **альфа-** и **бета-тестирование**.

> ***Альфа-тестирование*** *проводится заказчиком в организации разработчика, разработчик фиксирует все выявленные заказчиком ошибки и проблемы использования.*

> ***Бета-тестирование*** *проводится конечным пользователем в организации заказчика, разработчик в этом процессе участия не принимает.*

Фактически, **бета-тестирование** — это реальное применение ПС в среде, которая не управляется разработчиком. Заказчик сам записывает все обнаруженные проблемы и сообщает о них разработчику. **Бета-тестирование** проводится в течение фиксированного срока (около года). По результатам выявленных проблем разработчик изменяет ПС и тем самым подготавливает продукт полностью на базе заказчика.

3.5. Системное тестирование

Системное тестирование подразумевает выход за рамки области действия программного проекта и проводится не только разработчиком. Классическая проблема **системного тестирования** — указание причины. Она возникает, когда разработчик одного системного элемента обвиняет разработчика другого элемента в причине возникновения дефекта.

Для защиты от подобного обвинения разработчик программного элемента должен:

- предусмотреть средства обработки ошибки, которые тестируют все вводы информации из других элементов системы;
- провести тесты, моделирующие неудачные данные или другие потенциальные ошибки интерфейса ПС;
- записать результаты тестов, чтобы использовать их как доказательство невиновности в случае «указания причины»;
- принять участие в планировании и проектировании **системных тестов**, чтобы гарантировать адекватное тестирование ПС.

> ***Системные тесты*** *должны проверять, что все системные элементы правильно объединены и выполняют назначенные функции.*

Рассмотрим основные типы **системных тестов**.

Тестирование восстановления

Многие компьютерные системы должны восстанавливаться после отказов и возобновлять обработку в пределах заданного времени. В некоторых случаях система должна быть **отказоустойчивой**, то есть отказы обработки не должны быть причиной прекращения работы системы. В других случаях системный отказ должен быть устранён в пределах заданного интервала времени, иначе заказчику наносится серьезный экономический ущерб.

Тестирование восстановления использует самые разные пути для того, чтобы заставить ПС отказать, и проверяет полноту выполненного восстановления. При *автоматическом восстановлении* оцениваются правильность повторной инициализации, механизмы копирования контрольных точек, восстановление данных, перезапуск. При *ручном восстановлении* оценивается, находится ли среднее время восстановления в допустимых пределах.

Тестирование безопасности

Компьютерные системы очень часто являются мишенью незаконного проникновения. Под проникновением понимается широкий диапазон действий: попытки хакеров проникнуть в систему из спортивного интереса, месть рассерженных служащих, взлом мошенниками для незаконной наживы. **Тестирование безопасности** проверяет фактическую реакцию защитных механизмов, встроенных в систему, на проникновение.

В ходе тестирования безопасности испытатель играет роль взломщика. Ему разрешено всё:

- попытки узнать пароль с помощью внешних средств;
- атака системы с помощью специальных утилит, анализирующих защиты;
- подавление, ошеломление системы, (D)DoS-атаки, переполнение буфера (в надежде, что она откажется обслуживать других клиентов);

- целенаправленное введение ошибок (инъекции) в надежде проникнуть в систему в ходе восстановления;
- просмотр несекретных данных в надежде найти ключ для входа в систему.

Конечно, при неограниченном времени и ресурсах хорошее **тестирование безопасности** взломает любую систему. Задача проектировщика системы — сделать цену проникновения более высокой, чем цена получаемой в результате информации.

Многие разработчики ПО устраивают открытые конкурсы хакеров по взлому своей новой системы. Такие конкурсы имеют строгие временны́е границы. Это хороший способ провести **тестирование безопасности**, результаты которого не вызовут сомнений у конечных пользователей.

Стрессовое тестирование

На предыдущих шагах тестирования способы «белого» и «чёрного ящиков» обеспечивали полную оценку нормальных программных функций и качества функционирования.

> ***Стрессовые тесты*** *проектируются для навязывания программам ненормальных ситуаций.*

В сущности, проектировщик стрессового теста спрашивает, как сильно можно расшатать систему, прежде чем она откажет? По существу, испытатель пытается разрушить систему. **Стрессовое тестирование** производится при ненормальных запросах на ресурсы системы (по количеству, частоте, размеру).

Пример .57 (стрессовые тесты).
- генерируется 10 прерываний в секунду (при средней частоте 1–2);
- скорость ввода данных увеличивается прямо пропорционально их важности (чтобы определить реакцию входных функций);
- формируются варианты, требующие максимума памяти (или других ресурсов);
- генерируются варианты, вызывающие переполнение виртуальной памяти;
- проектируются варианты, вызывающие чрезмерный поиск данных на диске.

Разновидность стрессового тестирования называется **тестированием чувствительности**. В некоторых ситуациях (обычно в математических алгоритмах) очень малый диапазон данных, содержащийся в границах правильных данных системы, может вызвать ошибочную обработку или резкое понижение производительности.

> ***Тестирование чувствительности*** *обнаруживает комбинации данных, которые могут вызвать нестабильность или неправильность обработки.*

Тестирование производительности

В *системах реального времени* и *встроенных системах* недопустимо ПО, которое реализует требуемые функции, но не соответствует *требованиям производительности.*

> ***Тестирование производительности*** *проверяет скорость работы ПО в компьютерной системе.*

Производительность тестируется на всех шагах процесса **тестирования**. Даже на уровне элемента при проведении тестов «белого ящика» может оцениваться производительность индивидуального модуля. Тем не менее, пока все системные элементы не объединятся полностью, не может быть установлена истинная производительность системы.

Иногда **тестирование производительности** сочетают со **стрессовым тестированием**. При этом нередко требуется специальный аппаратный и программный инструментарий. Например, часто требуется точное измерение используемого ресурса (процессорного цикла и т. д.). Внешний инструментарий регулярно отслеживает интервалы выполнения, регистрирует события (например, прерывания) и машинные состояния. С помощью инструментария испытатель может обнаружить состояния, которые приводят к деградации и возможным отказам системы.