Технология разработки программного обеспечения

Лекция №1

Определение технологии разработки программного обеспечения (ПО). Программные продукты (изделия). Жизненный цикл ПО.

Известно, что **основной задачей первых трех десятилетий компьютерной эры являлось развитие аппаратных компьютерных средств**. Это было обусловлено высокой стоимостью обработки и хранения данных. В 80-е годы успехи микроэлектроники привели к резкому увеличению производительности компьютера при значительном снижении стоимости.

Основной задачей 90-х годов и начала XXI века стало совершенствование качества компьютерных приложений, возможности которых целиком определяются программным обеспечением (ПО).

Современный персональный компьютер теперь имеет производительность большой ЭВМ 80-х годов. Сняты практически все аппаратные ограничения на решение задач. Оставшиеся ограничения приходятся на долю ПО.

Чрезвычайно **актуальными** стали следующие проблемы:

- аппаратная сложность опережает наше умение строить ПО, использующее потенциальные возможности аппаратуры;
- наше умение строить новые программы отстает от требований к новым программам;
- нашим возможностям эксплуатировать существующие программы угрожает низкое качество их разработки.

Ключом к решению этих проблем является грамотная организация процесса создания ПО, реализация технологических принципов промышленного конструирования программных систем (ПС).

Настоящий курс лекций посвящен систематическому изложению принципов современных технологий построения качественного программного обеспечения, моделей и методов (формирования требований, анализа, синтеза и тестирования), используемых в инженерном цикле разработки, сложных программных продуктов.

Литература.

1. С. Орлов. Технологии разработки программного обеспечения: Учебник. — СПб.: Питер, 2002. — 464 с.
2. Иванова Г.С. Технология программирования: Учебник для вузов. - М.: Изд-во МГТУ им. Н.Э. Баумана, 2002. - 320 с.: ил. (Сер. Информатика в техническом университете.)

Определение технологии разработки (конструирования) программного обеспечения

Технология разработки (конструирования) программного обеспечения (ТРПО) — система инженерных принципов для создания экономичного ПО, которое надежно и эффективно работает в реальных компьютерах

Различают методы, средства и процедуры ТРПО.

Методы обеспечивают решение следующих задач:

- планирование и оценка проекта;
- анализ системных и программных требований;
- проектирование алгоритмов, структур данных и программных структур;
- кодирование;
- тестирование;
- сопровождение.

Средства (утилиты) ТРПО обеспечивают автоматизированную или автоматическую поддержку методов. В целях совместного применения утилиты могут объединяться в системы автоматизированного конструирования ПО. Такие системы принято называть CASE-системами. Аббревиатура CASE расшифровывается как **C**omputer **A**ided **S**oftware **E**ngineering (программная инженерия с компьютерной поддержкой).

Процедуры являются «клеем», который соединяет методы и утилиты так, что они обеспечивают непрерывную технологическую цепочку разработки.

Процедуры определяют:

- порядок применения методов и утилит;
- формирование отчетов, форм по соответствующим требованиям;
- контроль, который помогает обеспечивать качество и координировать изменения;
- формирование «вех», по которым руководители оценивают прогресс.

Процесс конструирования программного обеспечения состоит из последовательности шагов, использующих методы, утилиты и процедуры. **Эти последовательности шагов часто называют парадигмами ТРПО**. Применение парадигм ТРПО гарантирует систематический, упорядоченный подход к промышленной разработке, использованию и сопровождению ПО. Фактически, парадигмы вносят в процесс создания ПО организующее инженерное начало, необходимость которого трудно переоценить.

Наиболее популярные парадигмы ТРПО.

Классический жизненный цикл

Старейшей парадигмой процесса разработки ПО является классический жизненный цикл (автор Уинстон Ройс, 1970).

Royce, Walker W. Managing the development of large software systems: concepts and techniques. Proc. IEEE WESTCON, Los Angeles, August 1970, pp. 1-9.

Очень часто классический жизненный цикл называют каскадной или водопадной моделью, подчеркивая, что разработка рассматривается как последовательность этапов, причем переход на следующий, иерархически нижний этап происходит только после полного завершения работ на текущем этапе (рис. ниже).

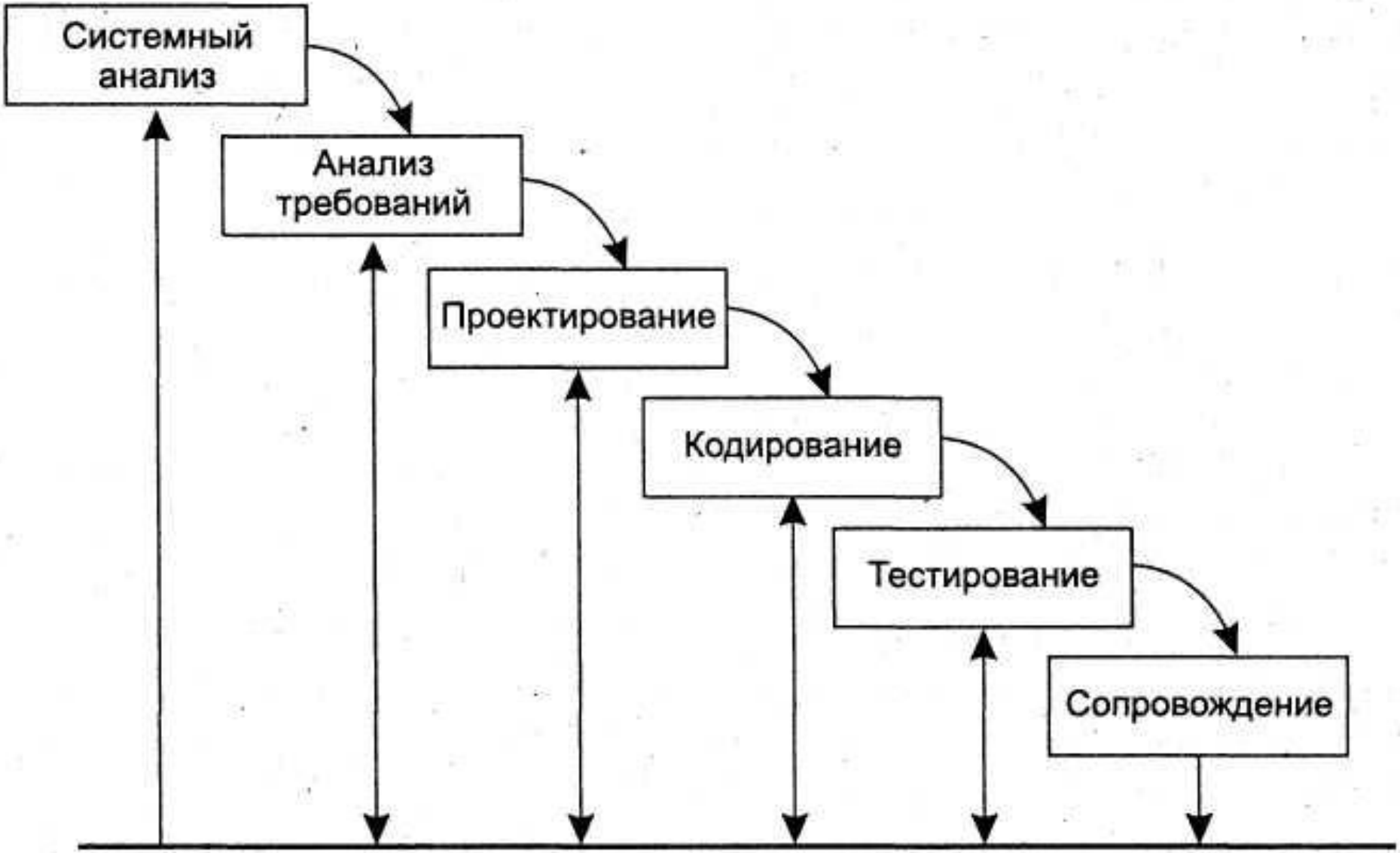


Рис. Классический жизненный цикл разработки ПО

Системный анализ задает роль каждого элемента в компьютерной системе, взаимодействие элементов друг с другом. Поскольку ПО является лишь частью большой системы, то анализ начинается с определения требований ко всем системным элементам и назначения подмножества этих требований программному «элементу». Необходимость системного подхода явно проявляется, когда формируется интерфейс ПО с другими элементами (аппаратурой, людьми, базами данных). На этом же этапе начинается решение задачи планирования проекта ПО.

В ходе планирования проекта определяются объем проектных работ и их риск, необходимые трудозатраты, формируются рабочие задачи и план-график работ.

Анализ требований относится к программному элементу — программному обеспечению. Уточняются и детализируются его функции, характеристики и интерфейс.
Все определения документируются в ***спецификации анализа***. Здесь же завершается решение задачи планирования проекта.

Проектирование состоит в создании представлений:

- архитектуры ПО;
- модульной структуры ПО;
- алгоритмической структуры ПО;
- структуры данных;
- входного и выходного интерфейса (входных и выходных форм данных).

Исходные данные для проектирования содержатся в ***спецификации анализа***, то есть в ходе проектирования выполняется трансляция требований к ПО во множество проектных представлений. При решении задач проектирования основное внимание уделяется качеству будущего программного продукта.

Кодирование состоит в переводе результатов проектирования в текст на языке программирования.

Тестирование — выполнение программы для выявления дефектов в функциях, логике и форме реализации программного продукта.

Сопровождение — это внесение изменений в эксплуатируемое ПО.

Цели изменений:

- исправление ошибок;
- адаптация к изменениям внешней для ПО среды;
- усовершенствование ПО по требованиям заказчика.

Сопровождение ПО состоит в повторном применении каждого из предшествующих шагов (этапов) жизненного цикла к существующей программе, но не в разработке новой программы.

Как и любая инженерная схема, классический жизненный цикл имеет достоинства и недостатки.

***Достоинства классического жизненного цикла**:* дает план и временной график по всем этапам проекта, упорядочивает ход конструирования.

***Недостатки классического жизненного цикла**:*

- реальные проекты часто требуют отклонения от стандартной последовательности шагов;
- цикл основан на точной формулировке исходных требований к ПО (реально в начале проекта требования заказчика определены лишь частично);
- результаты проекта доступны заказчику только в конце работы.

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 приспосабливаемости продукта под операционную систему, форме диалога с пользователем или в эффективности реализуемого алгоритма. В этих случаях целесообразно использовать **макетирование.**

Макетирование

Основная цель макетирования — снять неопределенности в требованиях заказчика.

Макетирование (прототипирование) — это процесс создания модели требуемого программного продукта.

Модель может принимать одну из трех форм:

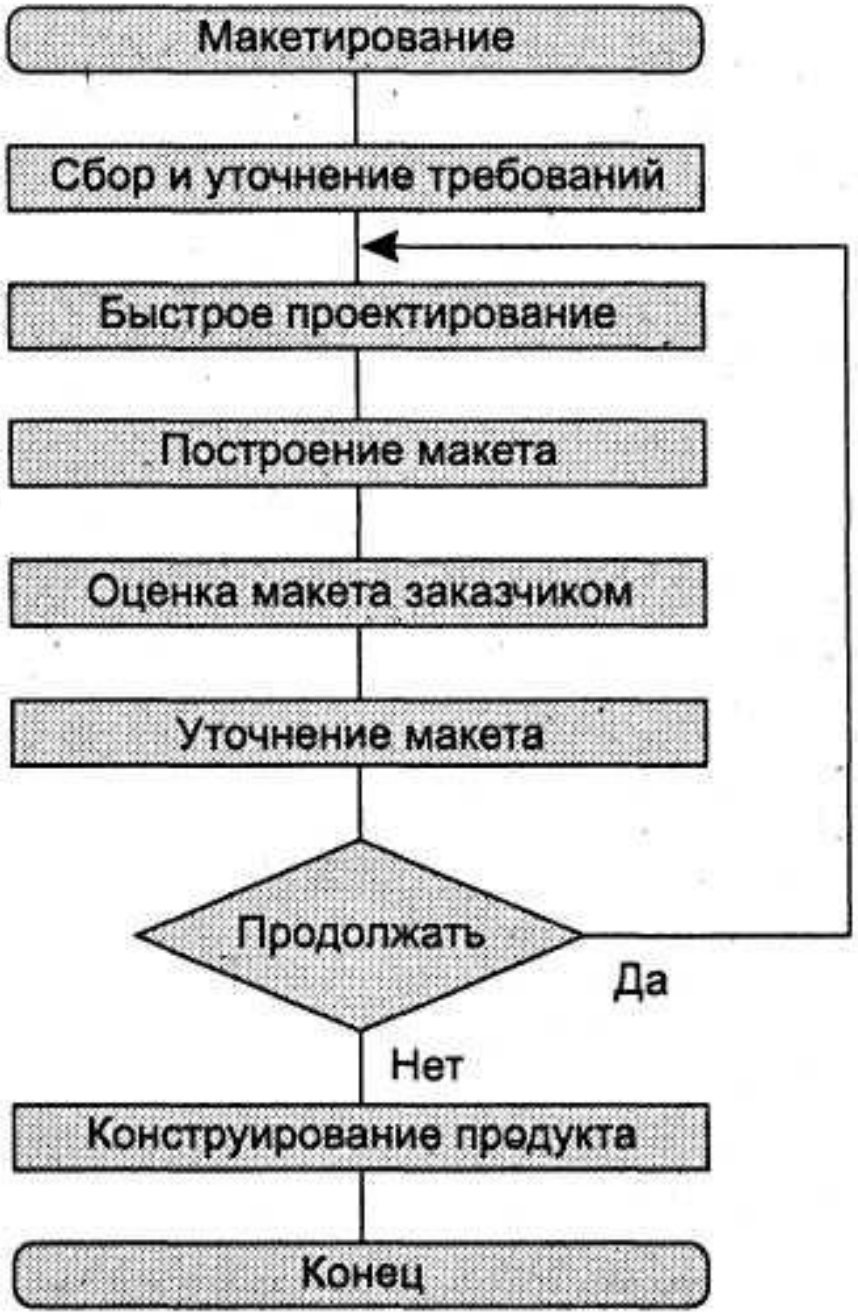
- бумажный макет или макет на основе ПК (изображает или рисует человеко-машинный диалог);
- работающий макет (выполняет некоторую часть требуемых функций);
- существующая программа (характеристики которой затем должны быть улучшены).

Макетирование основывается на многократном повторении итераций, в которых участвуют заказчик и разработчик.

Рис. Макетирование

Рис.

Последовательность действий при макетировании



Макетирование начинается со сбора и уточнения требований к создаваемому ПО. Разработчик и заказчик встречаются и определяют все цели ПО, устанавливают, какие требования известны, а какие предстоит доопределить.

Затем выполняется быстрое проектирование. В нем внимание сосредоточивается на тех характеристиках ПО, которые должны быть видимы пользователю.

Быстрое проектирование приводит к построению макета.

Макет оценивается заказчиком и используется для уточнения требований к ПО.

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 ПО.

***Недостатки макетирования**:*

- заказчик может принять макет за продукт;
- разработчик может принять макет за продукт.

Суть недостатков заключается в следующем. Когда заказчик видит работающую версию ПО, он перестает сознавать, что детали макета скреплены «жевательной резинкой и проволокой»; он забывает, что в погоне за работающим вариантом оставлены нерешенными вопросы качества и удобства сопровождения ПО. Когда заказчику говорят, что продукт должен быть перестроен, он начинает возмущаться и требовать, чтобы макет «в три приема» был превращен в рабочий продукт. Очень часто это отрицательно сказывается на управлении разработкой ПО

С другой стороны, для быстрого получения работающего макета разработчик часто идет на определенные компромиссы. Могут использоваться не самые подходящие язык программирования или операционная система. Для простой демонстрации возможностей может применяться неэффективный алгоритм. Спустя некоторое время разработчик забывает о причинах, по которым эти средства не подходят. В результате далеко не идеальный выбранный вариант интегрируется в систему.

Очевидно, что преодоление этих недостатков требует борьбы с житейским соблазном — принять желаемое за действительное.

Стратегии конструирования ПО

Существуют 3 стратегии конструирования ПО:

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.

Инкрементная модель

Инкрементная модель является классическим примером инкрементной стратегии конструирования (рис. ниже). Она объединяет элементы последовательной водопадной модели с итерационной философией макетирования.

Отметим, что современная реализация инкрементного подхода — экстремальное программирование XP. Оно ориентировано на очень малые приращения функциональности.

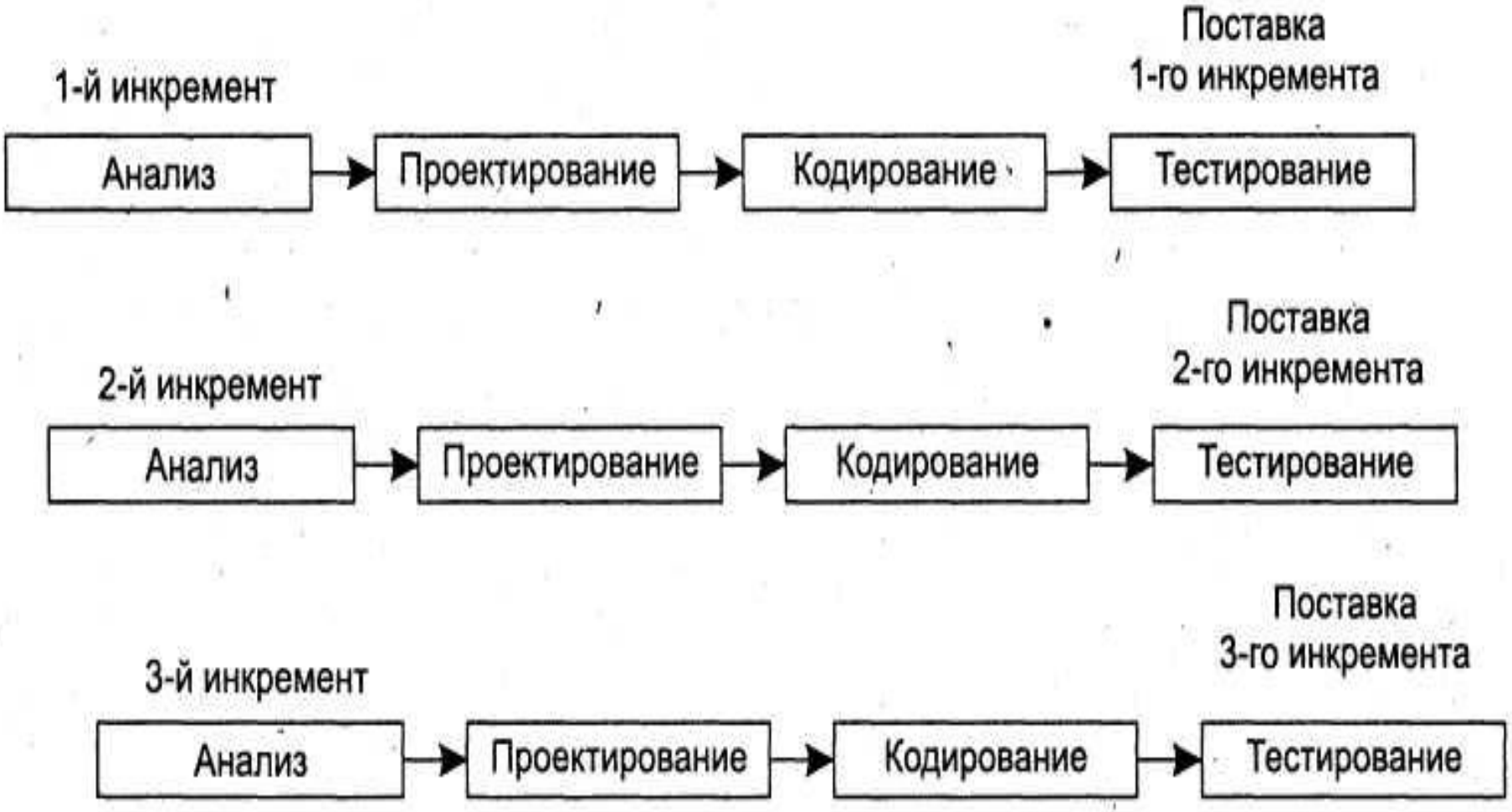


Рис. Инкрементная модель

Каждая линейная последовательность здесь вырабатывает поставляемый инкремент ПО. Например, ПО 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 итеративен, но, в отличие от макетирования, инкрементная модель обеспечивает на каждом инкременте работающий продукт.

Быстрая разработка приложений

Модель быстрой разработки приложений (Rapid Application Development) — второй пример применения инкрементной стратегии конструирования (рис. ниже).

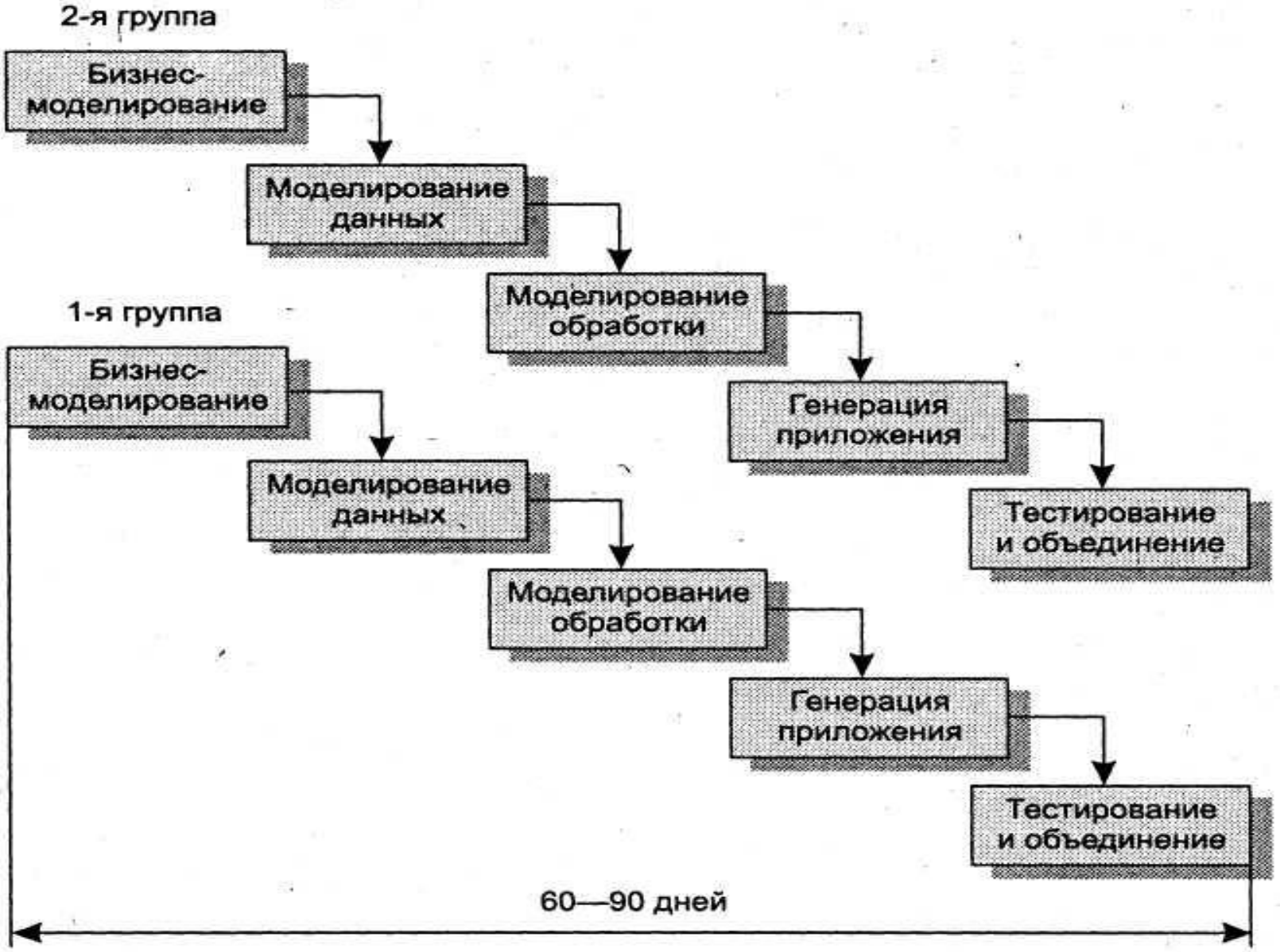


Рис. Модель быстрой разработки приложений

RAD-модель обеспечивает экстремально короткий цикл разработки. RAD — высокоскоростная адаптация линейной последовательной модели, в которой быстрая разработка достигается за счет использования компонентно-ориентированного конструирования. Если требования полностью определены, а проектная область ограничена, RAD-процесс позволяет группе создать полностью функциональную систему за очень короткое время (60-90 дней).

RAD-подход ориентирован на разработку информационных систем и выделяет следующие этапы:

Бизнес-моделирование.

Моделируется информационный поток между бизнес-функциями.
Необходимо ответить на следующие вопросы.

Какая информация руководит бизнес-процессом?
Какая генерируется информация?
Кто генерирует ее?
Где информация применяется?
Кто обрабатывает ее?

Моделирование данных.

Информационный поток, определенный на этапе бизнес - моделирования, отображается в набор объектов данных, которые требуются для поддержки бизнеса. Идентифицируются характеристики (свойства, атрибуты) каждого объекта, определяются отношения между объектами.

Моделирование обработки.

Определяются преобразования объектов данных, обеспечивающие реализацию бизнес-функций. Создаются описания обработки для добавления, модификации, удаления или нахождения (исправления) объектов данных.

Генерация приложения.

Предполагается использование методов, ориентированных на языки программирования 4-го поколения. Вместо создания ПО с помощью языков программирования 3-го поколения, RAD-процесс работает с повторно используемыми программными компонентами или создает повторно используемые компоненты. Для обеспечения конструирования используются утилиты автоматизации.

Тестирование и объединение.

Поскольку применяются повторно используемые компоненты, многие программные элементы уже протестированы. Это уменьшает время тестирования (хотя все новые элементы должны быть протестированы).

Применение RAD возможно в том случае, когда каждая главная функция может быть завершена за 3 месяца. Каждая главная функция адресуется отдельной группе разработчиков, а затем интегрируется в целую систему.

Применение RAD имеет свои **недостатки и ограничения.**

1. Для больших проектов в RAD требуются существенные людские ресурсы (необходимо создать достаточное количество групп).
2. RAD применима только для таких приложений, которые могут разбиты на отдельные модули и в которых производительность не является критической величиной.
3. RAD не применима в условиях высоких технических рисков (то есть при использовании новой технологии).

Спиральная модель

Спиральная модель — классический пример применения эволюционной стратегии конструирования.

Спиральная модель (автор Барри Боэм, 1988) базируется на лучших свойствах классического жизненного цикла и макетирования, к которым добавляется новый элемент — анализ риска, отсутствующий в этих парадигмах.

Boehm, B.W. A spiral model of software development and enhancement. IEEE Computer, 21(5), 1988, pp. 61-72.

Рис. Спиральная модель: *1* — начальный сбор требований и планирование проекта; *2* — та же работа, но на основе рекомендаций заказчика; 3 — анализ риска на основе начальных требований; 4 — анализ риска на основе реакции заказчика; 5 — переход к комплексной системе; *6* — начальный макет системы; *7* — следующий уровень макета; *8* — сконструированная система; *9* — оценивание заказчиком

Модель определяет четыре действия, представляемые четырьмя квадрантами спирали.

1. Планирование — определение целей, вариантов и ограничений.
2. Анализ риска — анализ вариантов и распознавание/выбор риска.
3. Конструирование — разработка продукта следующего уровня.
4. Оценивание — оценка заказчиком текущих результатов конструирования.

Интегрирующий аспект спиральной модели очевиден при учете радиального измерения спирали. С каждой итерацией по спирали (продвижением от центра к периферии) строятся все более полные версии ПО.

В первом витке спирали определяются начальные цели, варианты и ограничения, распознается и анализируется риск. Если анализ риска показывает неопределенность требований, на помощь разработчику и заказчику приходит макетирование (используемое в квадранте конструирования). Для дальнейшего определения проблемных и уточненных требований может быть использовано моделирование.

Заказчик оценивает инженерную (конструкторскую) работу и вносит предложения по модификации (квадрант оценки заказчиком). Следующая фаза планирования и анализа риска базируется на предложениях заказчика. В каждом цикле по спирали результаты анализа риска формируются в виде «продолжать, не продолжать». Если риск слишком велик, проект может быть остановлен.

В большинстве случаев движение по спирали продолжается, с каждым шагом продвигая разработчиков к более общей модели системы. В каждом цикле по спирали требуется конструирование (нижний правый квадрант), которое может быть реализовано классическим жизненным циклом или макетированием. Заметим, что количество действий по разработке (происходящих в правом нижнем квадранте) возрастает по мере продвижения от центра спирали.

Достоинства спиральной модели:

1. наиболее реально (в виде эволюции) отображает разработку программного обеспечения;
2. позволяет явно учитывать риск на каждом витке эволюции разработки;
3. включает шаг системного подхода в итерационную структуру разработки;
4. использует моделирование для уменьшения риска и совершенствования программного изделия.

Недостатки спиральной модели:

1. новизна (отсутствует достаточная статистика эффективности модели);
2. повышенные требования к заказчику;
3. трудности контроля и управления временем разработки.

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. ниже).

Рис. Компонентно-ориентированная модель

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.

Тяжеловесные и облегченные процессы

В XXI веке потребности общества в программном обеспечении информационных технологий достигли экстремальных значений. Программная индустрия буквально «захлебывается» от потока самых разнообразных заказов. «Больше процессов разработки, хороших и разных!» — скандируют заказчики. «Сейчас, сейчас! Только об этом и думаем!» — отвечают разработчики.

Традиционно для упорядочения и ускорения программных разработок предлагались строго упорядочивающие тяжеловесные (**heavyweight**) процессы. В этих процессах прогнозируется весь объем предстоящих работ, поэтому они называются прогнозирующими (**predictive**) процессами. Порядок, который должен выполнять при этом человек-разработчик, чрезвычайно строг — «шаг вправо, шаг влево — виртуальный расстрел!». Иными словами, человеческие слабости в расчет не принимаются, а объем необходимой документации способен отнять покой и сон у «совестливого» разработчика.

В последние годы появилась группа новых, облегченных (**lightweight**) процессов. Теперь их называют подвижными (**agile**) процессами. Они привлекательны отсутствием бюрократизма, характерного для тяжеловесных (прогнозирующих) процессов. Новые процессы должны воплотить в жизнь разумный компромисс между слишком строгой дисциплиной и полным ее отсутствием. Иначе говоря, порядка в них достаточно для того, чтобы получить разумную отдачу от разработчиков.

Подвижные процессы требуют меньшего объема документации и ориентированы на человека. В них явно указано на необходимость использова-ния природных качеств человеческой натуры (а не на применение действий, направленных напере-кор этим качествам).Более того, подвижные процессы учитывают особенности современного заказчика, а именно частые изменения его требо-ваний к программному продукту. Известно, что для прогнозирующих процессов частые изменения требований подобны смерти. В отличие от них, подвижные процессы адаптируют изменения требований и даже выигрывают от этого. Словом, подвижные процессы имеют адаптивную природу.

Таким образом, в современной инфраструктуре программной инженерии существуют два семейства процессов разработки:

- семейство прогнозирующих (тяжеловесных) процессов;
- семейство адаптивных (подвижных, облегченных) процессов.

У каждого семейства есть свои достоинства, недостатки и область применения:

- адаптивный процесс используют при частых изменениях требований, малочисленной группе высококвалифицированных разработчиков и грамотном заказчике, который согласен участвовать в разработке;
- прогнозирующий процесс применяют при фиксированных требованиях и многочисленной группе разработчиков разной квалификации.

XP-процесс

Экстремальное программирование (e**X**treme **P**rogramming, **XP**) — облегченный (подвижный) процесс (или методология), главный автор которого — Кент Бек (1999).

XP - процесс ориентирован на группы малого и среднего размера, строящие программное обеспечение в условиях неопределенных или быстро изменяющихся требований. XP - группу образуют до 10 сотрудников, которые размещаются в одном помещении.

Основная идея XP — устранить высокую стоимость изменения, характерную для приложений с использованием объектов, паттернов (*Паттерн является решением типичной проблемы в определенном контексте)* и реляционных баз данных.

Поэтому XP - процесс должен быть высокодинамичным процессом. XP - группа имеет дело с изменениями требований на всем протяжении итерационного цикла разработки, причем цикл состоит из очень коротких итераций.

Четырьмя базовыми действиями в XP - цикле являются: **кодирование, тестирование, выслушивание заказчика и проектирование**. Динамизм обеспечивается с помощью четырех характеристик: непрерывной связи с заказчиком (и в пределах группы), простоты (всегда выбирается минимальное решение), быстрой обратной связи (с помощью модульного и функционального тестирования), смелости в проведении профилактики возможных проблем.

Большинство принципов, поддерживаемых в XP (**минимальность, простота, эволюционный цикл разработки, малая длительность итерации, участие пользователя, оптимальные стандарты кодирования и т. д.**), продиктованы здравым смыслом и применяются в любом упорядоченном процессе. Просто в XP эти принципы, как показано в таблице, достигают «экстремальных значений».

Таблица. Экстремумы в экстремальном программировании

Практика здравого смысла	**XP - экстремум**	**XP - реализация**
Проверки кода	Код проверяется все время	Парное программирование
Тестирование	Тестирование выполняется все время, даже с помощью заказчиков	Тестирование модуля, функциональное тестирование

Проектирование	Проектирование является частью ежедневной деятельности каждого разработчика	Реорганизация (refactoring)
Простота	Для системы выбирается простейшее проектное решение, поддерживающее ее текущую функциональность	Самая простая вещь, которая могла бы работать

Архитектура	Каждый постоянно работает над уточнением архитектуры	Метафора
Тестирование интеграции	Интегрируется и тестируется несколько раз в день	Непрерывная интеграция
Короткие итерации	Итерации являются предельно короткими, продолжаются секунды, минуты, часы, а не недели, месяцы или годы	Игра планирования

Базис XP образуют перечисленные ниже двенадцать методов.

1. Игра планирования (Planning game) — быстрое определение области действия следующей реализации путем объединения деловых приоритетов и технических оценок. Заказчик формирует область действия, приоритетность и сроки с точки зрения бизнеса, а разработчики оценивают и прослеживают продвижение (прогресс).
2. Частая смена версий (Small releases) — быстрый запуск в производство простой системы. Новые версии реализуются в очень коротком (двухнедельном) цикле.

3. Метафора (Metaphor) — вся разработка проводится на основе простой, общедоступной истории о том, как работает вся система.
4. Простое проектирование (Simple design) — проектирование выполняется настолько просто, насколько это возможно в данный момент.
5. Тестирование (Testing) — непрерывное написание тестов для модулей, которые должны выполняться безупречно; заказчики пишут тесты для демонстрации законченности функций. «Тестируй, а затем кодируй» означает, что входным критерием для написания кода является «отказавший» тестовый вариант.

6. Реорганизация (Refactoring) — система реструктурируется, но ее поведение не изменяется; цель — устранить дублирование, улучшить взаимодействие, упростить систему или добавить в нее гибкость.
7. Парное программирование (Pair programming) — весь код пишется двумя программистами, работающими на одном компьютере.
8. Коллективное владение кодом (Collective ownership) — любой разработчик может улучшать любой код системы в любое время.

9. Непрерывная интеграция (Continuous integration) — система интегрируется и строится много раз в день, по мере завершения каждой задачи. Непрерывное регрессионное тестирование, то есть повторение предыдущих тестов, гарантирует, что изменения требований не приведут к регрессу функциональности.
10. 40-часовая неделя (40-hour week) — как правило, работают не более 40 часов в неделю. Нельзя удваивать рабочую неделю за счет сверхурочных работ.

1. Локальный заказчик (On-site customer) — в группе все время должен находиться представитель заказчика, действительно готовый отвечать на вопросы разработчиков.
2. Стандарты кодирования (Coding standards) — должны выдерживаться правила, обеспечивающие одинаковое представление программного кода во всех частях программной системы.

Игра планирования и частая смена версий зависят от заказчика, обеспечивающего набор «историй» (коротких описаний), характеризующих работу, которая будет выполняться для каждой версии системы. Версии генерируются каждые две недели, поэтому разработчики и заказчик должны прийти к соглашению о том, какие истории будут осуществлены в пределах двух недель. Полную функциональность, требуемую заказчику, характеризует пул историй; но для следующей двухнедельной итерации из пула выбирается подмножество историй, наиболее важное для заказчика.

В любое время в пул могут быть добавлены новые истории, таким образом, требования могут быстро изменяться. Однако процессы двухнедельной генерации основаны на наиболее важных функциях, входящих в текущий пул, следовательно, изменчивость управляется. Локальный заказчик обеспечивает поддержку этого стиля итерационной разработки.

«Метафора» обеспечивает, глобальное «видение» проекта. Она могла бы рассматриваться как высокоуровневая архитектура, но XP подчеркивает желательность проектирования при минимизации проектной документации. Точнее говоря, XP предлагает непрерывное перепроектирование (с помощью реорганизации), при котором нет нужды в детализированной проектной документации, а для инженеров сопровождения единственным надежным источником информации является программный код. Обычно после написания кода проектная документация выбрасывается.

Проектная документация сохраняется только в том случае, когда заказчик временно теряет способность придумывать новые истории. Тогда систему помещают в «нафталин» и пишут руководство страниц на пять-десять по «нафталиновому» варианту системы. Использование реорганизации приводит к реализации простейшего решения, удовлетворяющего текущую потребность. Изменения в требованиях заставляют отказываться от всех «общих решений».

Парное программирование — один из наиболее спорных методов в XP, оно влияет на ресурсы, что важно для менеджеров, решающих, будет ли проект использовать XP. Может показаться, что парное программирование удваивает ресурсы, но исследования доказали: парное программирование приводит к повышению качества и уменьшению времени цикла. Для согласованной группы затраты увеличиваются на 15%, а время цикла сокращается на 40-50%. Для Интернет-среды увеличение скорости продаж покрывает повышение затрат.

Сотрудничество улучшает процесс решения проблем, улучшение качества существенно снижает затраты сопровождения, которые превышают стоимость дополнительных ресурсов по всему циклу разработки.

Коллективное владение означает, что любой разработчик может изменять любой фрагмент кода системы в любое время. Непрерывная интеграция, непрерывное регрессионное тестирование и парное программирование XP обеспечивают защиту от возникающих при этом проблем.

«Тестируй, а затем кодируй» — эта фраза выражает акцент XP на тестировании. Она отражает принцип, по которому сначала планируется тестирование, а тестовые варианты разрабатываются параллельно анализу требований, хотя традиционный подход состоит в тестировании «Черного ящика». Размышление о тестировании в начале цикла жизни — хорошо известная практика конструирования ПО (правда, редко осуществляемая практически).

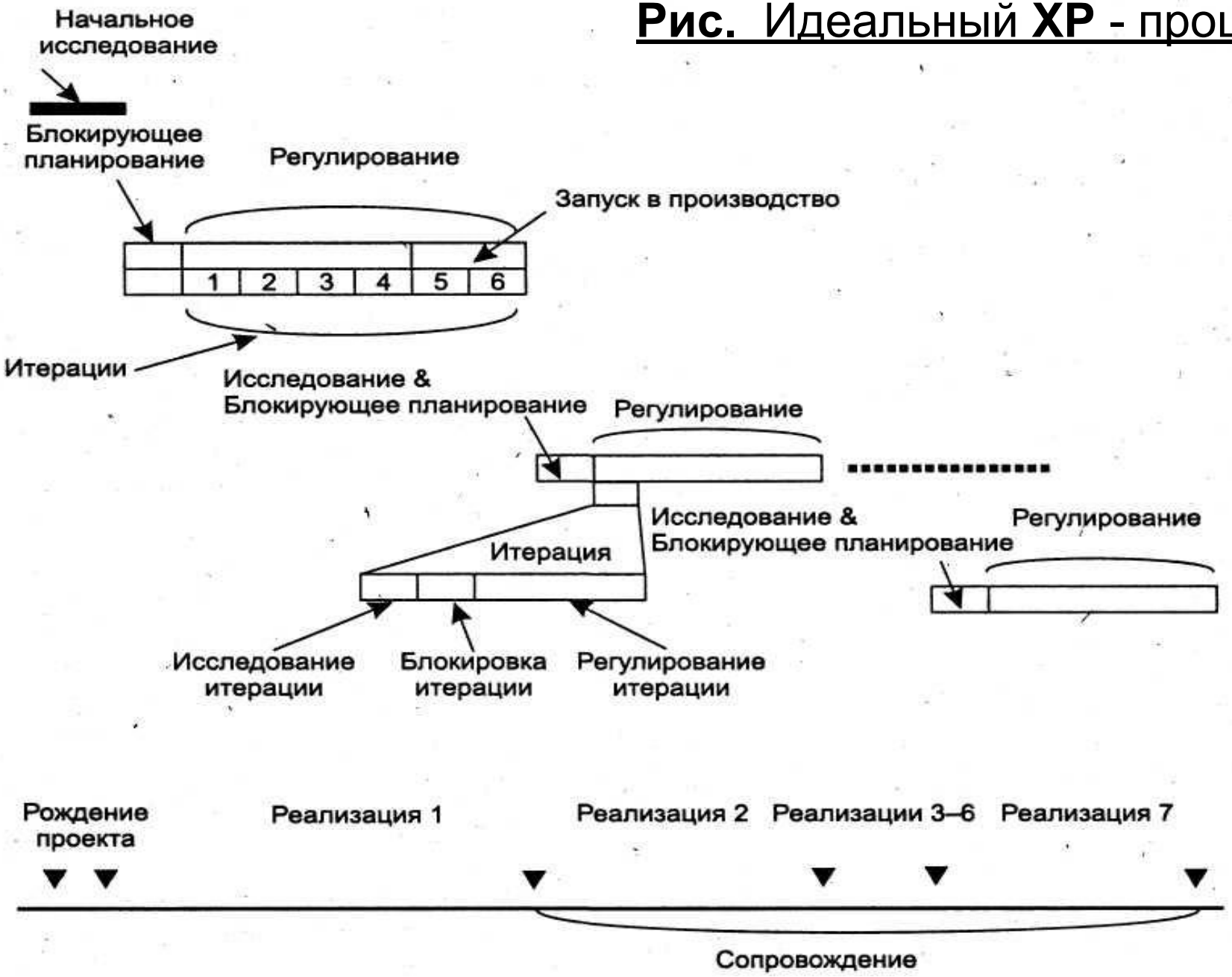
Основным средствам управления **XP** является метрика, а среда метрик — «большая визуальная диаграмма». Обычно используют 3-4 метрики, причем такие, которые видимы всей группе. Рекомендуемой в **XP** метрикой является «скорость проекта» — количество историй заданного размера, которые могут быть реализованы в итерации.

При принятии **XP** рекомендуется осваивать его методы по одному, каждый раз выбирая метод, ориентированный на самую трудную проблему группы. Конечно, все эти методы являются «не более чем правилами» — группа может в любой момент поменять их (если ее сотрудники достигли принципиального соглашения по поводу внесенных изменений). Защитники **XP** признают, что **XP** оказывает сильное социальное воздействие, и не каждый может принять его. Вместе с тем, **XP** — это методология, обеспечивающая преимущества только при использовании законченного набора базовых методов.

Рассмотрим структуру «идеального» **XP**-процесса. Основным структурным элементом процесса является **XP**-реализация, в который многократно вкладывается базовый элемент — **XP**-итерация. В состав **XP**-реализации и **XP**-итерации входят три фазы — исследование, блокировка, регулирование. Исследование (exploration) — это поиск новых требований (историй, задач), которые должна выполнять система. Блокировка (commitment) — выбор для реализации конкретного подмножества из всех возможных требований (иными словами, планирование). Регулирование (steering) — проведение разработки, воплощение плана в жизнь.

XP рекомендует: первая реализация должна иметь длительность 2 – 6 месяцев, продолжительность остальных реализаций — около двух месяцев, каждая итерация длится приблизительно две недели, а численность группы разработчиков не превышает 10 человек. **XP** – процесс для проекта с семью реализациями, осуществляемый за 15 месяцев, показан на рис. ниже.

Рис. Идеальный **XP** - процесс



Процесс инициируется начальной исследовательской фазой.
Фаза исследования, с которой начинается любая реализация и итерация, имеет клапан «пропуска», на этой фазе принимается решение о целесообразности дальнейшего продолжения работы.

Предполагается, что длительность первой реализации составляет 3 месяца, длительность второй — седьмой реализаций — 2 месяца. Вторая — седьмая реализации образуют период сопровождения, характеризующий природу **XP**-проекта. Каждая итерация длится две недели, за исключением тех, которые относят к поздней стадии реализации — «запуску в производство» (в это время темп итерации ускоряется).

Наиболее трудна первая реализация — пройти за три месяца от обычного старта (скажем, отдельный сотрудник не зафиксировал никаких требований, не определены ограничения) к поставке заказчику системы промышленного качества очень сложно.

The End