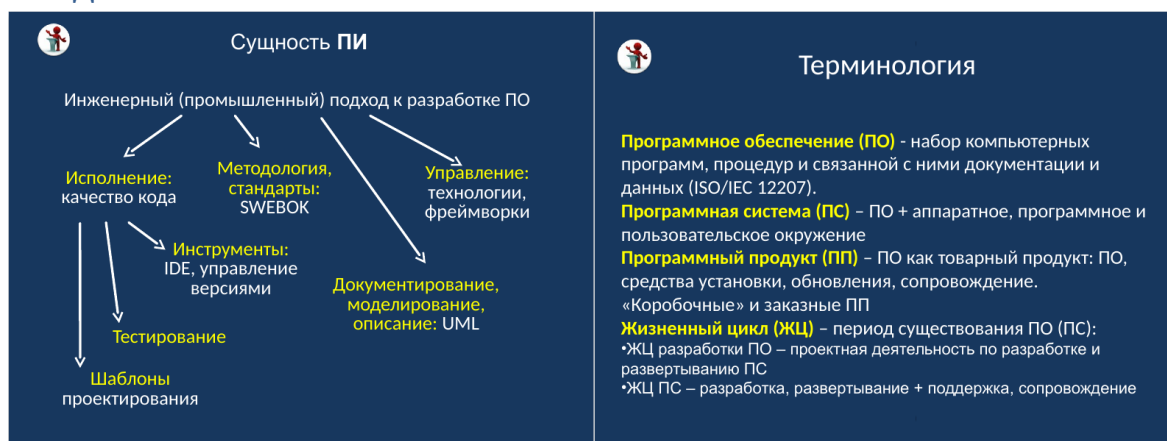






Оглавление

1. Сущность программной инженерии (ПИ). Связь с computer science. Особенности в сравнении и другими инженерными дисциплинами. Свод знаний и ПИ SWEBOK.....	2
2. Жизненный цикл (ЖЦ) программного продукта и проекта. «Легкие» и «тяжелые» модели процессов разработки ПО. Этапы и технологические процессы (дисциплины) ЖЦ. Результаты этапов и основные документы. Каскадная, итеративная и спиральная модели.....	4
3. Унифицированный процесс UP. Фазы жизненного цикла: исследование, анализ, реализация, внедрение. Содержание и результаты фаз. Итерация и ее рабочие потоки: требования, анализ, проектирование, реализация, тестирование, их содержание.	10
4) Фаза исследования. Основные дисциплины и артефакты. Дисциплина «анализ предметной области», бизнес-анализ. Диаграммы потоков данных, деятельности. Моделирование предметной области.	12
5. Фаза анализа и проектирования. Дисциплина «анализ требований». Способы извлечения и фильтрации требований. Бизнес-требования, бизнес-требования, системные требования, функциональные требования. Разработка и управление требованиями. Документ «спецификация требований к ПО ». Диаграммы прецедентов.....	13
6. Фаза анализа и проектирования. Понятие архитектуры, ее многомерность. Основные методы проектирования и их особенности: структурное, функциональное, объектно-ориентированное, компонентное, проектирование на основе структур данных. Классы анализа. Виды классов: граница, управление, сущность. Диаграммы устойчивости. Архитектурные аспекты технологического процесса проектирования (по SWEBOK)	17
7. Фаза анализа и проектирования. Дисциплина проектирование (design). Ключевые моменты проектирования по SWEBOK: параллелизм, контроль и обработка событий, распределение компонентов, обработка ошибок и исключительных ситуаций и обеспечение отказоустойчивости, взаимодействие и представление (MVC), сохраняемость данных (доступность «долгоживущих» данных).....	21
8) Фаза анализа и проектирования. Многоуровневая архитектура клиент-серверных приложений. Тонкие и толстые клиенты. Локальное и сетевое взаимодействие слоев через интерфейсы и протоколы. Совместное использование кода различными типами клиентов.....	22
9. Фаза анализа и проектирования Проектирование графического интерфейса (GUI). Основные аспекты. Архитектурное проектирование, основанное на GUI. Факторы, характеризующие GUI: производительность, человеческие ошибки, обучение, субъективное восприятие, запоминание, поиск, визуализация, навигация	23
10. Виды моделей. Сущность UML как средства моделирования. Структура UML, статическая и динамическая составляющие модели. Составные элементы: сущности, отношения, диаграммы. Виды сущностей: структурные сущности – класс, интерфейс, кооперация, прецедент, активный класс, компонент, узел; поведенческие сущности – взаимодействия, деятельности, автоматы, пакет, примечание.	24
11.UML. Виды отношений: зависимость, ассоциация, агрегация, композиция, включение, обобщение, реализация. Отношения. Связи – отношения между объектами. Направленность	

связи, Сообщения. Диаграммы объектов. Ассоциации – отношения между классами. Свойства ассоциации: имя, кратность, навигация, атрибуты. Рефлексивные ассоциации, деревья и сети. Классы атрибутов ассоциаций (классы-ассоциации). Зависимости. Зависимости использования «use», «call», «parameter», «send» и «instantiate». Зависимости абстракции. Зависимости доступа.	27
12. UML. Принятые деления: классификатор-экземпляр, интерфейс-реализация. Расширения: ограничения, стереотипы. Классификация диаграмм. Диаграммы классов (объектов). Диаграммы взаимодействий, коммуникационные диаграммы.	31
13. UML. Диаграммы деятельности. Технология сетей Петри. Параллелизм. Поток управления, узел действия, ребро, узел управления, объектный узел, буферизация и в объектном узле. Объектные узлы – параметры, состояния объектных узлов. Контакты. Прерывающие ребра. Контакты исключений. Потоки объектов. Их аналоги в программировании. Диаграммы состояний. Конечные автоматы.	33
14. Экстремальное и гибкое программирование. Манифест экстремального программирования (XP). Гибкие (agile) технологии. SCRUM. Agile UP, ICONIX.	38
15. SCRUM как технологический фреймворк.. Терминология. Спринт. Митинг. Собственник проекта. Команда. SCRUM-мастер. Беклог проекта и спринта. Планирование спринта. Диаграмма сгорания. Оценка трудоемкости. Покер-планирование.	42
16) Оценка программного кода. Метрики кода. Метрики количественные, сложности потока управления и потока данных, метрики ООП, прагматические метрики. Средства оценки качества программного кода.	44

1. Сущность программной инженерии (ПИ). Связь с computer science. Особенности в сравнении и другими инженерными дисциплинами. Свод знаний и ПИ SWEBOOK



 <h3>ПИ и информатика (computer science)</h3> <p>Computer science – теоретический фундамент - знания, модели и методы организации вычислительных процессов:</p> <ul style="list-style-type: none"> •теория информации •теория алгоритмов (алгоритмическая разрешимость) •структуры данных и алгоритмы, дискретная математика, теория графов и т.п. •наукоемкое ПО - мат. основы предметной области (обработка сигналов, криптография) •трудоемкость и эффективность алгоритмов •формальные языки и трансляторы <p>Знания computer science обеспечивают качество ПО, но не гарантируют успех его разработки (необходимое, но не достаточное условие). Фрагментарный характер использования computer science в проектировании</p>	 <h3>ПИ и управление программными проектами</h3> <p>Управление проектами (PMI):</p> <ul style="list-style-type: none"> •оценка рисков •организация команды •метрика (сроки исполнения, стоимость, объем работ) •планирование <p>Управление программными проектами – специфика:</p> <ul style="list-style-type: none"> •специфические риски, связанные с качеством персонала, оценками трудозатрат, адекватностью представлений •различные принципы организации и самоорганизации исполнителей •метрика программного продукта, метрика качества кода •особенности и структура ЖЦ ПО
 <h3>Сравнение с инж. подходом в других отраслях</h3> <p>Нематериальный характер продукта:</p> <ul style="list-style-type: none"> • 0% затрат на тиражирование (аналогия с интеллектуальным продуктом) •отсутствие аналогов изделия в окружающем мире – инкрементальный подход: программный прототип •отсутствие «охватывающих» объективных законов и невозможность теоретического расчета и проверки – основное отличие ремесла от науки. Работоспособность по проверяется тестированием готового продукта <p><small>«...проект архитектуры не является чертежом. Все подробности программной системы описываются только кодом на языке высокого уровня, который, таким образом, является чертежом программы. А поскольку все операции, ведущие к созданию чертежа, являются проектированием, то и вся разработка ПО должна считаться проектированием» <i>Кони Бюрер. От ремесла к науке: поиск основных принципов разработки ПО</i></small></p>	 <h3>Сравнение с инж. подходом в других отраслях</h3> <p>Общее:</p> <ul style="list-style-type: none"> •Получение продукта с гарантированным функционалом, качеством и др. свойствами, сроками исполнения и в рамках отработанного технологического процесса; •Управление проектом (менеджмент) [PMBOK (Совокупность знаний по управлению проектами)] <p>Особенное:</p> <ul style="list-style-type: none"> •Основанием программной инженерии является информатика, а не естественные науки. •Основной упор делается на дискретной, а не на непрерывной математике. •Концентрация на абстрактных/логических объектах вместо конкретных/физических артефактов. •Отсутствие «производственной» фазы в традиционном промышленном смысле (проектирование - изготовление). •«Сопровождение» программного обеспечения в основном связано с продолжающейся разработкой или эволюцией, а не с традиционным физическим износом.

Свод знаний о программной инженерии SWEBOOK.

Основной документ SWEBOOK — управление программной инженерией, разделами которого являются:

- инициирование и определение содержания
- планирования программного проекта
- выполнение программного проекта
- обзор и оценка
- закрытие проекта
- измерения в программной инженерии

Связанные документы SWEBOOK:

- требования к программному обеспечению (Software Requirements)
- конфигурационное управление (Software Configuration Management)
- процесс программной инженерии (Software engineering Process)
- качество (Software Quality)

Структура и содержание SWEBOOK

SWEBOOK описывает 10 **областей знаний**:

- Software requirements – программные требования;
- Software design – дизайн (архитектура);

- Software construction – конструирование ПО;
- Software testing – тестирование;
- Software maintenance – эксплуатация (поддержка) ПО;
- Software configuration management – конфигурационное управление;
- Software engineering management – управление в программной инженерии;
- Software engineering process – процессы программной инженерии;
- Software engineering tools and methods – инструменты и методы;
- Software quality – качество ПО.

В дополнение к ним, SWEBOOK также включает обзор **смежных дисциплин**, связь с которыми представлена как фундаментальная, важная и обоснованная для программной инженерии:

- Computer engineering;
- Computer science;
- Management;
- Mathematics;
- Project management;
- Quality management;
- System engineering.

Стоит отметить, что принятые разграничения между областями знаний, их компонентами (subareas) и другими элементами достаточно произвольны. При этом, в отличие от PMBOK, области знаний SWEBOOK не включает «входы» и «выходы».

Для каждой подобласти знаний SWEBOOK описывает ключевые акронимы, представляет область в виде «подобластей» (subareas) или (по определению SWEBOOK) «секций». Декомпозиция каждой секции в форме списка тем (topics) с их описанием.

2. Жизненный цикл (ЖЦ) программного продукта и проекта. «Легкие» и «тяжелые» модели процессов разработки ПО. Этапы и технологические процессы (дисциплины) ЖЦ. Результаты этапов и основные документы. Каскадная, итеративная и спиральная модели.

Про Жизненный цикл -----

Жизненный цикл (ЖЦ) – период существования ПО (ПС):

- ЖЦ разработки ПО – проектная деятельность по разработке и развертыванию ПС
- ЖЦ ПС – разработка, развертывание + поддержка, сопровождение

Жизненный цикл системы представляет собой эволюцию (прогресс, развитие) системы через смену ее состояний от первичного замысла до прекращения её существования. Использование этого понятия (усмотрение в системе эволюции через набор состояний от замысла до забвения) означает применение к системе *подхода жизненного цикла (подхода с точки зрения жизненного цикла, жизненнотциклового подхода)*.

Подход жизненного цикла – *подход, методы (онтология и нотация описания) которого требуют при рассмотрении и описании системы рассматривать и описывать её как проходящую во времени через ряд состояний, от появления первичного замысла и до прекращения существования, и принимать любые решения по поводу изменения системы с учётом их последствий на протяжении всего времени её существования.*

В рамках подхода жизненного цикла вводится понятие стадии жизненного цикла, как определённого периода в развитии (эволюции) системы, отличающегося более-менее стабильным состоянием системы, нахождением системы в какой-то определенной форме (замысла, проекта, собираемой конструкции, заказанного у различных поставщиков оборудования, эксплуатируемого объекта и т.д.).

Понимаемые таким образом стадии не обязаны следовать строго друг за другом, они могут накладываться во времени (например, проект здания существует не только до строительства, но и во время него). Однако начало и конец каждой стадии жизненного цикла должны быть отмечены принятием полномочными стейкхолдерами системы специального решения. Конкретный состав стадий и их связи определяются спецификой как целевой системы, так и используемого процессного подхода.

Формально: жизненный цикл – это смена состояний системы (эволюция системы) в период времени от замысла до прекращения её существования.

Комментарий: жизненный цикл – всегда жизненный цикл конкретной системы. Не бывает «жизненного цикла» кроме как в текстах стандартов, в жизни всегда «жизненный цикл Х», где Х – название целевой системы. Процессы жизненного цикла – это те процессы, которые акторы выполняют над/с системой, и которые меняют состояние системы, заставляя ее эволюционировать в ходе её жизненного цикла. «Управление жизненным циклом» -- общепринятое название подхода к описанию процессов жизненного цикла (а часто и название самой группы процессов жизненного цикла, описанных с использованием такого подхода).

Про Легкие и Тяжелые модели процессов разработки ПО-----

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 [29]. Теперь их называют подвижными (agile) процессами [8], [25], [36]. 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 использования природных качеств человеческой натуры (а не на применение действий, направленных наперекор этим качествам).

Более того, подвижные процессы учитывают особенности современного заказчика, а именно частые изменения его требований 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.

Пример облегченного процесса – XP-программирование(экстремальное программирование)

Про этапы и технологические процессы (дисциплины) ЖЦ-----

Описание процессов ЖЦ в «широком» смысле -

ГОСТ Р ИСО/МЭК 12207-99 «Процессы жизненного цикла программных средств»

Основные процессы жизненного цикла:

заказ, поставка, разработка, эксплуатация, сопровождение

Вспомогательные процессы жизненного цикла:

документирование, управления конфигурацией, обеспечение качества, верификация, аттестация, совместный анализ, аудит, решение проблем

Организационные процессы жизненного цикла:

управление, создание инфраструктуры, усовершенствование, обучение

Описание процессов ЖЦ в «узком» (технологическом) смысле – специфическое профессиональное содержание процессов разработки ПО –

SWEBOK - *Software Engineering Body of Knowledge*,
IEEE 2004 Version - Свод Знаний по ПИ:

- *Software requirements* – программные требования
- *Software design* – дизайн (архитектура)
- *Software construction* – конструирование ПО
- *Software testing* - тестирование
- *Software maintenance* – эксплуатация (поддержка) ПО
- *Software configuration management* – конфигурационное управление
- *Software engineering management* – управление в ПИ
- *Software engineering process* – процессы ПИ
- *Software engineering tools and methods* – инструменты и методы
- *Software quality* – качество ПО

Технологические процессы ЖЦ (из RUP):

- Моделирование предметной области;
- Управление требованиями;
- Анализ и проектирование;
- Реализация (конструирование)
- Тестирование;
- Распространение ПО (установка, обучение);

Вспомогательные:

- Управление проектом (организация, менеджмент);
- Управление конфигурацией (сопровождение);
- Управление средой разработки;

Про модели-----

Модель жизненного цикла (life cycle model): структура, состоящая из процессов, работ и задач, включающих в себя разработку, эксплуатацию и сопровождение программного продукта, охватывающая жизнь системы от установления требований к ней до прекращения ее использования.

- Каскадная
- Итеративная / инкрементальная
- Спиральная

Каскадная- Данная модель предполагает строго последовательное (во времени) и однократное выполнение всех фаз проекта с жестким (детальным) предварительным планированием в контексте predetermined или однажды и целиком определенных требований к программной системе.



Каскадная модель

Однократное последовательное выполнение всех фаз (технологических процессов)



Принятое решение на определенном этапе не может быть отменено по результатам последующих этапов, все ошибки исправляются тестированием (???)

Итеративная- Итеративная модель предполагает разбиение жизненного цикла проекта на последовательность итераций, каждая из которых напоминает “мини-проект”, включая все фазы жизненного цикла в применении к созданию меньших фрагментов функциональности, по сравнению с проектом, в целом. Цель каждой итерации – получение работающей версии программной системы, включающей функциональность, определенную интегрированным содержанием всех предыдущих и текущей итерации. Результата финальной итерации содержит всю требуемую функциональность продукта. Таким образом, с завершением каждой итерации, продукт развивается инкрементально.



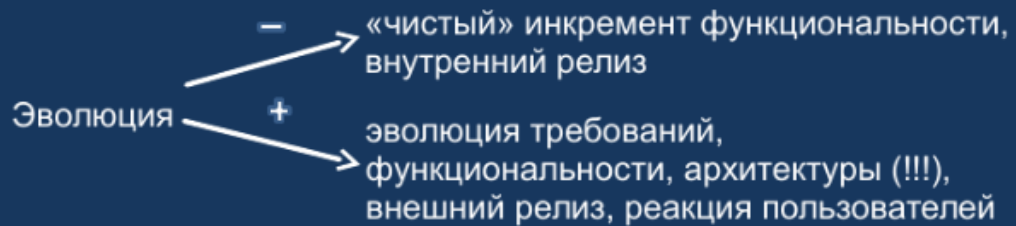
Итерация и инкремент

Итерация – последовательное выполнение одинаковых фаз во времени

Инкремент - добавление функциональности

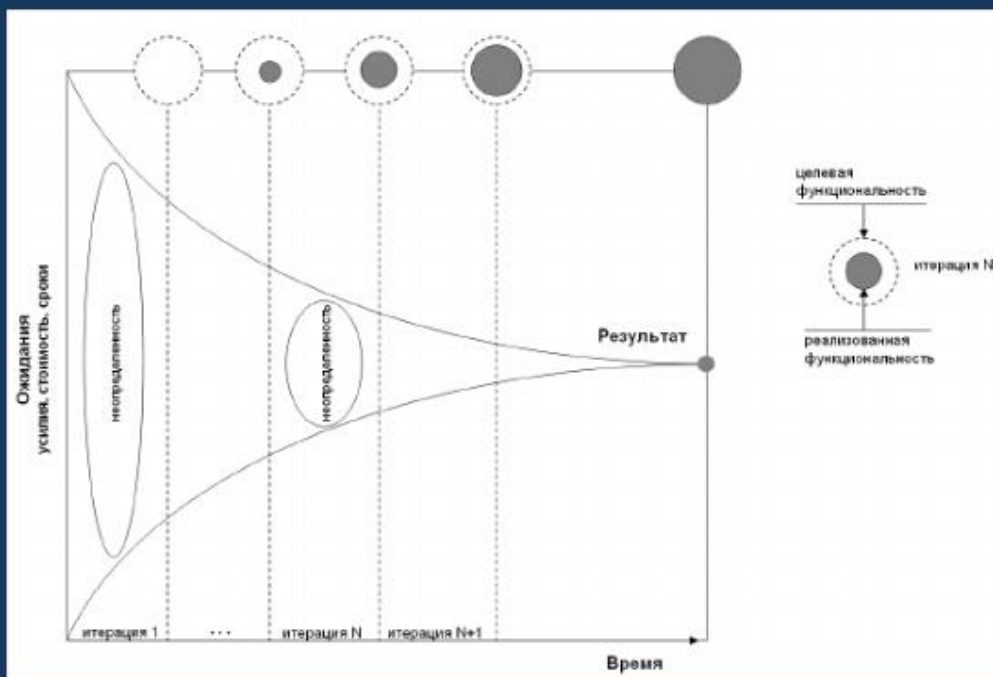
Каждая итерация (шаг) включает в себя все технологические процессы (в разном % и качественном отношении)

Итерация и эволюция





Итерация и инкремент



Снижение неопределенности и инкрементальное расширение функциональности при итеративном подходе

Спиральная - была впервые сформулирована Барри Боэмом (Barry Boehm) в 1988 году [Boehm, 1988]. Отличительной особенностью этой модели является специальное внимание рискам, влияющим на организацию жизненного цикла.



Спиральная модель

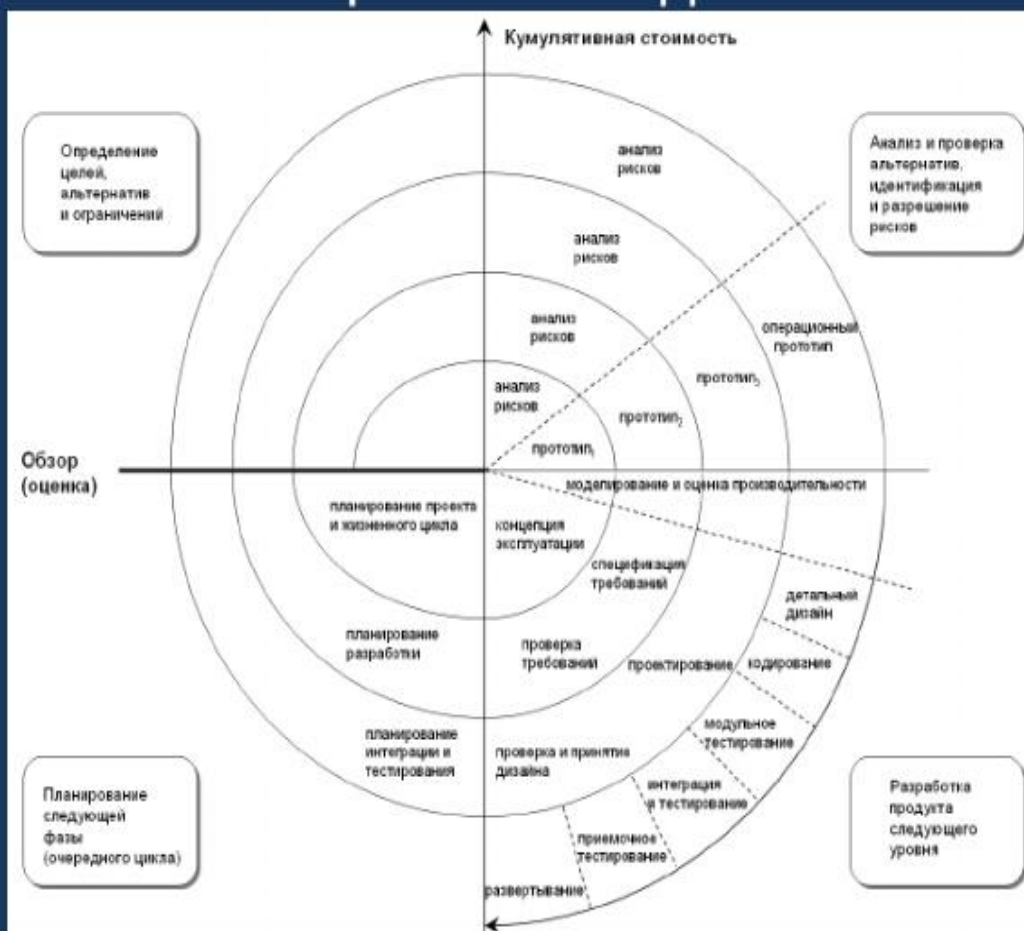
Эволюционная модель с анализом и оценкой рисков на каждой итерации [Boehm, 1988]. TOP-10 рисков по Боэму:

1. Дефицит специалистов
2. Нереалистичные сроки и бюджет
3. Реализация несоответствующей функциональности
4. Разработка неправильного пользовательского интерфейса
5. "Золотая сервировка", перфекционизм, ненужная оптимизация и оттачивание деталей
6. Непрерывающийся поток изменений
7. Нехватка информации о внешних компонентах, определяющих окружение системы или вовлеченных в интеграцию
8. Недостатки в работах, выполняемых внешними (по отношению к проекту) ресурсами
9. Недостаточная производительность получаемой системы
10. "Разрыв" в квалификации специалистов разных областей знаний

Большая часть этих рисков связана с организационными и процессными аспектами взаимодействия специалистов в проектной команде.



Спиральная модель



3. Унифицированный процесс UP. Фазы жизненного цикла: исследование, анализ, реализация, внедрение. Содержание и результаты фаз. Итерация и ее рабочие потоки: требования, анализ, проектирование, реализация, тестирование, их содержание.

Унифицированный процесс UP - фреймворк для построения процессов разработки программного обеспечения, позволяющий команде разработки преобразовывать требования заказчика в работоспособный продукт. Примером методологии разработки, основанной на Unified Process, является Rational Unified Process (RUP). Unified Process активно использует унифицированный язык моделирования (UML). В ядре UML лежит модель, которая позволяет команде разработке в упрощенном виде понять многообразие сложных процессов, необходимых для разработки программного обеспечения. Различные модели, используемые в Unified Process, позволяют визуализировать систему, описать её структуру и поведение, задокументировать принимаемые в процессе разработки решения.

Фазы ЖЦ:

- 1: **Исследование/Идея** – фаза, на которой определяется видение проекта и принимаются все основные решения, касающиеся реализации проекта
- 2: **Уточнение плана/Инициализация** – фаза конкретизации видения, выделенного на предыдущем шаге и построение некоторого архитектурного прототипа

3: **Построение/Реализация** – реализация некоторой демонстрационной версии проекта, содержащей базовые возможности

4: **Развёртывание/Завершение** – завершение разработки и доведение проекта до уровня дальнейшей эксплуатации

Результаты этапов и основные документы.

Начальный этап — идея, которая может представлять собой:

- потребность
- возможность
- проблему

Фаза 1: **Инициализация** (определить что и зачем делать).

Выходные документы: Концепция;

Фаза 2: **Планирование** (определить как делать).

Требования к системе, план управления

Фаза 3: **Реализация** (создать документированный и протестированный программный продукт).

Рабочее расписание, отчёт о состоянии, документы проекта, исходные коды

Фаза 4: **Завершение** (проверить соответствие концепции и проведение приёмо-сдаточных испытаний (ПСИ) продукта на соответствие свойств требованиям).

Протоколы и акт ПСИ, итоговый отчёт, архив проекта

Результаты:

- Удовлетворение потребности
- Реализация возможности
- Решение проблемы

Итерация и её структура: требования, анализ, проектирование, реализация, тестирование.

Каждый этап процесса разделяется на итерации. Итерация — это полный цикл разработки, вырабатывающий промежуточный продукт. По мере перехода от итерации к итерации промежуточный продукт инкрементно усложняется, постепенно превращаясь в конечную систему. В состав каждой итерации входят все рабочие потоки — от сбора требований до тестирования.

Каждый этап и итерация уменьшают некоторый риск и завершается контрольной вехой. К вехе привязывается техническая проверка степени достижения ключевых целей. По результатам проверки возможна модификация дальнейших действий.

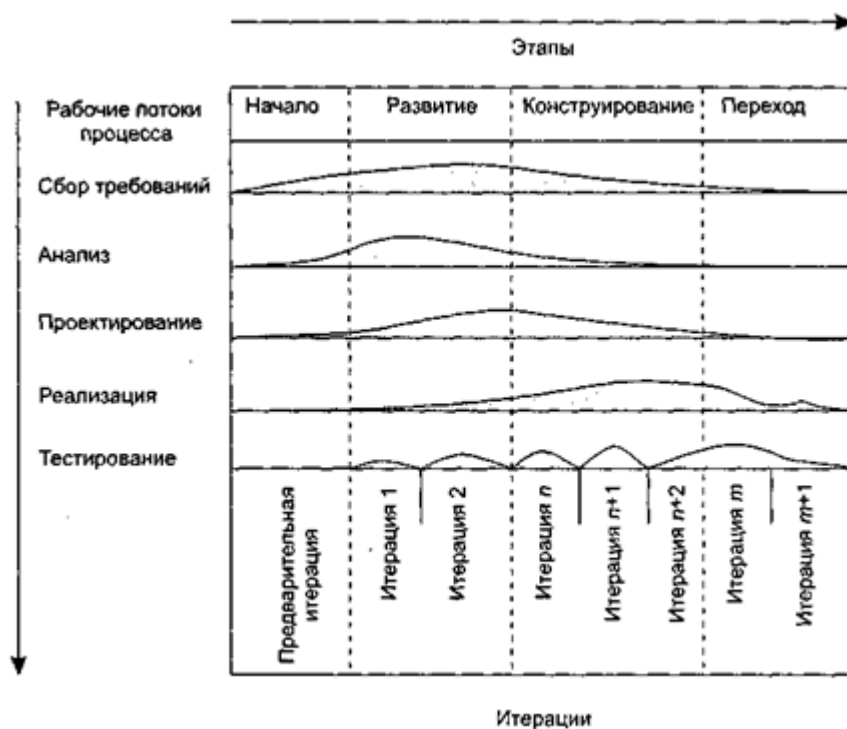


Рис. 15.2. Два измерения унифицированного процесса разработки (RUP)

Как показано на рис. 15.2, в структуре унифицированного процесса разработки выделяют два измерения:

- q горизонтальная ось представляет время и демонстрирует характеристики жизненного цикла процесса;
- q вертикальная ось представляет рабочие потоки процесса, которые являются логическими группировками действий.

Первое измерение задаёт динамический аспект развития процесса в терминах циклов, этапов, итераций и контрольных вех. Второе измерение задаёт статический аспект процесса в терминах компонентов процесса, рабочих потоков, приводящих к выработке искусственных объектов (артефактов), и участников.

Рабочие потоки процесса

Рабочие потоки процесса имеют следующее содержание:

- q **Сбор требований** — описание того, что система должна делать;
- q **Анализ** — преобразование требований к системе в классы и объекты, выявляемые в предметной области;
- q **Проектирование** — создание статического и динамического представления системы, выполняющего выявленные требования и являющегося эскизом реализации;
- q **Реализация** — производство программного кода, который превращается в исполняемую систему;
- q **Тестирование** — проверка всей системы в целом.

Каждый рабочий поток определяет набор связанных артефактов и действий. Артефакт — это документ, отчёт или выполняемый элемент. Артефакт может вырабатываться, обрабатываться или потребляться. Действие описывает задачи — шаги обдумывания, шаги исполнения и шаги проверки. Шаги выполняются участниками процесса (для создания или модификации артефактов).

Между артефактами потоков существуют зависимости. Например, модель Use Case, генерируемая в ходе сбора требований, уточняется моделью анализа из процесса анализа, обеспечивается проектной моделью из процесса проектирования, реализуется моделью реализации из процесса реализации и проверяется тестовой моделью из процесса тестирования.

4) Фаза исследования. Основные дисциплины и артефакты. Дисциплина «анализ предметной области», бизнес-анализ. Диаграммы потоков данных, деятельности. Моделирование предметной области.

Фаза исследования начинается, когда необходимость разработки признана руководством проекта, и заключается в том, что для проекта обосновываются требуемые ресурсы и формулируются требования к разрабатываемому изделию.

Артефакт - это общее название для любых видов информации, создаваемой, изменяемой или используемой сотрудниками при создании системы. Примерами артефактов могут служить диаграммы UML и связанный с ними текст, наброски и прототипы пользовательских интерфейсов, компоненты, планы тестирования и тестовые примеры.

Предметная область сильно влияет на все аспекты проекта: требования к системе, взаимодействие с пользователем, модель хранения данных, реализацию и т.д. Анализ предметной области, позволяет выделить ее сущности, определить первоначальные требования к функциональности и определить границы проекта.

Бизнес-анализ — дисциплина выявления деловых потребностей и нахождения решений деловых проблем. Решения часто включают компонент разработки систем, но могут также состоять из усовершенствования процессов, организационных изменений или стратегического планирования и разработки политики.

Диаграммы потоков данных (Data Flow Diagrams — DFD) представляют собой иерархию функциональных процессов, связанных потоками данных. Цель такого представления — продемонстрировать, как каждый процесс преобразует свои входные данные в выходные, а также выявить отношения между этими процессами.

Основными компонентами диаграмм потоков данных являются:

- внешние сущности;
- системы и подсистемы;
- процессы;
- накопители данных;
- потоки данных.

Внешняя сущность представляет собой материальный объект или физическое лицо, являющиеся источником или приемником информации. Определение некоторого объекта или системы в качестве внешней сущности указывает на то, что она находится за пределами границ анализируемой системы.

Процесс представляет собой преобразование входных потоков данных в выходные в соответствии с определенным алгоритмом.

Накопитель данных — это абстрактное устройство для хранения информации, которую можно в любой момент поместить в накопитель и через некоторое время извлечь.

Поток данных определяет информацию, передаваемую через некоторое соединение от источника к приемнику.

Модель предметной области — это визуальное представление концептуальных классов или объектов реального мира в терминах предметной области. Моделирование предметной области — один из начальных этапов проектирования системы, необходимый для выявления, классификации и формализации сведений обо всех аспектах предметной области, определяющих свойства разрабатываемой системы.

5. Фаза анализа и проектирования. Дисциплина «анализ требований». Способы извлечения и фильтрации требований. Бизнес-требования, бизнес-требования, системные требования, функциональные требования. Разработка и управление требованиями. Документ «спецификация требований к ПО ». Диаграммы прецедентов.

Дисциплина «анализ требований».

Анализ требований — часть процесса разработки программного обеспечения, включающая в себя сбор требований к программному обеспечению (ПО), их систематизацию, выявление взаимосвязей, а также документирование.

В процессе сбора требований важно принимать во внимание возможные противоречия требований различных заинтересованных лиц, таких как заказчики, разработчики или пользователи.

Полнота и качество анализа требований играют ключевую роль в успехе всего проекта. Требования к ПО должны быть документируемые, выполнимые, тестируемые, с уровнем детализации, достаточным для проектирования системы. Требования могут быть функциональными и нефункциональными.

Анализ требований включает три типа деятельности:

- Сбор требований — общение с клиентами и пользователями, чтобы определить, каковы их требования; анализ предметной области.
- Анализ требований — определение, являются ли собранные требования неясными, неполными, неоднозначными или противоречащими; решение этих проблем; выявление взаимосвязи требований.
- Документирование требований — требования могут быть задокументированы в различных формах, таких как простое описание, сценарии использования, пользовательские истории, или спецификации процессов.

Процесс анализа требований к информационной системе включает следующие фазы:

- Разработка требований
 - Выявление требований
 - Анализ требований
 - Спецификация требований
 - Проверка требований
- Управление требованиями

Способы извлечения и фильтрации требований.

- Интервью, опросы, анкетирование
- Мозговой штурм, семинар
- Наблюдение за производственной деятельностью, «фотографирование» рабочего дня
- Анализ нормативной документации
- Анализ моделей деятельности
- Анализ конкурентных продуктов
- Анализ статистики использования предыдущих версий системы

Идентификация стейкхолдеров

Стейкхолдер — физическое лицо или организация, имеющая права, долю, требования или интересы относительно системы или её свойств, удовлетворяющих их потребностям и ожиданиям (ISO/IEC 15288:2008, ISO/IEC 29148:2011).

Опрос стейкхолдеров

Опрос стейкхолдеров является широко используемой техникой при сборе требований. Эти опросы могут выявлять требования, не попавшие в рамки проекта либо противоречащие ранее собранным. Однако каждый стейкхолдер будет иметь собственные требования, ожидания и видение системы.

Сеансы совместного развития требований (СРТ)

Требования часто имеют сложное пересекающееся функциональное назначение, не известное отдельным стейкхолдерам. Такие требования часто упускаются или не полностью определяются во время их опросов. Такие требования могут быть выявлены при проведении сеансов СРТ. Такие сеансы проводятся под надзором подготовленного специалиста. Стейкхолдеры участвуют в обсуждениях, чтобы определить требования, проанализировать их детали и выявить скрытые пересекающиеся взаимосвязи между требованиями.

Бизнес-требования, бизнес-требования, системные требования, функциональные требования.

Бизнес-требования (business requirements) содержат высокоуровневые цели организации или заказчиков системы. Как правило, их высказывают те, кто финансируют проект, покупатели системы, менеджер реальных пользователей, отдел маркетинга. В этом документе объясняется, почему организации нужна такая система, то есть описаны цели, которые организация намерена достичь с ее помощью. Мне нравится записывать бизнес-требования в форме документа об образе и границах проекта, который еще иногда называют уставом проекта (project charter) или документом рыночных требований (market requirements document). Определение границ проекта представляет собой первый этап управления общими проблемами увеличения объема работ.

Требования пользователей (user requirements) описывают цели и задачи, которые пользователям даст система. К отличным способам представления этого вида требований относятся варианты использования, сценарии и таблицы «событие — отклик». Таким образом, в этом документе указано, что клиенты смогут делать с помощью системы.

Функциональные требования (functional requirements) определяют функциональность ПО, которую разработчики должны построить, чтобы пользователи смогли выполнить свои задачи в рамках бизнес-требований. Иногда они называются требованиями поведения (behavioral requirements), они содержат положения с традиционным «должен» или «должна»: «Система должна по электронной почте отправлять пользователю подтверждение о заказе».

Функциональные требования документируются в спецификации требований к ПО (software requirements specification, SRS), где описывается так полно, как необходимо, ожидаемое поведение системы.

Системные требования (system requirements) — это высокоуровневые требования к продукту, которые содержат многие подсистемы. Говоря о системе, мы подразумеваем программное обеспечение или подсистемы ПО и оборудования. Люди — часть системы, поэтому определенные функции системы могут распространяться и на людей.

Разработка и управление требованиями.

Управление требованиями к программному обеспечению ([англ. software requirements management](#)) — процесс, включающий идентификацию, выявление, документирование, анализ, отслеживание, приоритизацию требований, достижение соглашения по требованиям и затем управление изменениями и уведомление соответствующих заинтересованных лиц. Управление требованиями — непрерывный процесс на протяжении всего проекта разработки программного обеспечения.

Цель управления требованиями состоит в том, чтобы гарантировать, что организация документирует, проверяет и удовлетворяет потребности и ожидания её клиентов и внутренних или внешних заинтересованных лиц. Управление требованиями начинается с выявления и анализа целей и ограничений клиента. Управление требованиями, далее, включает поддержку требований, интеграцию требований и организацию работы с требованиями и сопутствующей информацией, поставляющейся вместе с требованиями.

Документ «спецификация требований к ПО

Спецификация требований программного обеспечения ([англ. Software Requirements Specification, SRS](#)) является полным описанием поведения системы, которая будет создана. Она включает ряд сценариев использования, которые описывают все виды взаимодействия пользователей с программным обеспечением. Сценарии использования также известны как функциональные требования. В дополнении к сценариям использования, спецификация программного обеспечения также содержит нефункциональные (или дополнительные) требования. Нефункциональные требования — требования, которые налагают дополнительные ограничения на систему (такие как требования эффективности работы, стандарты качества, или проектные ограничения).

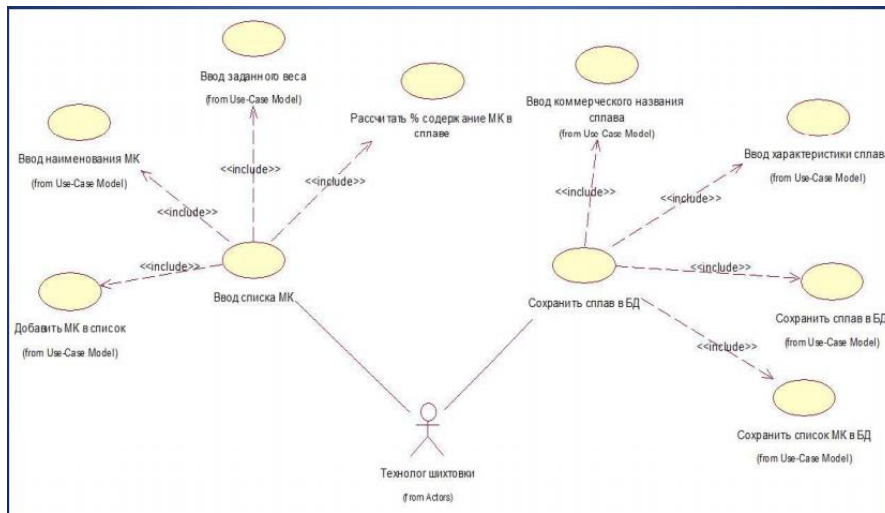
Рекомендуемые подходы для спецификации требований программного обеспечения описаны стандартом IEEE 830—1998. Этот стандарт описывает возможные структуры, желательное содержание, и качества спецификации требований программного обеспечения.

Диаграммы прецедентов.

Диаграмма прецедентов (диаграмма вариантов использования) в UML — диаграмма, отражающая отношения между актёрами и прецедентами и являющаяся составной частью модели прецедентов, позволяющей описать систему на концептуальном уровне.

Прецедент описывает поведение, демонстрируемое системой с целью получения значимого результата для одного или более актеров.

Прецедент соответствует отдельному сервису системы, определяет один из вариантов её использования и описывает типичный способ взаимодействия пользователя с системой. Варианты использования обычно применяются для спецификации внешних требований к системе



6. Фаза анализа и проектирования. Понятие архитектуры, ее многомерность. Основные методы проектирования и их особенности: структурное, функциональное, объектно-ориентированное, компонентное, проектирование на основе структур данных. Классы анализа. Виды классов: граница, управление, сущность. Диаграммы устойчивости. Архитектурные аспекты технологического процесса проектирования (по SWEBOK)

Архитектура - это:

- **значимые решения** по поводу организации ПС
- **структурные элементы** и их интерфейсы, при помощи которых компонуется система
- **поведение** - взаимодействие между этими элементами
- **компоновка элементов** в иерархию подсистем
- **стиль архитектуры** который направляет эту организацию [Philippe Kruchten, *The Rational Unified Process: An Introduction*]

Свойства архитектуры:

- **минимализм, отсутствие избыточности** - описание ПС, достаточное для понимания ее сущности и процессов функционирования
- **множественность представления**

Представление «4+1»



Представления (срезы) архитектуры

Представление	Роль	Содержание
Прецедентное (функциональное)	Системный архитектор (аналитик)	Ключевые сценарии (прецеденты), описывающие реализацию функционала системы, требования, предметная область
Логическое	Пользователь	Представление системы с точки зрения конечного пользователя (внешний вид, логическая структура).
Реализационное	Программист	Структура программного кода: пакеты, классы, библиотеки, интерфейсы, абстракции.
Процедурное	Системные интеграторы	Процессы, потоки, параллелизм, взаимодействие. Производительность, масштабируемость.
Представление распространения (развертывания)		Топология развертывания системы, распределение компонент по узлам сети, процедуры развертывания и администрирования

Представления (срезы) архитектуры

- **функциональное представление** – сценарии, прецеденты, требования, предметная область;
- **логическое представление** – реализация функциональности на системе взаимодействующих классов в логической модели проекта;
- **процедурное представление** – параллелизм, потоки, взаимодействие, масштабируемость, производительность;
- **представление развертывания** – компоновка, топология, коммуникации, администрирование, настройка;
- **представление разработки** – структура программного кода, библиотеки, интерфейсы, абстракции

Архитектура – минимально избыточное, согласованное описание системы, включая структуру, поведение, компоновку, стили разработки и значимые решения, создающее адекватное представление о всех аспектах ее организации

Процесс определения архитектуры, компонентов, интерфейсов и других характеристик системы или ее компонентов называется проектированием. Результат процесса проектирования – дизайн. Рассматриваемое как процесс, проектирование есть инженерная деятельность в рамках жизненного цикла (в данном контексте – программного обеспечения), в которой надлежащим образом анализируются требования для создания описания внутренней структуры ПО, являющейся основой для конструирования программного обеспечения как такового. Программный дизайн (как результат деятельности по проектированию) должен описывать архитектуру программного обеспечения, то есть представлять декомпозицию программной системы в виде организованной структуры компонент и интерфейсов между компонентами. Важнейшей характеристикой готовности дизайна является тот уровень детализации компонентов, который позволяет заняться их конструированием. Термины дизайн и архитектура могут использоваться взаимозаменяемым образом, но чаще говорят о дизайне как о целостном взгляде на архитектуру системы.

Проектирование программных систем можно рассматривать как деятельность, результат которой состоит из двух составных частей:

- Архитектурный или высокоуровневый дизайн (software architectural design, top-level design) - описание высокоуровневой структуры и организации компонентов системы;
- Детализированная архитектура (software detailed design) – описывающая каждый компонент в том объеме, который необходим для конструирования.

Методы проектирования

6.2 Функционально-ориентированное или структурное проектирование (Function-Oriented –

Structured Design) - Это один из классических методов проектирования, в котором декомпозиция сфокусирована на идентификации основных программных функций и, затем, детальной разработке и уточнении этих функций “сверху-вниз”. Структурное проектирование, обычно, используется после проведения структурного анализа с применением диаграмм потоков данных и связанным описанием процессов. Исследователи предлагают различные стратегии и метафоры или подходы для трансформации DFD в программную архитектуру, представляемую в форме структурных схем. Например, сравнивая управление и поведение с получаемым эффектом.

6.3 Объектно-ориентированное проектирование (Object-Oriented Design) Представляет собой множество методов проектирования, базирующихся на концепции объектов. Данная область активно эволюционирует с середины 80-х годов, основываясь на понятиях объекта (сущности), метода (действия) и атрибута (характеристики). Здесь главную роль играют полиморфизм и инкапсуляция, в то время, как в компонентно-ориентированном подходе большее значение придается мета-информации, например, с применением технологии отражения (reflection). Хотя корни объектно-ориентированного проектирования лежат в абстракции данных (к которым добавлены поведенческие характеристики), так называемый responsibility-

driven design или проектирование на основе <функциональной> ответственности по SWEBOK* может рассматриваться как альтернатива объектно-ориентированному проектированию.

6.4 Проектирование на основе структур данных (Data-Structure-Centered Design)

В данном подходе фокус сконцентрирован в большей степени на структурах данных, которыми управляет система, чем на функциях системы. Инженеры по программному обеспечению часто вначале описывают структуры данных входов (inputs) и выходов (outputs), а, затем, разрабатывают структуру управления этими данными (или, например, их трансформации).

6.5 Компонентное проектирование (Component-Based Design) Программные компоненты являются независимыми единицами, которые обладают однозначно-определенными (well-defined) интерфейсами и зависимостями (связями) и могут собираться и развертываться независимо друг от друга. Данный подход призван решить задачи использования, разработки и интеграции таких компонент с целью повышения повторного использования активов (как архитектурных, так и в форме кода).

Классы анализа

Классы анализа задают элементы начальной концептуальной модели для 'того в системе, что имеет ответственности и алгоритм работы'. Они представляют прототипы классов системы и являются 'первым подходом' к основным абстракциям, которыми должна управлять система. Классы анализа могут поддерживаться сами по себе, если требуется концептуальный, "высокого уровня" обзор системы. Классы анализа также дают начало основным абстракциям проекта системы: классам проекта и подсистемам.

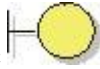
Представление UML: Класс, определяемый как <<граничащий>>, <<сущностный>> или <<управляющий>>.

Класс анализа может иметь следующие свойства:

- **имя:** имя класса
- **описание:** краткое описание роли класса в системе
- **ответственности:** список ответственностей класса
- **атрибуты:** атрибуты класса

Классы анализа подразделяются на следующие стереотипы:

- Пограничные классы
- Управляющие классы
- Сущностные классы

Пограничный класс  - это класс, применяемый для моделирования взаимодействия между околосистемными и внутрисистемными объектами. Такое взаимодействие предусматривает передачу и преобразование событий и внесение изменений в представление системы (например, интерфейс).

Пограничные классы моделируют части системы, зависящие от ее окружения. Сущностные и управляющие классы - части, не зависящие от околосистемных объектов. Таким образом, изменение GUI или протокола связи вызывает изменение только пограничных, но не сущностных и управляющих классов.

К наиболее распространенным пограничным классам относятся окна, протоколы связи, интерфейсы принтеров, детекторы и терминалы. Нет необходимости моделировать рутинные части интерфейсов, такие как кнопки, как отдельные пограничные классы. Обычно все окно можно рассматривать как удобный структурный пограничный объект. Пограничные классы полезны также при захвате интерфейсов API, которые могут быть не объектно-ориентированными, например устаревшего кода.

Пограничный класс выступает посредником между интерфейсом и внешним по отношению к системе объектом. Пограничные объекты изолируют систему от изменений в окружающем мире (изменений в

интерфейсах связи с другими системами, требованиях пользователей и т.п.), не позволяя этим изменениям повлиять на остальную часть системы.

В системе может быть несколько типов пограничных классов:

- **Классы пользовательского интерфейса** - промежуточные классы в соединении с пользователями системы
- **Классы интерфейса системы** - промежуточные классы в соединении с другой системой
- **Классы интерфейса устройств** - классы, предоставляющие интерфейс устройствам (например, детекторам), обнаруживающим внешние события



Управляющий класс - это класс, служащий для моделирования управляющего поведения для одного или нескольких вариантов использования. Управляющие объекты (экземпляры управляющих классов) часто контролируют другие объекты, поэтому их поведение можно назвать координирующим. Управляющие классы инкапсулируют конкретное поведение для варианта использования. Управляющие классы могут давать вклад в понимание системы, поскольку они представляют динамику системы, обработку основных задач и управляющие потоки.

Управляющие классы предоставляют поведение, которое:

- Не зависит от околосистемных объектов (не меняется при изменении этих объектов),
- Определяет управляющую логику (порядок событий) и транзакции в варианте использования.
- Мало меняется при изменении внутренней структуры или поведения сущностных классов,
- Использует или задает содержимое нескольких сущностных классов и поэтому должно координировать их поведение.
- Выполняется по-разному при каждой активации (поток событий может находиться в нескольких состояниях).



Сущностный класс - это класс, применяемый для моделирования хранимой информации и связанного с ней поведения. Сущностные объекты (экземпляры сущностных классов) служат для хранения и обновления информации о некотором явлении, например о событии, пользователе или реально существующем объекте. Обычно это постоянные объекты с атрибутами и взаимосвязями, необходимыми в течение длительного времени, иногда - в течение всей жизни системы.

Сущностный объект обычно не связан с какой-нибудь одной конкретной реализацией варианта использования в анализе; иногда сущностный объект не связан даже с самой системой. Значения его атрибутов и взаимосвязей часто задаются субъектом. В некоторых случаях сущностный объект может понадобиться для выполнения внутренних системных задач. Поведение сущностных объектов может быть столь же сложным, сколь и у других стереотипов объектов. Однако, в отличие от других объектов, это поведение тесно связано с явлением, которое представляет сущностный объект. Сущностные объекты не зависят от среды (субъектов).

Сущностные классы предоставляют другую точку зрения для изучения системы. Они демонстрируют логическую структуру данных, которая поможет вам понять, что система должна предлагать своим пользователям.

Сущностные классы представляют хранилища информации в системе; они обычно используются для представления ключевых концепций системы. Сущностные объекты, как правило, пассивны и постоянны. Их основные обязанности заключаются в хранении и управлении информацией в системе.

Источником для образования сущностных классов часто служат Глоссарий (разработанный во время определения требований) и модель бизнес-домена (разработанная во время моделирования бизнеса, если оно выполнялось).

Диаграмма устойчивости (robustness diagram).

Для такого уровня абстракции наиболее приемлема диаграмма устойчивости UML. Так как процесс ее построения в первую очередь направлен на формализацию текста прецедента. Формализация заключается в выделении на основании прецедента набора объектов и взаимосвязей между ними. Более подробно диаграмма устойчивости описана в книгах из раздела "Ресурсы".

Фактически, эта модель дает полную информацию о статическом виде предметной области системы.

Диаграмма устойчивости строится на базе трех классов анализа: пограничный класс, класс сущности и управляющий класс. Это способствует разделению предметной области на три составляющие части: объекты, совершаемые над ними действия и точки доступа к этим действиям. Это разделение очень напоминает расслоение, присутствующее в архитектуре трехуровневого приложения. Где приложения представляется в виде трех уровней: представление, логика и данные.

7. Фаза анализа и проектирования. Дисциплина проектирование (design). Ключевые моменты проектирования по SWEBOK: параллелизм, контроль и обработка событий, распределение компонентов, обработка ошибок и исключительных ситуаций и обеспечение отказоустойчивости, взаимодействие и представление (MVC), сохраняемость данных (доступность «долгоживущих» данных).

Проектирование ПО(design) - процесс определения архитектуры, компонентов, интерфейсов и других характеристик системы и состава программного продукта. или ее компонентов называется проектированием. Результат процесса проектирования – дизайн. Программный дизайн должен описывать архитектуру программного обеспечения, то есть представлять декомпозицию программной системы в виде организованной структуры компонент и интерфейсов между компонентами. Важнейшей характеристикой готовности дизайна является тот уровень детализации компонентов, который позволяет заняться их конструированием. Дизайн = архитектура (почти).

Проектирование программных систем можно рассматривать как деятельность, результат которой состоит из двух составных частей:

- Архитектурный или высокоуровневый дизайн (software architectural design, top-level design) – описание высокоуровневой структуры и организации компонентов системы;
- Детализированная архитектура (software detailed design) – описывающая каждый компонент в том объеме, который необходим для конструирования.

2.1 Параллелизм (Concurrency) Эта тема охватывает вопросы, подходы и методы организации процессов, задач и потоков для обеспечения эффективности, атомарности, синхронизации и распределения (по времени) обработки информации.

2.2 Контроль и обработка событий (Control and Handling of Events) В самом названии данной темы заложен комплекс обсуждаемых вопросов. В частности, данная тема касается и неявных методов обработки событий,

часто реализуемых в виде функции обратного вызова (call-back), как одной из фундаментальных концепций обработки событий.

2.3 Распределение компонентов (Distribution of Components) Распределение (и выполнение) по различным узлам обработки в терминах аппаратного обеспечения, сетевой инфраструктуры и т.п. Один из важнейших вопросов данной темы – использование связующего программного обеспечения (middleware*). * часто middleware переводят как “промежуточное программное обеспечение”. Такой вариант перевода, к сожалению, рассматривает связующее ПО во второстепенной – “промежуточной” роли. Читатель, безусловно, может не согласиться с такой трактовкой, однако, многолетняя практика автора в обсуждении архитектурных вопросов с различными специалистами демонстрирует именно такой взгляд пользователей, не знакомых или не имеющих успешного опыта разработки и эксплуатации распределённых систем.

2.4 Обработка ошибок и исключительных ситуаций и обеспечение отказоустойчивости (Errors and Exception Handling and Fault Tolerance) Вопрос темы, как ни странно, формулируется достаточно просто – как предотвратить сбой или, если сбой все же произошел, обеспечить дальнейшее функционирование системы.

2.5 Взаимодействие и представление (Interaction and Presentation) Тема касается вопросов представления информации пользователям и взаимодействия пользователей с системой, с точки зрения реакции системы на действия пользователей. Речь в этой теме идет о реакции системы в ответ на действия пользователей и организации ее отклика с точки зрения внутренней организации взаимодействия, например, в рамках популярной концепции ModelView-Controller

2.6 Сохраняемость данных (Data Persistence) Именно сохраняемость, а не сохранность, так как тема касается не доступа к базам данных, как такового, а также не гарантий сохранности информации. Суть вопроса – как должны обрабатываться “долгоживущие” данные.

8) Фаза анализа и проектирования. Многоуровневая архитектура клиент-серверных приложений. Тонкие и толстые клиенты. Локальное и сетевое взаимодействие слоев через интерфейсы и протоколы. Совместное использование кода различными типами клиентов.

Цель процесса анализа и проектирования состоит в разработке технических инструкций, предписывающих, как реализовать ПС, удовлетворяющую сформулированным требованиям. Для этого следует хорошо понять требования к ПС и преобразовать их в проект системы, выбрав правильную стратегию реализации. На ранних стадиях процесса должна быть создана устойчивая архитектура, на основе которой можно спроектировать ПС, легкую для понимания, построения и развертывания.

Главной задачей анализа является преобразование требований в форму, понятную разработчику, то есть, определение подсистем, компонентов и классов, с помощью которых реализуется требуемое поведение ПС.

Проектирование – это уточнение результатов анализа, направленное на оптимизацию с учетом ограничений, накладываемых нефункциональными требованиями, средой реализации и т. д.

«Клиент — сервер» — вычислительная или сетевая архитектура, в которой задания или сетевая нагрузка распределены между поставщиками услуг, называемыми серверами, и заказчиками услуг, называемыми клиентами. Фактически клиент и сервер — это программное обеспечение. Обычно эти программы расположены на разных вычислительных машинах и взаимодействуют между собой через вычислительную сеть посредством сетевых протоколов, но они могут быть расположены также и на одной машине. Программы-сервера ожидают от клиентских программ запросы и предоставляют им свои ресурсы в виде данных или в виде сервисных функций. Поскольку одна программа-сервер может выполнять запросы от множества программ-клиентов, её размещают на специально выделенной вычислительной машине, настроенной особым образом, как правило, совместно с другими программами-серверами, поэтому производительность этой машины должна быть высокой.

Тонкий клиент — компьютер или программа-клиент в сетях с клиент-серверной или терминальной архитектурой, который переносит все или большую часть задач по обработке информации на сервер.

Толстый клиент — это приложение, обеспечивающее (в противовес тонкому клиенту) расширенную

функциональность независимо от центрального сервера. Часто сервер в этом случае является лишь хранилищем данных, а вся работа по обработке и представлению этих данных переносится на машину клиента.

9. Фаза анализа и проектирования Проектирование графического интерфейса (GUI). Основные аспекты. Архитектурное проектирование, основанное на GUI. Факторы, характеризующие GUI: производительность, человеческие ошибки, обучение, субъективное восприятие, запоминание, поиск, визуализация, навигация

- Интерфейс должен быть интуитивно понятным. Таким, чтобы пользователю не требовалось объяснять как им пользоваться.
- Для упрощения процесса изучения необходима справка. Буквально — графическая подсказка, объясняющая значение того или иного ЭИ. Полное руководство должно быть частью интерфейса, доступной в любой момент.
- Возвращайте пользователя в то место, где он закончил работу в прошлый раз. Зачем нажимать все заново?
- Чаще всего, пользователи в интерфейсе сначала ищут сущность(существительное), а затем действие(глагол) к ней. Следуйте правилу «существительное -> глагол». Например, шрифт - > изменить.
- Чем быстрее человек увидит результат — тем лучше. Пример — «живой» поиск, когда варианты, в процессе набора поискового запроса. Основной принцип: **программа должна взаимодействовать с пользователем на основе наименьшей значимой единицы ввода.**
- Используйте квазирежимы. Например, ввод заглавных букв с зажатой клавишей shift — это квазирежим. С включенным capslock — режим. Основное отличие в том, что человек может забыть в каком режиме он находится, а в квазирежиме(с зажатой доп. клавишей) это сделать гораздо сложнее.
- Следует с осторожностью предоставлять пользователю возможность, по установке личных настроек. Представьте, сколько времени потратит сотрудник настраивая Word, если его интерфейс был полностью переделан предыдущим.
- Чем больше пользователь работает с какой-то конкретной задачей, тем больше он на ней концентрируется и тем меньше перестает замечать подсказки и сообщения, выводимые программой. Чем более критической является задача, тем меньше вероятность того, что пользователь заметит предупреждения относительно тех или иных потенциально опасных действий.

Факторы, характеризующие GUI: производительность, человеческие ошибки, обучение, субъективное восприятие, запоминание, поиск, визуализация, навигация

Существует четыре основных (все остальные — производные) критерия качества любого интерфейса, а именно: скорость работы пользователей, количество человеческих ошибок, скорость обучения и субъективное удовлетворение пользователей (подразумевается, что соответствие интерфейса задачам пользователя является неотъемлемым свойством интерфейса).

Скорость выполнения работы является важным критерием эффективности интерфейса. Длительность выполнения работы пользователем состоит из длительности восприятия исходной информации, длительности интеллектуальной работы (в смысле – пользователь думает, что он должен сделать), длительности физических действий пользователя и длительности реакции системы. Как правило, длительность реакции системы является наименее значимым фактором.

Важным критерием эффективности интерфейса является количество человеческих ошибок. В некоторых случаях одна или две человеческие ошибки погоды не делают, но только тогда, когда эти ошибки легко исправляются. Однако часто минимальная ошибка приводит к совершенно катастрофическим последствиям, например, за одну секунду операционистка в банке может сделать кого-то богаче, а банк, в свою очередь, беднее (впрочем, обычно беднее становятся все). Классический сюжет из жизни летчика, который после взлета хотел убрать шасси, но вместо этого включил систему аварийного катапультирования, возник отнюдь не на пустом месте.

В традиционной науке о человеко-машинном взаимодействии роль обучения операторов чрезвычайно велика. Цель ставится возможность работы с системой для любого человека, независимо от его свойств и навыков, при этом целенаправленное обучение пользователей, как правило, не производится. Всё это делает проблему обучения пользователей работе с компьютерной системой чрезвычайно важной. Начиная с определенного объема функциональности системы, количество пользователей, знающих всю функциональность, неуклонно снижается. Т.е. чем объемнее система, тем больше шансов на то, что среднестатистический пользователь знает о ней очень немного (относительно общего объема функциональности). Пользователи работают с системой не слишком эффективно, поскольку вместо методов адекватных они используют методы знакомые. Вовторых, достаточно часто случается, что пользователи, не зная, что имеющийся продукт делает то, что им нужно, ищут (и находят) продукт конкурента. Втретьих, при таком положении вещей затруднительно продавать новые версии продукта: если пользователь не умеет пользоваться и тем, что есть, убеждать его совершить покупку ради новой функциональности придется на довольно шатком фундаменте.

Пользователи воспринимают одинаково положительно как удобные, но приятные интерфейсы, так и простые, эффективные, но сухие и скучные. Таким образом, субъективные факторы имеют тот же вес, что и объективные. Разумеется, субъективность доминирует над объективностью только в тех случаях, когда покупателем системы выступает сам пользователь, но и в прочих случаях роль «крутоты» зачастую существенна, хотя бы потому, что повышение количества радости при прочих равных почти всегда приводит к повышению человеческой производительности. Это делает неактуальными вечные споры о первичности формы или функции. И то и другое важно

10. Виды моделей. Сущность UML как средства моделирования. Структура UML, статическая и динамическая составляющие модели. Составные элементы: сущности, отношения, диаграммы. Виды сущностей: структурные сущности – класс, интерфейс, кооперация, прецедент, активный класс, компонент, узел; поведенческие сущности – взаимодействия, деятельности, автоматы, пакет, примечание.

Виды моделей.

1. Модель бизнес-процессов (business use case model) описывает деятельность организации;
2. Модель бизнес-анализа (business analysis model) определяет контекст системы;

3. Модель вариантов использования (use case model) выражает функциональные требования к системе;
4. Модель анализа (analysis model) – необязательная. Определяет проектные решения на концептуальном уровне;
5. Проектная модель (design model) охватывает словарь предметной области и области решения;
6. Модель данных (data model) – необязательная. Определяет представление данных в базах данных и других репозиториях;
7. Модель размещения (deployment model) специфицирует топологию аппаратных средств, на которых работает система, вместе с механизмами параллелизма и синхронизации;
8. Модель реализации (implementation model) определяет, какие части используются для сборки и реализации физической системы.

Сущность UML как средства моделирования. Структура UML, статическая и динамическая составляющие модели.

Унифицированный язык моделирования UML (Unified Modeling Language) – это язык для определения, представления, проектирования и документирования программных систем, организационно-экономических систем, технических систем и других систем различной природы. UML содержит стандартный набор диаграмм и нотаций самых разнообразных видов.

Разработка UML преследовала следующие цели: • предоставить разработчикам единый язык визуального моделирования; • предусмотреть механизмы расширения и специализации языка; • обеспечить независимость языка от языков программирования и процессов разработки; • интегрировать накопленный практический опыт.

Структура:

1. Семантика языка UML. Представляет собой некоторую метамодель, которая определяет абстрактный синтаксис и семантику понятий объектного моделирования на языке UML.

2. Графическая Нотация языка UML. Представляет собой графическую нотацию для визуального представления семантики языка UML.

Статическая составляющая модели – структурные сущности, а Поведенческие сущности — это динамические части моделей UML.

Составные элементы: сущности, отношения, диаграммы.

сущности - абстракции, являющиеся основными элементами модели;

отношения - связи между сущностями;

диаграммы - группирующие представляющие интерес множества сущностей и отношений.

Виды сущностей: структурные сущности – класс, интерфейс, кооперация, прецедент, активный класс, компонент, узел

- **классы (Class)** — это набор объектов, разделяющих одни и те же атрибуты, операции, отношения и семантику. Класс реализует один или несколько интерфейсов и изображается в виде прямоугольника, включающего имя класса, имена атрибутов, операций, примечание;
- **интерфейсы (Interface)** — это набор операций, которые определяют сервис класса или компоненты.

Интерфейс графически изображается в виде круга и, как правило, присоединяется к классу или к компоненту, который реализует данный интерфейс;

- **кооперации (Collaboration)** — определяют взаимодействие и служат для объединения ролей и других элементов, которые взаимодействуют вместе так, что получающееся в результате поведение объекта оказывается большим, чем просто сумма всех элементов. Изображается в виде эллипса с пунктирной границей;
- **прецеденты (Use case)** — описание набора последовательностей действий, которые выполняются системой и имеют значение для конкретного действующего лица (Actor). Прецеденты изображаются в виде эллипса и используются для структурирования поведенческих сущностей в модели;
- **активные классы (Active class)** — это классы, чьими экземплярами являются активные объекты, которые владеют процессом или потоком управления и могут инициировать управляющее воздействие. Стереотипами конкретного класса являются процесс (Process) и поток (Thread). Графически такой класс изображается как класс с жирной границей;
- **компоненты (Component)** — это физически заменяемые части системы, обеспечивающие реализацию ряда интерфейсов. Компонент — это физическое представление таких логических элементов, как классы, интерфейсы и кооперации. Предметная область компонентов относится к реализации. Изображаются компоненты в виде прямоугольника с ярлыками слева и, как правило, имеют только имя и примечание;
- **узлы (Node)** — физические объекты, которые существуют во время исполнения программы и представляют собой коммуникационный ресурс, обладающий, по крайней мере, памятью, а зачастую и процессором. На узлах могут находиться выполняемые объекты и компоненты. Изображаются узлы в виде куба, имеют имя и примечание.

поведенческие сущности – взаимодействия, деятельности, автоматы, пакет, примечание.

- **взаимодействия (Interaction)** — включают набор сообщений, которыми обмениваются указанные объекты с целью достижения указанной цели. Взаимодействие описывается в контексте кооперации и изображается направленной линией, маркируется именем операции сверху;
- **автоматы (State machine)** — спецификации поведения, представляющие собой последовательности состояний, через которые проходит в течение своей жизни объект, или взаимодействие в ответ на происходящие события (а также ответные действия объекта на эти события). Автомат прикреплен к исходному элементу (классу, кооперации или методу) и служит для определения поведения его экземпляров. Изображается автомат как прямоугольник с закругленными углами.
- **Деятельность (activity)** можно считать частным случаем состояния, который характеризуется продолжительными (по времени) не атомарными вычислениями.
- **пакеты (Package)** — обобщенный механизм для организации элементов в группы.
- **Примечание (Note)** — пояснительные элементы языка. Они содержат текст комментария, изображаются в виде прямоугольника с загнутым уголком страницы.

11.UML. Виды отношений: зависимость, ассоциация, агрегация, композиция, включение, обобщение, реализация. Отношения. Связи – отношения между объектами. Направленность связи, Сообщения. Диаграммы объектов. Ассоциации – отношения между классами. Свойства ассоциации: имя, кратность, навигация, атрибуты. Рефлексивные ассоциации, деревья и сети. Классы атрибутов ассоциаций (классы-ассоциации). Зависимости. Зависимости использования «use», «call», «parameter», «send» и «instantiate». Зависимости абстракции. Зависимости доступа.

Классы изображаются в виде прямоугольников, **ассоциации** – в виде сплошных линий, направления ассоциаций указываются стрелками, **агрегации** и **композиции** – в виде сплошных линий с ромбом на конце, связь **обобщения** – в виде сплошных линий с треугольником на конце, **зависимость** – в виде пунктирной линии со стрелкой.

Отношения

К базовым отношениям между объектами, которые позволяют строить блоки UML, можно отнести следующие (рис. 4):

Тип отношения	UML-синтаксис		Краткая семантика
	источник	цель	
Зависимость	-----	>	Исходный элемент зависит от целевого элемента и изменение последнего может повлиять на первый.
Ассоциация	————		Описание набора связей между объектами.
Агрегация	◇————		Целевой элемент является частью исходного элемента.
Композиция	◆————		Строгая (более ограниченная) форма агрегирования.
Включение	⊕————		Исходный элемент содержит целевой элемент.
Обобщение	————	▷	Исходный элемент является специализацией более обобщенного целевого элемента и может замещать его.
Реализация	-----	▷	Исходный элемент гарантированно выполняет контракт, определенный целевым элементом.

Рис. 4. Отношения UML

• **зависимость** (Dependency) — это семантическое отношение между двумя сущностями, при котором изменение одной из них (независимой сущности) может отразиться на семантике другой (зависимой). Виды зависимостей, которые соответствуют нескольким видам отношений между объектами, перечислены ниже: — абстракция (Abstraction) — представляет собой изменение уровня абстрактности для некоторого понятия. Как правило, один из элементов, более абстрактный, а второй — более конкретный, хотя возможны ситуации, когда оба элемента являются двумя возможными вариантами понятия, существующими на одном уровне абстракции. К зависимости абстракции относятся следующие стереотипы (в порядке возрастания специфичности отношений): трассировать (Trace), уточнять (Refine), реализовать (есть собственная нотация)

и выводиться (Derive),

— связывание (Binding) — связывает элемент с шаблоном. Аргументы, необходимые для параметров шаблона, прикреплены к зависимости связывания в виде списка,

— комбинирование (Combination) — соотносит две части описания классификатора (любой элемент модели, описывающий определенные черты структуры и поведения системы), чтобы получить полное описание элемента,

— разрешение (Permission) — зависимость (всегда изображается в виде особого стереотипа), связывающая тот или иной пакет (или класс) с другим пакетом (или классом), которому он предоставляет разрешение использовать свое содержимое. Стереотипами зависимости разрешения являются: быть доступным (Access), быть дружественным (Friend) и импортировать (Import),

— использование (Usage) — описывает ситуацию, когда одному элементу для правильной реализации или функционирования требуется присутствие другого элемента. К стереотипам этого вида зависимости относятся: вызывать (Call), создать экземпляр (Instantiate), параметр (Parameter) и отправить (Send);

- **ассоциация** (Association) — структурное отношение, описывающее множество связей между объектами классификаторов, где связь (Link) — это соединение между объектами, которое описывает связи между их экземплярами. Ассоциации являются как бы клеем, который связывает систему воедино. Без ассоциаций мы имели бы просто некоторое количество классов, не способных взаимодействовать друг с другом. У ассоциации может быть имя, однако основную информацию об ассоциации следует искать у ее полюсов, где описывается, каким образом каждый объект участвует в ассоциации: у ассоциации есть список, состоящий из двух или более полюсов ассоциации: каждый из них определяет роль, которую играет данный классификатор в этой ассоциации. Один и тот же классификатор может играть несколько ролей, которые не являются взаимозаменяемыми. Каждый полюс ассоциации описывает свойства, применимые к конкретному объекту этой ассоциации, например сколько раз один объект может появляться в связях (множественность). Некоторые свойства (такие как допустимость навигации) применимы только к бинарным ассоциациям, хотя большинство свойств относится и к бинарным, и к n-арным ассоциациям;

- **обобщение** (Generalization) — это отношение специализации/обобщения, при котором объекты специализированного элемента (потомка — Child) можно подставить вместо объектов обобщенного элемента (родителя, предка — Parent). В случае обобщения классов прямой предок может именоваться суперклассом, а прямой потомок — подклассом;

- **реализация** (Realization) — отношение между спецификацией и ее программной реализацией; указание на то, что поведение наследуется без структуры.

Мы перечислили четыре основных отношения. В UML также существуют их варианты: уточнение (Refinement), трассировка (Trace), включение (Include), расширение (Extend).

Визуализация представления проектируемой системы с различных точек зрения в UML реализована посредством диаграмм — проекций системы. Диаграмма (Diagram) — это графическое представление множества элементов, которое изображается в виде связного графа с вершинами (сущностями) и ребрами (отношениями).

- **диаграмма классов** (Class diagram) — структурная диаграмма, на которой показано множество классов, интерфейсов, коопераций и отношений между ними (рис. 5).

Диаграмма классов определяет классы системы и различного рода связи, которые существуют между ними (ассоциации, агрегации, композиции, зависимости, обобщения, реализации). На диаграммах классов изображаются также атрибуты классов, операции классов и ограничения, которые накладываются на связи между классами. (От РОМАНОВА)

Диаграммы объектов - Они показывают множество объектов - экземпляров классов (изображенных на диаграмме классов) и отношений между ними в некоторый момент времени. То есть *диаграмма объектов* - это своего рода снимок состояния системы в определенный момент времени, показывающий множество объектов, их состояния и отношения между ними в данный момент.

Таким образом, диаграммы объектов представляют *статический вид системы с точки зрения проектирования и процессов*, являясь основой для сценариев, описываемых диаграммами взаимодействия. Говоря другими словами, *диаграмма объектов используется для пояснения и детализации диаграмм взаимодействия*, например, диаграмм последовательностей.

Ассоциации

Имя ассоциации - необязательный элемент ее обозначения. Однако если оно задано, то записывается с заглавной буквы рядом с линией ассоциации. Отдельные классы ассоциации могут играть определенную роль в соответствующем отношении, на что явно указывает имя конечных точек ассоциации на диаграмме.

Следующий элемент обозначений - **кратность** ассоциации. Кратность относится к концам ассоциации и обозначается в виде интервала целых чисел, аналогично кратности атрибутов и операций классов, но без прямых скобок. Этот интервал записывается рядом с концом соответствующей ассоциации и означает потенциальное число отдельных экземпляров класса, которые могут иметь место, когда остальные экземпляры или объекты классов фиксированы.

Рефлексивные сообщения (который объект посылает сам себе) изображаются над псевдосоединением – дугой над объектом.

Зависимости использования

Существует пять зависимостей использования: **«use»**, **«call»**, **«parameter»**, **«send»** и **«instantiate»**, каждая из которых рассматривается ниже.

Зависимость «use»

Самым распространенным стереотипом зависимости является «use», который просто обозначает, что клиент каким-то образом использует поставщика. Если на диаграмме указана просто пунктирная линия со стрелкой зависимости без стереотипа, можно быть совершенно уверенным, что подразумевается зависимость «use».

Эта зависимость генерируется любой из следующих ситуаций.

1. Операции класса А необходим параметр класса В.
2. Операция класса А возвращает значение класса В.
3. Операция класса А где-то в своей реализации использует объект класса В, но не в качестве атрибута.

Варианты 1 и 2 довольно просты, а вот вариант 3 представляет больший интерес. Такая ситуация возможна, если одна из операций класса А создала временный объект класса В.

Хотя одна зависимость «use» может использоваться как универсальная для всех трех перечисленных случаев, есть и другие, более специализированные стереотипы зависимостей, которые можно было бы применить.

Ситуации 1 и 2 можно более точно смоделировать с помощью зависимости «parameter», а ситуацию 3 – с помощью зависимости «call». Однако от UML-модели не часто требуется такой уровень детализации, и большинство разработчиков моделей считают, что намного понятней и проще просто устанавливать между соответствующими классами зависимость «use», как показано выше.

Зависимость «call»

Зависимость «call» (вызов) устанавливается между операциями – операция-клиент вызывает операцию-поставщик. Этот тип зависимости в UML-моделировании используется не очень широко. Он применяется на уровне детализации, более глубоком, чем тот, на который большинство разработчиков готовы пойти. Кроме того, в настоящее время очень немногие средства моделирования поддерживают зависимости между операциями.

Зависимость «parameter»

Поставщик является параметром операции клиента.

Зависимость «send»

Клиент – это операция, посылающая поставщика (который должен быть сигналом) в некоторую неопределенную цель. Пока будем представлять их как особые типы классов, используемые для передачи данных между клиентом и целью.

Зависимость «instantiate»

Клиент – это экземпляр поставщика.

Зависимости абстракции

Зависимости абстракции моделируют зависимости между сущностями, находящимися на разных уровнях абстракции. В качестве примера можно привести класс аналитической модели и тот же класс в проектной модели. Существует четыре зависимости абстракции: **«trace»**, **«substitute»**, **«refine»** и **«derive»**.

Зависимость «trace»

Зависимость «trace» часто используется, чтобы проиллюстрировать отношение, в котором поставщик и клиент представляют одно понятие, но находятся в разных моделях. Например, поставщик и клиент могут находиться на разных стадиях разработки. Поставщик мог бы быть аналитическим представлением класса, а клиент – более детальным проектным представлением. Также «trace» можно использовать, чтобы показать отношение между функциональным требованием, таким как «банкомат должен обеспечивать возможность снятия наличных денег вплоть до достижения кредитного лимита карты», и прецедентом, поддерживающим это требование.

Зависимость «substitute»

Зависимость «substitute» (заместить) показывает, что клиент во время выполнения может заменять поставщика. Замещаемость основывается на общности контрактов и интерфейсов клиента и поставщика, т.е. они должны предоставлять один и тот же набор сервисов. Обратите внимание, что замещаемость не достигается посредством отношений специализации/обобщения между клиентом и поставщиком (специализация/обобщение обсуждаются в статьях далее). В сущности, «substitute» специально разработана для использования в средах, не поддерживающих специализации/обобщения.

Зависимость «refine»

Тогда как зависимость «trace» устанавливается между элементами разных моделей, «refine» (уточнить) может использоваться между элементами одной и той же модели. Например, в модели может быть две версии класса, одна из которых оптимизирована по производительности. Поскольку оптимизация производительности является разновидностью уточнения, это отношение между двумя классами можно смоделировать как зависимость «refine» с примечанием, описывающим суть уточнения.

Зависимость «derive»

Стереотип «derive» (получить) используется, когда необходимо явно показать возможность получения одной сущности как производной от другой. Например, в имеющемся классе BankAccount есть список Transaction (транзакция), в котором каждая Transaction содержит Quantity (количество) денег. По требованию всегда можно вычислить текущий баланс, суммируя Quantity по всем Transaction. Существует три способа показать, что balance (баланс) счета (его Quantity) может быть производной сущностью. Все перечисленные способы обозначения балансов как производных сущностей эквивалентны.

Зависимости доступа

Зависимости доступа выражают способность доступа одной сущности к другой. Существует три зависимости доступа: **«access»**, **«import»** и **«permit»**.

Зависимость «access»

Зависимость «access» (доступ) устанавливается между пакетами. В UML пакеты используются для группировки сущностей. Самое главное здесь то, что «access» разрешает одному пакету доступ ко всему

открытому содержимому другого пакета. Однако каждый пакет определяет пространство имен, и с установлением отношения «access» пространства имен остаются изолированными. Это означает, что элементы клиентского пакета должны использовать имена путей (pathnames), когда хотят обратиться к элементам пакета поставщика. Более подробное обсуждение данного вопроса представлено далее.

Зависимость «import»

Зависимость «import» концептуально аналогична «access», за исключением того, что пространство имен поставщика объединяется с пространством имен клиента. Это обеспечивает возможность элементам клиента организовывать доступ к элементам поставщика без необходимости указывать в именах элементов имя пакета. Однако иногда это может приводить к конфликтам имен, если имена элемента клиента и элемента поставщика совпадают. Очевидно, что в этом случае необходимо использовать полные имена.

Зависимость «permit»

Зависимость «permit» (разрешить) обеспечивает возможность управляемого нарушения инкапсуляции, но в целом этого отношения следует избегать. Клиентский элемент имеет доступ к элементу поставщика независимо от объявленной видимости последнего. Часто зависимость «permit» устанавливается между двумя родственными классами, когда клиентскому классу выгодно (вероятно, по причинам производительности) иметь доступ к закрытым членам поставщика. Не все языки программирования поддерживают зависимости «permit». C++ позволяет классу объявлять друзей, которые имеют разрешение на доступ к его закрытым членам. Но эта возможность была (и, наверное, благоразумно) изъята из Java и C#.

12. UML. Принятые деления: классификатор-экземпляр, интерфейс-реализация. Расширения: ограничения, стереотипы.

Классификация диаграмм. Диаграммы классов (объектов).

Диаграммы взаимодействий, коммуникационные диаграммы.

Принятые деления: классификатор-экземпляр, интерфейс-реализация.

Принятые деления описывают конкретные способы представления мира.

Абстрактное понятие типа сущности – это классификатор, а отдельные конкретные сущности – экземпляры.

(ПРИМЕР Классификатор – это абстрактное понятие, например тип банковского счета. Экземпляр – конкретная сущность, например ваш банковский счет или мой банковский счет.)

Основная идея этих понятий в том, чтобы отделить то, *что* выполняет действие (интерфейс), от того, *как* это делается (реализации).

ПРИМЕР Интерфейс – это, например, кнопки на панели видеомаягнитофона. Реализация – устройство видеомаягнитофона.

Расширения: ограничения, стереотипы.

Ограничения позволяют добавлять новые правила в элементы модели.

Стереотипы позволяют определять новые элементы модели.

	Механизмы расширения UML
Ограничения	Расширяют семантику элемента, обеспечивая возможность добавлять новые правила.
Стереотипы	Обеспечивают возможность определять новые элементы модели UML на основании существующих: мы определяем семантику стереотипа самостоятельно. Стереотипы добавляют новый элемент в метамодель UML.
Помеченные значения	Предоставляют способ расширения спецификации элемента, обеспечивая возможность добавлять в него новую специальную информацию.

Классификация диаграмм.

1. д классов (для статического представления архитектуры системы);
2. д объектов (показывает работу системы в любой момент времени, отображает объекты и взаимосвязи между ними);
3. д вариантов использования (возможные варианты работы с системой с отображением ее функций и пользователей);
4. д взаимодействия (взаимодействие компонентов системы между собой; д последовательностей и д кооперации);
5. д состояний (отображает состояние объекта в процессе работы системы в виде автомата);
6. д деятельности (вариант д состояний, в которой большинство или все состояния явл состояниями деятельности; показывает поток переходов от одной деятельности к другой);
7. д компонентов (показывает организацию набора компонентов и зависимости между ними; обеспечивает статическое представление системы и связана с д классов, т.к. размещает классы по компонентам (модулям));
8. д пакетов;
9. д развертывания (позволяет отобразить механизмы конфигурирования системы, ее внедрения и сопровождения)

Диаграммы классов (объектов).

Диаграмма классов определяет классы системы и различного рода связи, которые существуют между ними (ассоциации, агрегации, композиции, зависимости, обобщения, реализации). На диаграммах классов изображаются также атрибуты классов, операции классов и ограничения, которые накладываются на связи между классами. Классы изображаются в виде прямоугольников, ассоциации – в виде сплошных линий, направления ассоциаций указываются стрелками, агрегации и композиции – в виде сплошных линий с ромбом на конце, связь обобщения – в виде сплошных линий с треугольником на конце, зависимость – в виде пунктирной линии со стрелкой. Роль, возложенная на класс изображается на диаграммах с помощью стереотипов. Класс может быть помечен как граничный (boundary), если он отвечает за взаимодействие с пользователем или внешней системой. Класс-контроллер реализует бизнес-логику приложения. Класс-сущность отвечает за представление данных. Активные классы (процессы или потоки) на диаграмме выделяют с помощью более толстых, чем у обычных классов границ.

Для группировки классов, обладающих некоторой общностью, применяются пакеты.

коммуникационные диаграммы.

Вторым видом диаграмм взаимодействия являются коммуникационные диаграммы. Как и диаграммы последовательности, они отображают поток событий варианта использования. На коммуникационных диаграммах внимание сконцентрировано на соединениях между объектами. Из них легче понять связи между объектами, однако, труднее уяснить последовательность событий. Объекты и/или действующие лица, обменивающиеся сообщениями, связываются линиями (соединениями), над которым в виде стрелок обозначаются сообщения. Нумерация сообщений указывает их последовательность во времени.

Диаграммы взаимодействий

Диаграммы взаимодействия описывают поведение взаимодействующих групп объектов в рамках потока событий. На диаграмме отображается ряд объектов и сообщения, которыми они обмениваются между собой. Сообщение – это средство, с помощью которого объект-отправитель запрашивает у объекта-получателя выполнение одной из его операций. Существует два вида диаграмм взаимодействия: диаграммы последовательности и коммуникационные диаграммы (ранее называемые кооперативными).

13. UML. Диаграммы деятельности. Технология сетей Петри.

Параллелизм. Поток управления, узел действия, ребро, узел управления, объектный узел, буферизация и в объектном узле.

Объектные узлы – параметры, состояния объектных узлов.

Контакты. Прерывающие ребра. Контакты исключений. Потoki объектов. Их аналоги в программировании. Диаграммы состояний.

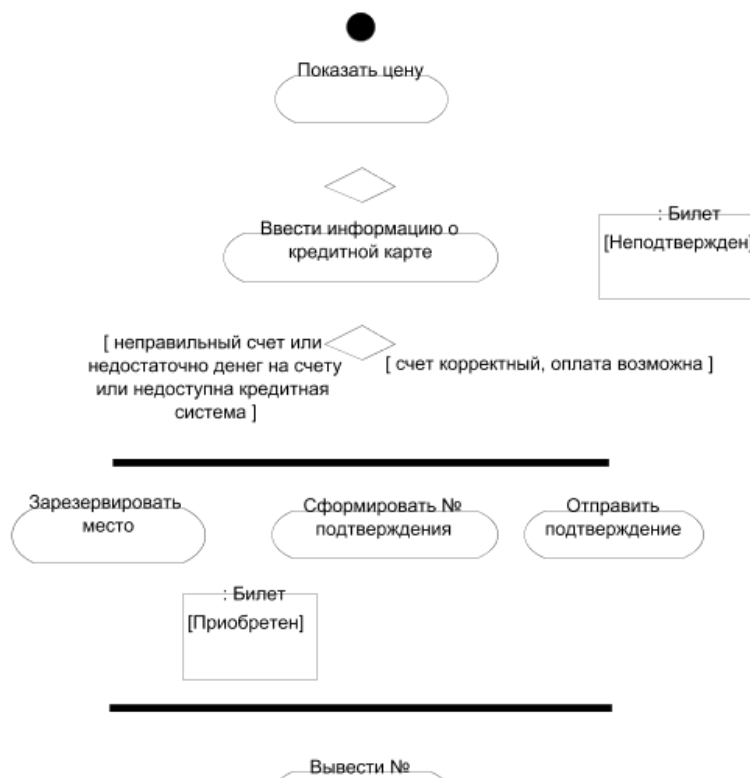
Конечные автоматы.

Диаграммы деятельности полезны в описании поведения, включающего большое количество параллельных процессов. Также их можно применять для представления потоков событий вариантов использования в наглядной графической форме.

В UML 2 проведена граница между диаграммами состояний, базирующимися на формализме конечных автоматов, и диаграммами деятельности, основанными на сетях Петри. Основным элементом диаграмм деятельности является узел действия. Каждый такой узел представляет собой элементарную единицу работы (это может быть решение некоторой задачи, которую необходимо выполнить вручную или автоматизированным способом, или выполнение метода класса). Узел действия изображается в виде закругленного прямоугольника с текстовым описанием. Диаграмма деятельности может иметь входной узел, вообще говоря, не один, (в UML 1 должен быть ровно один), определяющий начало потока управления. Финальный узел необязателен. На диаграмме может быть несколько финальных узлов. На диаграмме могут присутствовать объекты и потоки объектов, в тех случаях, если объект используется или изменяется в одном из узлов действий. Поток объектов отмечается сплошной или пунктирной стрелкой от узла действия к объектному узлу или от объектного узла к узлу действия, использующего объект. Ребра (сплошные стрелки) между узлами действий показывают потоки управления или потоки объектов. Узел разветвления (узел принятия решения), а также узел объединения потоков изображается ромбом. Если необходимо показать, что две или более ветвей потока выполняются параллельно, используются «линейки синхронизации» – узлы разделения и узлы слияния (на рисунке – жирные горизонтальные линии).

Диаграммы деятельности можно рассматривать как вольную трактовку формализма сетей **Петри**. Начальная точка порождает один курсор управления (или маркер) для каждого исходящего перехода. Если переход имеет вес (кратность) больше единицы, то для него порождается столько курсоров, каков его вес. Если для ребра определено событие, то курсор достигает конца ребра только после наступления такого события. Ограничивающее (сторожевое) условие также определяет, возможно ли перемещение курсора по ребру. При попадании курсора в узел действия происходит ожидание курсоров на всех входящих ребрах и лишь потом запускается единица работы. По завершении выполнения работы генерируются курсоры на всех исходящих из узла ребрах.

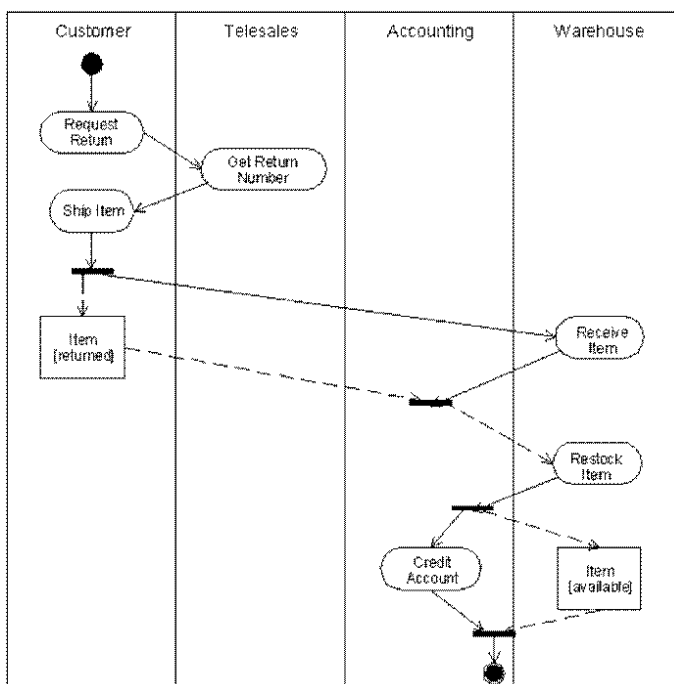
При попадании в узел разветвления (он имеет один вход и несколько выходов) курсор проходит дальше лишь по тому ребру, для которого выполнено сторожевое условие (вообще говоря, лучше, когда оно одно, если несколько – произвольно выбирается единственное для передачи курсора).



При попадании в узел объединения курсор из любого входа копируется на выходе (единственном). При попадании в узел разделения курсор дублируется на все выходы одновременно (происходит распараллеливание). Синхронизация обеспечивается узлом слияния, где происходит ожидание курсоров на всех входах, и лишь затем выдается курсор на выходе.

■ При попадании курсора в любой финальный узел вся деятельность, описываемая диаграммой, прекращается. По тем же правилам перемещаются курсоры объектов. Узлы действия могут иметь входные и выходные контакты для приема/выдачи курсоров объектов, изображаемые в виде квадратиков на границе узла. Если входных контактов несколько, действие в узле выполняется лишь тогда, когда на всех их пришли курсоры объектов. Входные контакты определяют входные параметры деятельности, выходные – аналогично.

Узлы разделения и слияния могут синхронизировать потоки управления с потоками объектов. Например:



Здесь прием возвращаемой позиции заказа на склад синхронизируется с изменением состояния объекта Item, который должен перейти в состояние returned (возвращен). Аналогично, возврат денег по возвращенному заказу синхронизируется со сменой состояния объекта Item на available (доступен). Обратите внимание, диаграмма с помощью вертикальных линий – «плавательных дорожек» разделяется на зоны ответственности (заказчик, продажи, бухгалтерия, склад).

Моделирование **параллельного** (равно как и квазипараллельного) поведения средствами UML сводится к описанию параллельных потоков управления и способов взаимодействия между ними. Средства описания параллелизма в UML отнюдь не противопоставлены средствам описания последовательного поведения, напротив, они образуют единое целое, поскольку параллельное программирование скорее общее правило, нежели экзотическое исключение.

Начнем с обсуждения самого распространенного случая параллелизма. Допустим, что имеются несколько параллельно выполняющих процессов, каждый из которых имеет свой собственный поток управления. Если эти процессы никак не взаимодействуют друг с другом (самый распространенный случай), то нам ничего и не нужно моделировать. Поведение каждого из процессов может быть описано любым из рассмотренных выше способов (конечным автоматом, блок-схемой или диаграммой взаимодействия). Если процессы не взаимодействуют, то они независимы и с точки зрения конечного результата поведения (состояния системы в целом) неважно, как именно реализованы параллельные процессы: как физически параллельные выполняемые на многопроцессорном компьютере, квазипараллельные или последовательно выполняемые в произвольном порядке. Но с точки зрения других аспектов поведения, таких как, например, производительность, время реакции, пропускная способность способ реализации параллелизма является очень существенным.

Диаграммы деятельности UML 2 содержат **узлы**, связанные ребрами; в результате чего образуется полный граф потоков. Управление и значения данных двигаются вдоль ребер и обрабатываются узлами, направляются на другие узлы или временно сохраняются. Более точно, в моделях деятельности имеются три вида узлов.

- **Узлы действия** оперируют получаемыми управляющими воздействиями и значениями данных и предоставляют управление и данные другим действиям.
- **Узлы управления** маршрутизируют перемещение маркеров по графу. Эти узлы содержат конструкции для выбора между альтернативными потоками (точки принятия решения — decision points), для параллельного движения по нескольким потокам (разветвления — forks) и т.д.
- **Объектные узлы** временно удерживают маркеры данных, которые ожидают продолжения движения по графу.

Узлы деятельности связываются направленными ребрами двух типов.

- **Ребра потоков управления** связывают действия. Они обозначают, что действие на целевом конце ребра (со стрелкой) не может начаться до того, как закончится исходное действие. По ребрам потоков управления могут проходить только маркеры управления.
- **Ребра потоков объектов** соединяют узлы объектов для обеспечения действий входными данными. По ребрам потоков объектов могут проходить только маркеры объектов и данных.

На рис. 4 представлена нотация описания ребер. Ребра потоков управления и ребра потоков объектов различаются по своему применению. Ребра управления связывают действия напрямую, тогда как ребра потоков объектов соединяют входы и выходы действий.

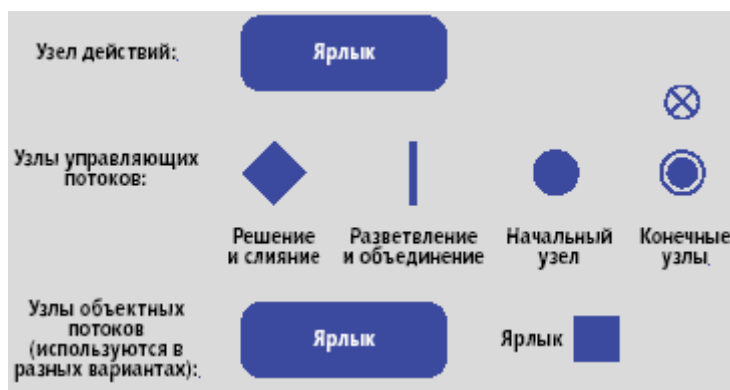


Рис. 3. Узлы деятельности

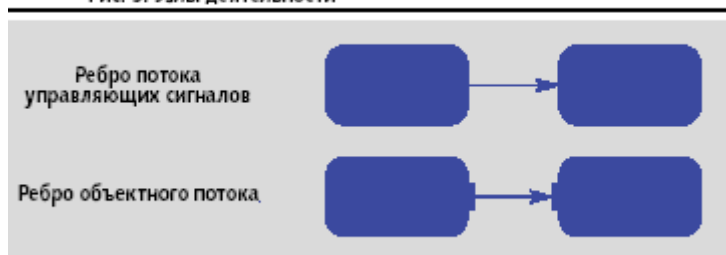


Рис. 4. Ребра деятельности

8.4.3. Входные и выходные **контакты** объектов



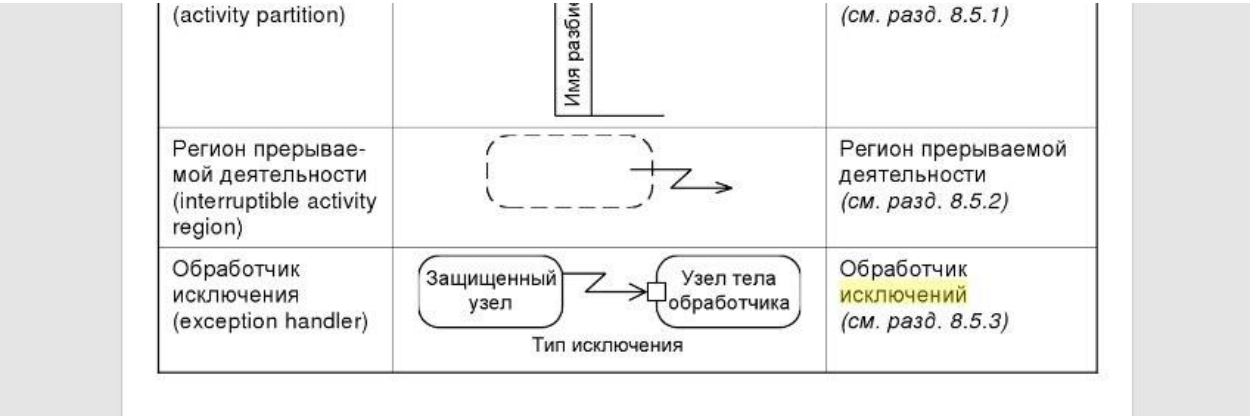
Входной контакт (input pin) является узлом объекта, который принимает значения от других действий в форме потока объектов. *Выходной контакт (output pin)* является узлом объекта, который предоставляет значения другим действиям в форме потока объектов.

прерывающие ребра отображаются в виде зигзагообразной стрелки или обычной стрелки с пиктограммой зигзага над ней.

Области с прерываемым выполнением действий:

- прерываются, когда маркер проходит по прерывающему ребру;
- когда область прерывается, все ее потоки немедленно прекращаются;
- прерывающие ребра отображаются в виде зигзагообразной стрелки или обычной стрелки с пиктограммой зигзага над ней.

Контакт исключений



Поток объектов — это поток, идущий к закреплению или узлу объекта либо в противоположном направлении.

Потоки

С помощью схемы активности можно описывать конвейер или ряд действий, выполняющихся одновременно и постоянно передающих данные от действия к действию. Суть следующего примера в том, что все действия могут создавать объекты и продолжать работу. Так как здесь нет потоков управления, действия могут начинаться сразу после получения первого объекта. Обратите внимание, что соединители в этом примере являются потоками объектов, так как по меньшей мере один конец каждого из них находится в узле параметра действия, узле объекта или закреплении ввода или вывода.

На диаграммах состояний могут применяться псевдосостояния выбора (choice pseudostate) – ромб. Разница между переходным псевдосостоянием и псевдосостоянием выбора в том, что сторожевые условия на стрелках, выходящих из псевдосостояния выбора, вычисляются после выполнения действий на входящих стрелках (в случае с переходным псевдосостоянием все действия совершаются после вычисления условий). В примере диаграмма состояний описывает автомат, ведущий аукцион, который сначала вычисляет прибыль, а потом делает выбор в зависимости от размеров прибыли, в какое состояние перейти.

На переходах из состояния выбора нельзя указывать события. Запрещается использовать сложные переходы с 2-мя и более состояниями выбора.

Диаграммы состояний создают лишь для классов, экземпляры которых имеют сложное поведение, т. е. по-разному обрабатывают одни и те же получаемые сообщения в зависимости от своего внутреннего состояния.

Конечный автомат (state machine) - модель для спецификации поведения объекта в форме последовательности его состояний, которые описывают реакцию объекта на внешние события, выполнение объектом действий, а также изменение его отдельных свойств.

В контексте языка UML понятие конечного автомата обладает дополнительной семантикой. Вершинами графа конечного автомата являются состояния и другие типы элементов модели, которые изображаются соответствующими графическими символами. Дуги графа служат для обозначения переходов из состояния в состояние. Конечный автомат описывает поведение отдельного объекта в форме последовательности состояний, охватывающих все этапы его жизненного цикла, начиная от создания объекта и заканчивая его уничтожением. Каждая диаграмма состояний представляет собой конечный автомат.

Основными понятиями, характеризующими конечный автомат, являются состояние и переход. Ключевое различие между ними заключается в том, что длительность нахождения системы в отдельном состоянии существенно превышает время, которое затрачивается на переход из одного состояния в другое. Предполагается, что в пределе время перехода из одного состояния в другое равно нулю (если дополнительно ничего не сказано). Другими словами, переход объекта из состояния в состояние происходит мгновенно.

В общем случае конечный автомат представляет динамические аспекты моделируемой системы в виде ориентированного графа, вершины которого соответствуют состояниям, а дуги - переходам. При этом поведение моделируется как последовательное перемещение по графу состояний от вершины к вершине по связывающим их дугам с учетом их ориентации. Для графа состояний системы можно ввести в рассмотрение специальные свойства.

14. Экстремальное и гибкое программирование. Манифест экстремального программирования (XP). Гибкие (agile) технологии. SCRUM. Agile UP, ICONIX.

Гибкая методология разработки ([англ. Agile software development, agile-методы](#)) — серия подходов к [разработке программного обеспечения](#), ориентированных на использование [итеративной](#) разработки, динамическое формирование требований и обеспечение их реализации в результате постоянного взаимодействия внутри самоорганизующихся рабочих групп, состоящих из специалистов различного профиля^[1]. Существует несколько методик, относящихся к классу гибких методологий разработки, в частности [экстремальное программирование](#), [DSDM](#), [Scrum](#), [FDD](#).

Применяется как эффективная практика организации труда небольших групп (которые делают однородную творческую работу) в объединении с управлением ими комбинированным (либеральным и демократическим) методом.

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, 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. Большинство agile-команд расположены в одном офисе, иногда называемом [англ. bullpen](#). Как минимум, она включает и «заказчиков» ([англ. product owner](#) — заказчик или его полномочный представитель, определяющий требования к продукту; эту роль может выполнять менеджер проекта, бизнес-аналитик или клиент). Офис может также включать тестировщиков, дизайнеров интерфейса, технических писателей и менеджеров.

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.

Экстремальное программирование (eXtreme Programming, XP) — облегченный (подвижный) процесс (или методология), главный автор которого — Кент Бек (1999) [11]. 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* 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.

Манифест

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

Ценности

Люди и взаимодействие важнее процессов и инструментов;

1. **Работающий продукт** важнее исчерпывающей документации;
2. **Сотрудничество с заказчиком** важнее согласования условий контракта;

3. Готовность к изменениям важнее следования первоначальному плану.

Таким образом, не отрицая важности того, что справа, всё-таки больше ценится то, что слева.

1. Наивысшим приоритетом является удовлетворение потребностей заказчика, благодаря регулярной и ранней поставке ценного программного обеспечения;
2. Изменение требований приветствуется, даже на поздних стадиях разработки;
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. Чтобы работа была сделана, создайте условия, обеспечьте поддержку и полностью доверьтесь 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;
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.^[3]

Гибкая методология разработки ([англ. Agile software development, agile-методы](#)) — серия подходов к [разработке программного обеспечения](#), ориентированных на использование [итеративной разработки](#), динамическое формирование требований и обеспечение их реализации в результате постоянного взаимодействия внутри самоорганизующихся рабочих групп, состоящих из специалистов различного профиля. Существует несколько методик, относящихся к классу гибких методологий разработки, в частности, известны как гибкие методики [экстремальное программирование](#), [DSDM](#), [Scrum](#).

Scrum ([/skrʌm/](#)^{[1][2]}; [англ. scrum «схватка»](#)) — методология гибкой разработки ПО.

Методология делает акцент на качественном контроле процесса разработки.

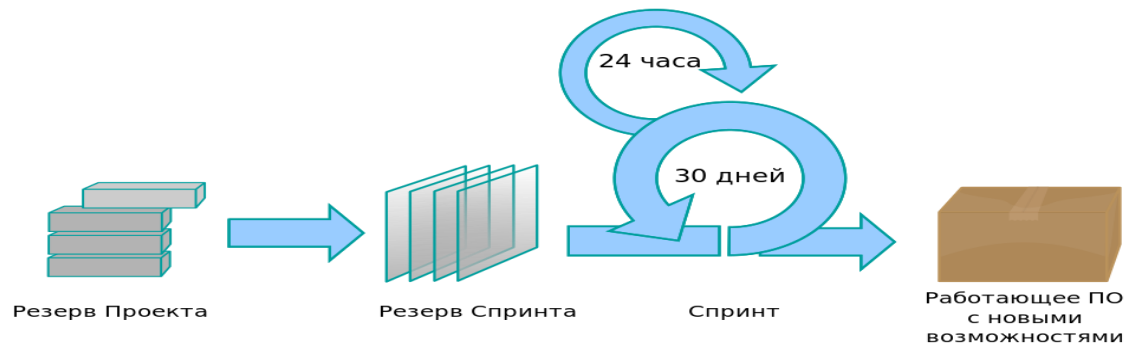
Кроме управления проектами по разработке ПО, Scrum может также использоваться в работе команд [поддержки программного обеспечения](#), или как подход к управлению разработкой и сопровождению программ: *Scrum of Scrums*.

Scrum — это набор принципов, на которых строится процесс разработки, позволяющий в жёстко фиксированные и небольшие по времени итерации, называемые спринтами (sprints), предоставлять [конечному пользователю](#) работающее ПО с новыми возможностями, для которых определён наибольший приоритет. Возможности ПО к реализации в очередном спринте определяются в начале спринта на этапе планирования и не могут изменяться на всём его протяжении. При этом строго фиксированная небольшая длительность спринта придаёт процессу разработки предсказуемость и гибкость.

Спринт

Спринт^[12] — итерация в скраме, в ходе которой создаётся функциональный рост программного обеспечения. Жёстко фиксирован по времени. Длительность одного спринта от 2 до 4 недель. В отдельных случаях, к примеру согласно скрам-стандарту компании [Nokia](#), длительность спринта должна быть не более 6 недель. Тем не менее, считается, что чем короче спринт, тем более гибким является процесс разработки, релизы выходят чаще, быстрее поступают отзывы от потребителя, меньше времени тратится на работу в неправильном направлении. С другой стороны, при более длительных спринтах команда имеет больше времени на решение возникших в процессе проблем, а владелец проекта уменьшает издержки на совещания,

демонстрации продукта и т. п. Разные команды подбирают длину спринта согласно специфике своей работы, составу команд и требований, часто методом проб и ошибок. Для оценки объёма работ в спринте можно использовать предварительную оценку, измеряемую в очках истории. Предварительная оценка фиксируется в бэклоге проекта.



Методология Agile Unified Process (Гибкая методология разработки)

Это упрощенная схема IBM Rational Unified Process. AUP состоит из перечня методов:

1. Моделирование в первую очередь используется для понимания бизнес требований, а также предметной области.
2. Реализация – это непосредственное преобразование модели в исполняемый код, который также имеет и модульные тесты.
3. Тестирование представляет собой способ поиска дефектов и верификации системы на предмет соответствия требований.
4. Размещение – это доставка уже готовой/законченной системы пользователю.
5. Управление конфигурациями – это управление доступом и версиями артефактов проекта.
6. Управление проектом – это непосредственные активности, связанные с ходом проекта. В частности управление и координация людей, управление рисками, финансами и т.д.
7. Под средой понимают совокупность процессов, инструментов, стандартов и правил.



Данная методология разработки программного обеспечения сфокусирована на анализе требований, а также моделировании. В рамках ICONIX используется подмножество UML для анализа требований. В частности диаграмма вариантов использования, диаграмма классов, диаграмма робастности и диаграмма последовательности.

Методология ICONIX представляет собой нечто среднее между очень громоздким рациональным унифицированным процессом (Rational Unified Process - RUP) и весьма компактной методологией программирования extreme (XP). Процесс ICONIX, как и RUP, основан на прецедентах, но не характеризуется множеством его недостатков. В этом процессе тоже применяется язык моделирования UML (Unified Modeling Language), однако основное внимание уделяется анализу требований. Отметим еще раз, что ICONIX, по представлению Айвара Джекобсона (Ivar Jacobson), «основан на прецедентах», вот почему с помощью этого процесса создаются вполне конкретные и простые для понимания прецеденты, которые легко использовать для разработки системы.

15. SCRUM как технологический фреймворк. Терминология. Спринт. Митинг. Собственник проекта. Команда. SCRUM-мастер. Беклог проекта и спринта. Планирование спринта. Диаграмма сгорания. Оценка трудоемкости. Покер-планирование.

Scrum — методология управления проектами, активно применяющаяся при разработке информационных систем для гибкой разработки программного обеспечения. Scrum чётко делает акцент на качественном контроле процесса разработки. Скрам содержит набор принципов, на которых строится процесс разработки, позволяющий в жёстко фиксированные и небольшие по времени итерации, называемые спринтами (sprints), предоставлять конечному пользователю работающее ПО с новыми возможностями, для которых определён наибольший приоритет. Возможности ПО к реализации в очередном спринте определяются в начале спринта на этапе

планирования и не могут изменяться на всём его протяжении. При этом строго фиксированная небольшая длительность спринта придаёт процессу разработки предсказуемость и гибкость.

Спринт — итерация в скраме, в ходе которой создаётся функциональный рост программного обеспечения. Жёстко фиксирован по времени. Длительность от 2 до 4 недель. Считается, что чем короче спринт, тем более гибким является процесс разработки, релизы выходят чаще, быстрее поступают отзывы от потребителя. Разные команды подбирают длину спринта согласно специфике своей работы, составу команд и требований, часто методом проб и ошибок. На протяжении спринта никто не имеет права менять список требований к работе, внесенном в резерв спринта.

Митинг - начинается точно вовремя; • все могут наблюдать, но только скрам команда может говорить(не владелец продукта); • длится не более 15 минут; • проводится в одном и том же месте в течение спринта. В течение совещания каждый член команды отвечает на 3 вопроса: • Что сделано с момента предыдущего ежедневного совещания? • Что будет сделано с момента текущего совещания до следующего? • Какие проблемы мешают достижению целей спринта?

Скрам мастер - проводит совещания (Scrum meetings) следит за соблюдением всех принципов скрам, разрешает противоречия и защищает команду от отвлекающих факторов. Данная роль не предполагает ничего иного, кроме корректного ведения скрам-процесса. Руководитель проекта скорее относится к владельцу проекта и не должен фигурировать в качестве скрам-мастера.

Беклог проекта — это список требований к функциональности, упорядоченный по их степени важности, подлежащих реализации. Элементы этого списка называются «историями пользователя» (user story) или backlog items. Резерв проекта открыт для редактирования для всех участников скрам процесса.

Беклог спринта — содержит функциональность, выбранную владельцем проекта из беклога проекта. Все функции разбиты по задачам, каждая из которых оценивается скрам-командой. Каждый день команда оценивает объем работы, который нужно проделать для завершения спринта.

Планирование спринта - Происходит в начале новой итерации Спринта. Из беклога проекта выбираются задачи, которые будут сделаны за спринт -> создается беклог спринта. Каждая задача оценивается в идеальных человеко-часах; Решение задачи не больше 12 часов или 1 дня, иначе разбивается на подзадачи. 2 этапа – первый этап – выбор задач, второй этап - обсуждение технических деталей реализации.

Диаграмма сгорания - Диаграмма отображает оставшиеся нерешенные задачи и трудозатраты, необходимые для их завершения в расчете на 21 рабочий день. Обновляется ежедневно с тем, чтобы в простой форме показать подвижки в работе над спринтом. График должен быть общедоступен.

Оценка трудоемкости — Абстрактная метрика оценки сложности истории, которая не учитывает затраты в человеко-часах. story points — это относительные оценки объёма работы в истории. Нет способа оценить одну единственную историю в story points, вы всегда сравниваете историю с другими историями через story points. Когда команда оценивает много историй в течении короткого промежутка времени, используется **покер планирование**. В нем используются несколько колод с числами фиббоначи 0.5,1,2,3,5,8,13,21 ... Каждая задача оценивается членами команды числами из колоды, которые означают сложность задачи в story points.

16) Оценка программного кода. Метрики кода. Метрики количественные, сложности потока управления и потока данных, метрики ООП, прагматические метрики. Средства оценки качества программного кода.

Количественные метрики

- количество строк кода,
- количество пустых строк,
- количество комментариев,
- процент комментариев (отношение числа строк, содержащих комментарии к общему количеству строк, выраженное в процентах),
- среднее число строк для функций (классов, файлов).

Также к группе метрик, основанных на подсчете некоторых единиц в коде программы, относят метрики Холстеда. Данные метрики основаны на следующих показателях:

n1 — число уникальных операторов программы, включая символы-разделители, имена процедур и знаки операций (словарь операторов),

n2 — число уникальных операндов программы (словарь операндов),

N1 — общее число операторов в программе,

N2 — общее число операндов в программе,

n1' — теоретическое число уникальных операторов,

n2' — теоретическое число уникальных операндов.

Еще одним типом метрик ПО, относящихся к количественным, являются метрики Джилба. Они показывают сложность программного обеспечения на основе насыщенности программы условными операторами или операторами цикла.

Метрики сложности потока управления программы

Цикломатическая сложность. При вычислении цикломатической сложности используется граф потока управления программы. Узлы графа соответствуют неделимым группам команд программы, они соединены ориентированными рёбрами, если группа команд, соответствующая второму узлу, может быть выполнена непосредственно после группы команд первого узла. Цикломатическая сложность части программного кода — количество линейно независимых маршрутов через программный код.

Модификация метода Мак-Кейба — метод Хансена. Мера сложности программы в данном случае представляется в виде пары (цикломатическая сложность, число операторов). Преимуществом данной меры является ее чувствительность к структурированности ПО.

Метрики сложности потока управления данными

Метрика Чепина: суть метода состоит в оценке информационной прочности отдельно взятого программного модуля с помощью анализа характера использования переменных из списка ввода-вывода.

Все множество переменных, составляющих список ввода-вывода, разбивается на 4 функциональные группы :

1. Р — вводимые переменные для расчетов и для обеспечения вывода,
2. М — модифицируемые, или создаваемые внутри программы переменные,
3. С — переменные, участвующие в управлении работой программного модуля (управляющие переменные),

4. Т — не используемые в программе («паразитные») переменные.

Метрика Чепина :

$$Q = a_1 * P + a_2 * M + a_3 * C + a_4 * T,$$

где a_1, a_2, a_3, a_4 — весовые коэффициенты.

Еще одна метрика, учитывающая сложность потока данных — это метрика, связывающая сложность программ с обращениями к глобальным переменным.

Пара «модуль-глобальная переменная» обозначается как (p, g) , где p — модуль, имеющий доступ к глобальной переменной g . В зависимости от наличия в программе реального обращения к переменной g формируются два типа пар: фактические и возможные.

Возможное обращение к g с помощью p показывает, что область существования g включает в себя p .

Характеристика A_{up} говорит о том, сколько раз модули U_p действительно получали доступ к глобальным переменным, а число P_{up} — сколько раз они могли бы получить доступ.

Отношение числа фактических обращений к возможным определяется:

$$R_{up} = A_{up} / P_{up}.$$

Эта формула показывает приближенную вероятность ссылки произвольного модуля на произвольную глобальную переменную. Очевидно, что чем выше эта вероятность, тем выше вероятность «несанкционированного» изменения какой-либо переменной, что может существенно осложнить работы, связанные с модификацией программы.

Метрики ООП

Практически каждый класс имеет группу классов, с которыми он работает в кооперации, и от которых он не может быть легко отделен. Для повторного использования таких классов необходимо повторно использовать всю группу классов. Такая группа классов сильно связана и называется категорией классов.

Могут быть определены три метрики :

1. C_a : Центростремительное сцепление. Количество классов вне этой категории, которые зависят от классов внутри этой категории.

2. C_e : Центробежное сцепление. Количество классов внутри этой категории, которые зависят от классов вне этой категории.

3. I : Нестабильность: $I = C_e / (C_a + C_e)$. Эта метрика имеет диапазон значений $[0, 1]$.

$I = 0$ указывает максимально стабильную категорию.

$I = 1$ указывает максимально не стабильную категорию.

Можно определять метрику, которая измеряет абстрактность (если категория абстрактна, то она достаточно гибкая и может быть легко расширена) категории следующим образом:

A : Абстрактность: $A = n_A / n_{All}$.

n_A — количество_абстрактных_классов_в_категории.

n_{All} — общее_количество_классов_в_категории.

Значения этой метрики меняются в диапазоне $[0, 1]$.

0 = категория полностью конкретна,

1 = категория полностью абстрактна.

